Object-Connectivity and Observability for Class-Based Object-Oriented Languages

Habilitationsschrift

eingereicht bei der
Technischen Fakultät der
Christian-Albrechts-Universität zu Kiel

von

Martin Steffen

aus Bad Honnef

Kiel, 4. Juli 2006

Summary

This habilitation thesis investigates observability for object-oriented, *class-based* languages. Notably, we consider classes and heap-allocated objects, imperative field update with aliasing, and a thread-based model of execution, both for a single-threaded, deterministic, and for a multi-threaded language. The choice of features is inspired by modern, object-oriented languages such as *Java* or $C^{\#}$. Simplifying, the work answers the question:

> *What can be* observed *when programs are structured in* classes?

Observational equivalence equates two program phrases when no context exists able to differentiate between them. In the simplest case, typical for sequential settings, the observer checks for convergence, or more generally, reachability of a predefined point.

The contextual definition of progam equivalence is natural, straightforward, and fundamental. It does not, however, answer what the meaning of a program actually *is.* A denotational semantics explicitly assigns meaning to the program phrases, and this gives a second answer as to when two programs are equivalent, namely when they have the same denotation. The coincidence of the two notions of equivalence is called *full abstraction.*

The problem of full abstraction has been recognized as fundamental for the study of program semantics and, consequently, has been investigated in various settings. The key to the contextual definition is that the context or observer is programmed in the language itself. Hence, the language used and its constructs influence the notion of observable equivalence and a, consequently, fully abstract semantics gives insight into the nature of the language constructs at hand. We take particular interest in classes as constructs.

So, returning to the above question, the simplified answer is:

> *With program and context given by* classes, *an approximation of the* heap-structure *becomes part of the semantics.*

This answer can be analyzed as follows:

Cross-border instantiation and heap abstraction: Separating component and context classes make *instantiation* a possible interaction between component and context. Consequently, the environment can create objects which are *unconnected* to the rest of the component's heap. Vice versa, the component can create separate environment objects.

This separation of the heap must be taken into account for a fully abstract semantics and is the key semantical consequence of classes. The mentioned abstract representation of the heap-structure over-approximates the heap in that it formalizes the potential acquaintance or connectivity of objects: $o_1 \hookrightarrow o_2$ asserts that object o_1 potentially contains a reference to o_2. The potential connectivity partitions the heap into equivalence classes, which we call *cliques.* This connectivity is dynamic, in that new cliques of objects may be *created* via instantiation, and previously separate cliques of objects may *merge* by communication.

Separate observers and order of events: If the environment or observer is split into separate cliques of objects, it looses some power of observation, e.g., the absolute order of events interacting with the observers cannot be determined. This gives rise to a *tree-structured* semantics.

Classes as generators of objects and replay: Classes are generators of objects. Hence, two instances of a class are *"identical up-to their identity"*, i.e., they have the same behavior up-to renaming. This, on the other hand, increases the observational power of the environment in that, in one single experiment, it can create more than one instance of a class, and observe their behavior.

As mathematical vehicle for our analysis, we use a strongly-typed object calculus extended by classes. We define a trace-based semantics where the observable semantics of a component is based on traces, i.e., sequences of calls and returns exchanged with the environent. Traces are considered up to unobservability due to separate observers (giving rise to a tree-like representation), up to renaming of object identities, and up to replay.

For the semantics, we establish full abstraction wrt. a may-testing preorder, both in a *single-threaded* and a *multi-threaded* setting. This semantics is the first such result for class-based languages and cross-border instantiation.

As usual, the completeness part of the full abstraction result —denotational equivalence implies observational equivalence— is a constructive argument. In our setting, the argument has the following form: Given a trace, construct a program that exactly realizes this trace (up to the unavoidable imprecision of the semantics). Interestingly, the constructions for the single-threaded and the multi-threaded case are largely identical. The only significant additional programming task in the concurrent setting is to assure mutual exclusion, basically by implementing synchronized methods and a simple form of (non-reentrant) monitors in the calculus.

Contents

CHAPTER 1

Introduction

The first part of the thesis contains some introductory material, surveying some related work, and trying to convey the main intuitions of the thesis, without going into technical details.

1.1 Object-oriented programming

An established, major paradigm in programming and the design of programming languages is object-orientation. Indeed so much so that someplace "modern programming language" and "object-oriented programming language" seem to be taken as synonyms: There are old data-base languages, and there are object-oriented ones, there are some failed design methods, and object-oriented ones, etc.... Whether one agrees or not, the widespread acceptance of languages such as C^{++} [133], *Java* [65], and $C^{\#}$ [50] indicates that features offered by those languages are helpful in programming and structuring real-life software.

Central for object-oriented languages is, not surprisingly, the notion of *object*, a unit bundling together a state plus methods for querying and updating the (encapsulated) state. Objects interact and "communicate" via method calls, i.e., by message passing. They can be created or instantiated on demand and have a unique *identity*, their "self". They are heap-allocated and referenced by their identity.

Structuring the program state in form of the heap into encapsulated objects does not imply that the *code* is structured into objects, as well. Indeed, it is characteristic for many current object-oriented languages, in particular the ones mentioned above, that the code is structured into *classes*, which serve as blueprint for their instances, the objects.[1] For a deeper comparison and discussion of object-based vs. class-based languages see [37] and also [2].

1.2 Object calculi

In the same way as λ-calculi [26] form the core of sequential languages and various process calculi [28] have been devised to capture the essence of communication and concurrency, object calculi have been proposed as mathematical core of object-oriented languages.[2] They allow to study concepts and core features in a clean, mathematical way. Whereas λ-calculi are designed around the notion of function, process algebras around the notion of process, *objects* are the basic ingredients of object calculi. A standard reference is Abadi and Cardelli's [2]. See [40] for a recent account of object calculi dealing with parallelism, concurrency, distribution, and mobility.

For our semantical study, we take one particular object calculus as starting point, namely Gordon and Hankin's concurrent object calculus from [62], also used by Jeffrey and Rathke in [82]. Being interested in classes, we add the corresponding constructs, yielding a calculus offering the following key features:

- classes as structuring concept,
- objects as instances of classes,
- references and aliasing, and a

[1] Besides as generators of objects, classes often also play also the role of the *type* of its instance. Whereas this can seem advantageous from the perspective of economy of concepts, arguments against this identification can be put forward [41].

[2] The terms "object-based" and "object-oriented" are sometimes used to distinguish between two flavors of languages with objects: object-oriented languages, in this manner of speaking, support classes and inheritance, whereas object-based languages do without classes. Instead, they offer more complex operations on objects, for instance, general method update.

- multi-threading model of concurrency.

We are especially interested exploring the semantical consequences of *classes* in an object-oriented setting. To do so, we take a (standard) *observational* approach to semantics.

1.3 Semantics, observability, and full abstraction

Semantics addresses the question of "what it means". For computer science in general (and logics and/or mathematics), and in particular for programming languages, the answer is a mathematical one. Fundamental as the question of meaning is,[3] the answer is by no means unequivocal. Although, the distinction between the various semantical approaches is not clear-cut, one distinguishes roughly the following main flavors: Operational semantics sees a program as something that "runs" or evolves and concentrates on the change of configuration during execution. In particular, using inference rules to justify execution steps has proven a versatile, concise, and often straightforward mathematical tool. This structural operational semantics ("SOS") has been proposed by Plotkin [117]. The denotational approach, in contrast, explains a program phrase by mapping it to an independent mathematical domain. Finally, the term "axiomatic" is sometimes used for semantics à la Hoare [72], where the meaning of a program fragment is specified, in the form of pre- and post-conditions and respective rules, by its effect on the program state

A natural approach is not to start from the (hard) question what the meaning of a program or construct *is,* but what can be *seen* from the outside. Agreement on what is *observable* immediately answers when two programs are equivalent, namely, when no observation can tell them apart. So the client, from a practitioner's point of view, could insist: "I couldn't care less what the meaning of this new version of the component is, denotational or what have you, just make sure that when I *use* it in my programs in place of the current implementation, which is known to work, *nothing changes.*" Important is the black-box view, i.e., to look at the program from the outside; the observation "I can see that the programmer used a variable x in line 1753" is not interesting.

The way observations are done should not depend on the eyes or the mind of a human observer or some other additional definition. This leads to a *contextual* definition, where the observer (or context) is itself a program in the given language. Two programs P_1 and P_2 are equivalent if they can not be discriminated in the following sense:

> for *all* contexts $\mathcal{C}[\_]$, letting $\mathcal{C}[P_1]$ and $\mathcal{C}[P_2]$ run, one sees no difference,

where $\mathcal{C}[P]$ means the closed program consisting of P and the "rest" $\mathcal{C}[\_]$ (the context, the observer, the environment).

Here, we have cheated, obviously, in two points: still, (1) what does it mean to let a closed program $\mathcal{C}[P]$ *run,* and (2) what does one *see* about $\mathcal{C}[P]$ when it runs. The setting, however, has now become considerably easier, as $\mathcal{C}[P]$ is a

[3] Fundamental also in a mundane and practical sense: Without a reasonably clear account of what the meaning of the program is, how should one be able to program, let alone reason about the program, argue for its functionality, etc.?

closed program, as opposed to P. For describing the behavior of $\mathcal{C}[P]$, we can choose whichever semantical description seems most appropriate or easiest; this amounts to an operational semantics, in most cases. Also observability for (2) becomes a matter of conscious choice or specification. The simplest possible external observation about a closed program is that it halts or converges, written $\mathcal{C}[P] \Downarrow$. For sequential programs, termination is, indeed, *the* crucial observation; the resulting equivalence, known as "*observable equivalence*" has been introduced by Morris [107] for a call-by-name λ-calculus.

For concurrent programs, the idea requires a small amount of refinement, as termination is no longer a useful criterion to distinguish programs: Processes or reactive programs are often not supposed to terminate. Instead, the observer runs in parallel with the program under observation, typically interacting via message exchange. From the outside it is seen whether both reach a defined point (written $\mathcal{C}[P] \Downarrow_{succ}$) witnessed by a predefined communication, here called "success". In a non-deterministic setting and when comparing two processes wrt. their successfulness confronted with all possible observers, one distinguishes necessary and potential success, leading to must, resp., may testing equivalence. We write $\sqsubseteq_{may}$ for the corresponding may pre-order (one program yields success together with each observer that reports success together with the second program). The important notion of testing equivalence has been introduced by de Nicola and Hennessy [108].

The contextual approach gives a convincing, abstract, definition of when two programs are equivalent, but does not tell what actually the denotation of a program *is.* The quantification over all possible contexts gives the contextual definition its strength and simplicity. It makes it hard, however, to apply, when proving equivalence of two programs. For that purpose, an explicit denotation is better. Given both an implicit, contextual, and an explicit, denotational semantics, their coincidence is called *full abstraction* [101][116]: Two programs are observationally equivalent iff they have the same denotation. Let us write $\equiv_{obs}$ for observational and $\equiv_{\mathcal{D}}$ for denotational equivalence. The denotational semantics is an abstraction of the actual program, as it ignores internals of the code; for instance, representational concomitants —the fact that variable x appears at line 1753, the names of local variables— as well as internal execution steps will not be part of the semantics but abstracted. With the observational definition as reference, the denotational semantics is sound, if $P_1 \equiv_{\mathcal{D}} P_2$ implies $P_1 \equiv_{obs} P_2$. The inverse implication, hence "full" abstraction, corresponds to completeness. Having a fully abstract semantics, and not just a sound one, is useful since it allows to reason abstractly over programs, i.e., *all* properties of the program which are valid wrt. the notion of observation can be derived from the denotational semantics. Starting with Milner and Plotkin, the issue of full abstraction has been addressed from many angles and for many different language features. As mentioned, we investigate in this work an *object-oriented* calculus, in particular stressing the roles of classes. We refer to Section 6.2 in the conclusion for a discussion of related work in the area.

1.4 Components, objects, and classes

The notion of *component* is well-advertised as structuring concept for software development. Even if there is little agreement on what constitutes a "compo-

nent" in concrete software engineering terms, one thing is for sure: Components are intended for *composition.* This corresponds to the observational point of view, as discussed: Two components are observably equivalent, when no observing context can tell them apart. At the core is therefore the separation of a program into the component under observation and the environment or context or observer,[4] both programmed in classes.

This section presents on an intuitive level the consequences of incorporating *classes* into the observational set-up. Leaving aside sub-classing and subtyping, a class is nothing else than a generator of objects, i.e., it serves as a blueprint for its instances.

1.4.1 Cross-border instantiation and connectivity

The observational set-up separates *classes* into component and environment classes. Hence, not only calls and returns are exchanged at the interface between component and environment, but instantiation requests, as well.

If, for instance, the component creates an instance of an environment class, the interaction between the component and the newly created object can entail observable effects in the future, as the code of the object is externally provided and therefore this interaction belongs to the externally visible observer-program behavior. Hence, instances of environment classes belong to the environment, and, dually, those of internal classes to the component. To be more concrete, we illustrate the idea using *Java*-syntax.

Example 1.4.1. *Consider the following piece of Java-code:*

Listing 1.1: External class

```
public class P {  // component
    public static void main(String[] arg) {
        O x = new O();
        x.m(42);  // call to the instance of O
    }
}

class O {        // external  observer
    public void  m(int x) {
                 // body of m
    }
}
```

Class P is the component and O the environment or observer.[5] *The program instantiates one object of class O and calls its method m, and passes an integer as argument. The program fragment P is considered as black box, but O is in the hand of the experimenter which can use it for observations. The success-report from the may-testing set-up, mentioned shortly in Section 1.3, can be given here simply by printing* `''success''` *to standard-out. Clearly, the observer can see in the mentioned*

[4] In a game theoretical approach to semantics, one also speaks of player and opponent [18].

[5] For concreteness sake, we use actual *Java*-syntax. In the example, the program starts in the static main method of class P. The calculus later will not have static methods. In general, the examples in this section are given as concrete, executable *Java*-programs or fragments of programs. In the text we allow ourselves to refer to variables or identifiers of the form x1 ...by x_1 .... The connection should be clear in all cases.

sense whether P calls m, since it can fill in the method body by an appropriate print-instruction. Likewise, it can observe the value that P sends, in this case 42. Consequently, the call of m, including the integer argument, belongs to P's external behavior, and furthermore, the instance of the external class O belongs to the observer. □

Even if the instance of the environment class belongs to the environment, as well, the *reference* to the new external object is kept at the creator for the time being. So if the component instantiates two objects o_2 and o_3 of the environment, the situation looks informally as in Figure 1.1, where the dotted bubbles indicate the scope (the "area" within which the object is known) of o_2, respectively of o_3, after creation.

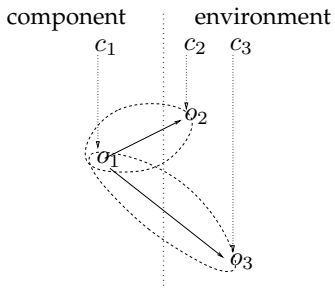


Figure 1.1: Instances of external classes

Clearly, in that situation, the component can control whether the two object can contact each other, irrespective of the implementation of the environment. An exact representation of the semantics must account for the inability of o_2 and o_3 to be in contact. More generally, the semantics must contain a representation of which object can possibly be in contact with others, i.e., an over-approximation of the heap's *connectivity.* Sets of objects which can possibly be in contact with each other form therefore equivalence classes of names —we call them *cliques*— and the semantics must represent them. New cliques can be created, as new objects can be instantiated without contact to others, and furthermore cliques can merge, when the component leaks the identity of a member of one clique to a member of another.

Thus, the component semantics must keep track of which objects of the environment are connected. The component has, of course, by no means full information about the complete system; after all, it can at most trace what happens at the interface, and the objects of the environment can exchange information "behind the component's back". Hence, the component conservatively over-approximates the potential acquaintance of objects in the environment and makes *worst-case assumptions* concerning the proliferation of knowledge, which means it assumes that:

1. Once a name is out, it is never forgotten.

2. If there is a possibility that a name is leaked from one environment object to another, this will happen.

1.4.2 Different observers and order of events

That the observer may fall into separate cliques of unconnected objects has implications for what can be observed. First of all, the absolute *order of events* cannot be determined, as separate observer cliques are not able to coordinate. For instance, the environment or observer, split into the cliques $[o_1]$ and $[o_2]$ on the right-hand sides of the three scenarios of Figure 1.2 cannot distinguish between the three variants of the component on the respective left-hand sides, as explained below. Note that the clique structure is dynamic, since commu-

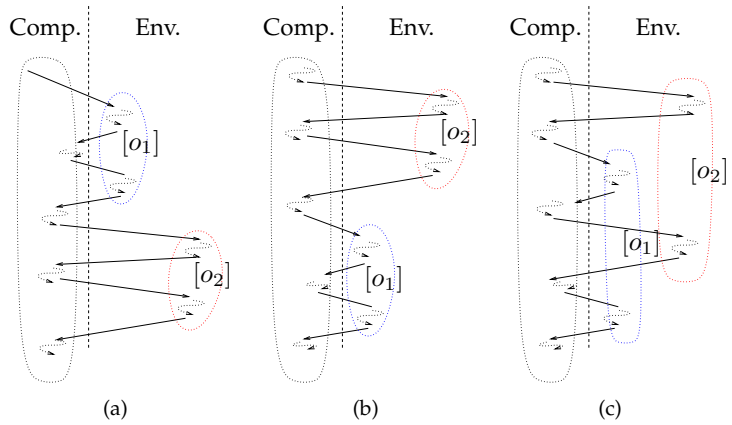


Figure 1.2: Order of interaction

nication can merge previously separate observer cliques. After merging, the now joint clique, indicated by the big "bubble" containing $[o_1]$ and $[o_2 2]$ in Figure 1.3, can coordinate and thus observe the order of further interaction, but, in general, the order of past interaction cannot be reconstructed. In other words, in Figure 1.3, the three components, i.e., the components on the left-hand side of Figure 1.3(a) – 1.3(c) respectively, are observably equivalent.

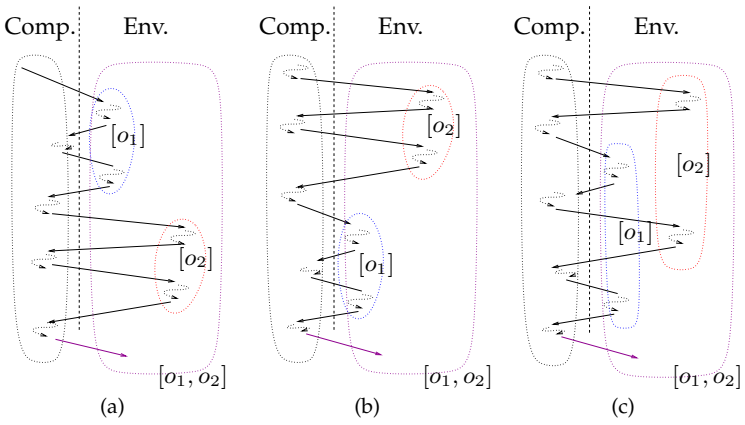


Figure 1.3: Order of interaction and merging

The following example illustrates the phenomenon using *Java*-code. To

mimic the may-testing framework, we adopt as notion of observation whether or not the observer together with the program prints out the single and predefined message `"success"`[6]

Example 1.4.2 (Order of events)**.** *Consider Listing 1.2, where the component creates two observers,*[7] *kept in x_1 and x_2.*

Listing 1.2: Order of events

```
public class P {                                    // component
    private static int x  = 0;
    public static void main(String[] arg) {
        O x1 = new O(); O x2 = new O();
        x1.m1();
        x2.n();
        x1.m2();
    }
}

class O {                                           // environment
    public void m1() { }
    public void n() { }
    public void m2() {
        System.out.println("success");
    }
}
```

The interaction with the first observer consists of calling m_1 and m_2, and with the second one of calling method n. As the two observer instances are separate, the component P as shown cannot be distinguished from a variant P', where the calls $x_1.m_1()$ and $x_2.n()$ are executed in swapped order*. Especially it cannot be distinguished by the observing environment given by class O.* □

Note that the phenomena discussed are a consequence of fact that the language is class-based. In an object-based setting and thus in absence of separate cliques of objects, they are not present.

The observer is programmed using classes, the observation —here the printing of "success"— is done by instances of the observer classes. Note that having more than one independent observer, especially having more than one instance of the observer classes, does not mean that *all* have to report success; only the first one counts. Consequently, interactions with separate observers cannot be observed in a single experiment and interactions which do not contribute to the success-reporting observer, may "fall under the table". Consider Figure 1.4(a): The component C_1 on the left interacts with an observer which falls into two cliques, the second of which reports success. Replacing C_1 by C_2 in Figure 1.4(b), which interacts shorter with the first observer clique, does not change the success of the experiment with the given observer. To pose the question more generally: For any possible observing environment O, if C_2 together with O can report success each time success can be reported by C_1 with O, what does this imply for C_1, i.e., given C_1 and $C_1 \sqsubseteq_{may} C_2$, what do we

[6] More precisely, we additionally assume that this message is unique, in that no one else than the observer can produce it, i.e., the program cannot fake success.

[7] Unlike in Example 1.4.5, the fact that we use here a single class O as template for the two observers is accidental and due to the fact that it allows more compact code; nothing would change, if we used two classes, instead.

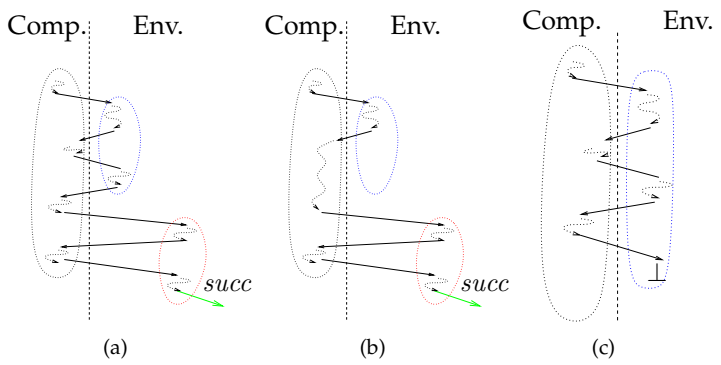


Figure 1.4: Two observers

know about C_2? Let us call s_1 the global trace of C_1 at the interface with the observers, with

$$s_1 = s_1^1 \, s_1^2 \, ,$$

where s_1^1 denotes the projection of the trace onto the first observer clique and s_1^2 onto the second. The exact nature of the single interactions, whether they are calls or returns, for instance, is of no import here.

Continuing the discussion with s_1^2, the observer can force the component to perform this local trace, as it is able report success only if it has seen the s_1^2 in its entirety. In other words: The observer as depicted in Figure 1.4(a) can force the second component C_2 to perform s_1^2. The interaction with the first observer clique does not contribute to the success.

This means that a component C_2 instead of C_1 which shows shorter interaction with the first observer and performs the same communication trace with the second observer as did C_1 (cf. Figure 1.4(b)) is equally successful. Moreover, the observer programmed in such a way to enforce the behavior of C_1 from Figure 1.4(a) cannot *prevent* the situation from Figure 1.4(b) from being successful. Of course, there is, among infinitely many others, *another* observer, that attaches the success to the trace s_1^1, which forces C_2 to perform s_1^1, as well. The crux is, that the observer from Figure 1.4(a) cannot enforce this part of the trace *in this run.*

This seems to indicate that the traces interacting with different observer cliques are independent, i.e., that the semantics of the component can be captured by sets of *projections* onto the cliques. This seems plausible insofar as it seems impossible to pass information from, in our case, the first observer clique to the second one, which reports success. There *is,* however, one piece of "information" passed from the first clique to the second: The fact that the interaction with the first clique was successfully completed, thereby allowing the component to proceed with s_1^2 that leads to success!

Figure 1.4(c) should clarify the problem. Here, C_2 insists on continuing its interaction s_1^1 by one further communication with the first observer. The observer reacts by terminating the thread. Alternatively, it could diverge for the same effect, since the notion of observation does not "see" divergence or termination and the effect is the same: The success-reporting state is not reached.

The next two examples illustrate the discussion using program code.

Example 1.4.3. *Consider the following piece of Java-code.*

Listing 1.3: Different observers

```
public class P1 {                                   // component
    public static void main(String[] arg) {
        O x1 = new O(); x1.m1();
        O x2 = new O(); x2.m2();
    }
}

class O {                                           // environment
    public void m1() { }
    public void m2() {
        System.out.println("success");
    }
}
```

The main program as instance of class P_1 instantiates two instances of class O, and calls method m_1 on the first instance and m_2 on the second. In this case the observer, consisting of two separate instances of O, can observe whether m_2 is called by inserting the corresponding print-instruction into the method body, as shown in the code.

Alternatively, the environment could observe whether m_1 is called by changing O in that m_1 reports success. Note that the notion of observation does not allow to ensure that both behaviors, the interaction via m_1 and the one via m_2, occur in the same run. *Also, if P_1 is replaced by a P_2 which invokes $x_1.m_1()$ two times before calling $x_2.m_2()$, the observer can distinguish P_1 from P_2 by programming m_2 in such a way that the second invocation blocks.* □

Example 1.4.4 (Non-determinism)**.** *If P_1 of Example 1.4.3 is replaced by one that either calls m_1 or m_2 but not both (cf. P_2 of Listing 1.4) then, if an observer can report success for P_1, then it can report success also for P_2, since each successful experiment needs to report only one success.*

Listing 1.4: Non-determinism

```
public class P2 {                                   // component
    private static int x  = 0;
    private static java.util.Random gen = new java.util.Random();
    public static void main(String[] arg) {
        choose();
        if (x==0) {
            O x1 = new O();
            x1.m1();}
        else
            {O x2 = new O();
            x2.m2();}
    }
    public static void choose () {
        x=gen.nextInt(2);return; }          // x ∈ {0,1}
}

class O {                                           // environment
    public void m1() { }
    public void m2() {
        System.out.println("success");
    }

}
```

One reason is the chosen notion of observation, where only the possible *occurrence of a* single *success-message is considered. It it further worth mentioning that the*

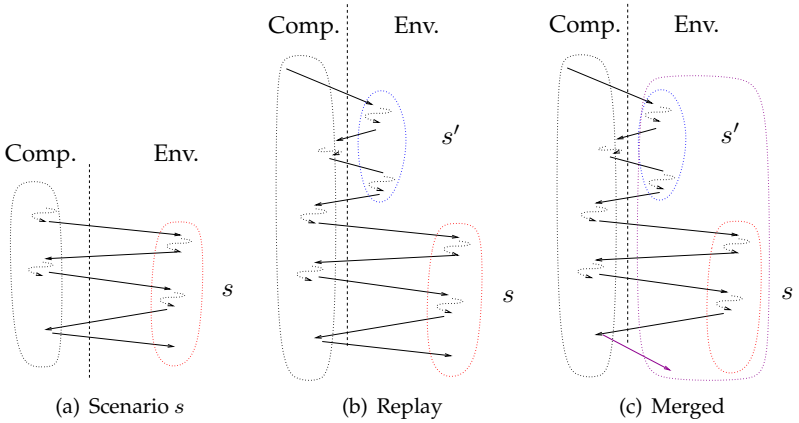
inverse implication does not hold: There exists an observer which may be successful in combination with the non-deterministic P_2, but will invariantly fail with P_1: This is an observer which, as in Listing 1.3, reports success in method m_2, but diverges or blocks or terminates in method m_1 (cf. also Figure 1.4(a) and the subsequent figures). □

1.4.3 Classes as generators of objects, replay, and determinism

Classes are generators for objects, and two instances of a class are *"identical up to their identity"*, i.e., they have the same behavior up to renaming. If the trace of a component contains a certain behavior of an object (or more generally of a clique of objects), then it is unavoidable that the component exhibits an additional trace where the equivalent behavior is shown by a second instance of the object (resp., object clique): Each behavior can be "replayed" on a fresh instance. With the possibility of cross-border instantiation, the component can create more than one equivalent instance of its observer, which performs equivalently.

Instantiation of classes into objects is a well-known example of the more general feature of modern languages of *genericity*, a mechanism to dynamically create programming language "entities". The simplest examples of genericity are allocation of reference cells and name creation (e.g. *Lisp*'s `gensym` function). The prototypical (concurrent) language for name creation is of course the π-calculus [103, 125]. In a sequential, functional setting, Pitt's and Stark's ν-calculus [115] extends a typed λ-calculus by name generation facilities. More background on the ν-calculus can be found in Stark's thesis [130] and [129]. The core in those models is the possibility to create a fresh reference or name different from all other names generated so far. For objects, however, the setting gets a bit more complex: Instantiation generates a new reference or name, but it is *attached* to the state and the code of the instance. So names are not just unstructured, basic entities of the calculus, as in the π-calculus, but part of the identities of objects with "behavior".

Consider Figures 1.5(a) and 1.5(b). The second one resembles Figure 1.3(a) before the merge. This time, however, we assume, that the interaction s' with the first clique is a prefix of the longer s up to renaming.



(a) Scenario s (b) Replay (c) Merged

Figure 1.5: Replay and merging

If s is a possible behavior of the system, then so is scenario 1.5(b): The s' is nothing else than (a prefix) of the s, apart from renaming. One can use the argument also in the reverse direction: If 1.5(b) is possible, then so is 1.5(a); in other words, both behaviors are equivalent.

If afterwards the observers are merged (cf. Figure 1.5(c)), this scenario clearly differs from the one where the interaction s' with the formerly separate clique is missing. Unlike in the situation of Figure 1.3, where the order of the previously separate cliques could not be enforced in retrospect, the merging here allows to compare the different identities (but of course still not the order).

Note that object-based calculi, for instance the one in [82], do feature instantiation. The difference is that the code is not arranged in classes. As a consequence, cross-border instantiation is not possible in that setting, i.e., there is no need to account for object connectivity, and furthermore, the issue of two instances of a class having the same behavior as in Figure 1.5 is not present. One can think of instantiation in the object-based setting to follow as what is sometimes called *singleton pattern* [54].

Example 1.4.5 (Replay)**.** *Consider the following code fragment, where the environment class O (not shown) is instantiated into an object confronted with three method calls:*

Listing 1.5: Replay(a)

```
public class P1 {                              // component
    public static void main(String[] arg) {
        O x = new O();
        x.m1(); x.m2(); x.m3();
    }
}
```

Now replace P_1 by P_2, which procures itself a second instance of O and interacts with it using the same methods calls in the same order *(actually only a prefix):*

Listing 1.6: Replay(b)

```
public class P2 {                              // component
    public static void main(String[] arg) {
        O x = new O();  O y = new O();
        y.m1(); y.m2();
        x.m1(); x.m2(); x.m3();
    }
}
```

As the second instance kept in variable y is identical to the first one except for its identity, there is no observable difference between P_1 and P_2: If m_1 or m_2 is used to report success in the situation with P_1, it will be able to do so also with P_2 and conversely. If m_3 is used to report success (after having seen interaction with m_1 and m_2, for instance), then again this does not help distinguishing P_1 and P_2

Now, when bringing the two observers into contact, as shown in Listing 1.7, then the (now merged) observer can compare what it has seen so far and could for instance distinguish P_3 as shown in the code with a variant where the method calls $y.m_1()$ and $y.m_2()$ are left out.

Listing 1.7: Replay(c)

```
public class P3 {                              // component
    public static void main(String[] arg) {
        O x = new O();  O y = new O();    // "same" obs. twice
```

```
        y.m1(); y.m2();
        x.m1(); x.m2(); x.m3();
        x.merge(y);                          // contact
    }
}
```

The possibility to create more than one instance from a class has a further impact when dealing with deterministic programs in the single-threaded setting. In a multi-threaded setting as for instance in [6], the programs are non-deterministic because of concurrency and race conditions. If a class is instantiated twice, its instances must behave "the same" up to renaming, i.e., when confronted with the same input, show the same reaction. For instance, the shorter trace s' of Figure 1.5(b) is not only possible, given s, but the left environment clique of 1.5(b) can do *nothing else* than what does the one on the right, when stimulated by the same input from the component. The scenario used environment cliques for illustration, but the same arguments apply to component cliques, as well.

1.5 Background material

I assume some acquaintance with semantics of programming languages, especially operational semantics. A standard reference for various object calculi is Abadi and Cardelli's book [2]. Excellent general references for semantics of programming language are [122] and [105]. Object-oriented languages in particular, with an emphasis on typing issues, are treated in [112] and [34]. A less theoretical survey about object-orientation is presented in [35], and about concurrency-issues in connection with *Java* in [93]. The monograph [47] provides a compendium of the theory of concurrency and Hoare-style verification of concurrent programs.

1.6 Structure of the thesis

The main technical part is split into two parts. In Part I develops the semantics in a non-concurrent, i.e., sequential, single-threaded setting. Later, in Part II, we extend syntax, semantics, and the results to include concurrency in the form of multi-threading. Part III contains concluding remarks and a discussion of related work and possible extensions. Part IV in the appendix contains those proofs omitted from the main body of the thesis.

Part I

Sequential

CHAPTER 2

A class-based calculus

In this chapter, we present a class-based calculus, basically an extension of a typed object calculus by classes. Later in Part II, we extend syntax, semantics, and the results to include concurrency in the form of multi-threading, but many of the semantical aspects, informally discussed in the introduction, already appear in the sequential setting. After a short introduction, Sections 2.2, 2.3, and 2.4 contain the syntax, the type system, and the operational semantics of the language for closed systems. After making precise the notion of observation in Section 2.5, we present the semantics for open systems, i.e., for programs interacting with the environment in Section 2.6.4.

2.1 Introduction

In this part we present a simple, single-threaded object-calculus with classes, which serve as templates for new objects. At an abstract level, the caluclus includes core features of prominent object-oriented languages such as *Java* [65] or $C^{\#}$ [50], in particular it supports instantiation from classes, method calls, and updateable object references with aliasing. The syntax is chosen in such a way that it can later be reused as a special case of the multi-threaded syntax.

2.2 Syntax

The syntax of the class-based calculus is more or less a syntactic extension of the object calculus from [62, 82]. Compared to an object-based framework, the basic change is the addition of *classes.* As in the class-based setting we do without general method update, we distinguish between *methods* and *fields*.

2.2.1 Types

The calculus is typed; also the operational semantics will be applied to well-typed program fragments, only. Besides base types B if wished —we will allow ourselves integers, booleans, ..., in illustrating examples— the type $none$ represents the absence of a return value. The name n of a class serves as the type for the instances of the class. Additionally we need for the type system as auxiliary constructs the type or interface of unnamed objects, written $[l_1{:}U_1, \ldots, l_k{:}U_k]$ and the type for classes, written $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$. It is assumed throughout that the labels l_i are all different in a type and that the order in which the labels occur does not play a role. We use furthermore the meta-mathematical notation $T.l$ to pick the type in T associated with label l, i.e., $T.l$ denotes U, when $T = [\ldots, l{:}U, \ldots]$ and analogously for $T = [\![\ldots, l{:}U, \ldots]\!]$. Only the types B and n are allowed to appear at the "user level", i.e., in a closed program given as a set of classes. Concerning the types U of methods, we write $\mathsf{Unit} \to T$ for $T_1 \times \ldots \times T_n \to T$ when $n = 0$, i.e., in particular for fields. The grammar is shown in Table 2.1.

$$\begin{array}{lcl} T & ::= & B \mid none \mid [l{:}U, \ldots, l{:}U] \mid [\![l{:}U, \ldots, l{:}U]\!] \mid n \\ U & ::= & T \times \ldots \times T \to T \end{array}$$

Table 2.1: Types

2.2.2 Classes, objects, and components

A program is given by a collection of classes and objects, together with an active entity, the thread, where the empty collection is denoted by $\mathbf{0}$. A class $n[\![O]\!]$ carries a name n and defines the implementation of its methods and fields, whereas objects $n[n, F]$ contain only fields plus a reference to the corresponding class. One difference between an object and a class concerns the nature of its name or identifier. Class names are the literals introduced when defining

the class; unlike object names, they may not be hidden using the ν-binder and may not be sent around.[1] Object names, on the other hand, are first-class citizens in that they can be stored in variables, passed to other objects as method parameters, making the scoping *dynamic,* and especially they can be created freshly by instantiating a class. There are no constant object names, at least not as values; the only way to get a new reference is instantiation.[2] The parallel composition of C_1 and C_2 is denoted by $C_1 \parallel C_2$. Note that in case of classes and objects, which are passive entities, the word "parallel" is not to be interpreted as referring to concurrent activity. The parallel composition of objects, classes (and later) multiple threads represents the *heap* of objects plus the collection of available classes plus the concurrently running threads. As the algebraic properties for the combination of objects, threads, and classes (e.g., associativity and commutativity), we use the same symbol $\parallel$ for combining all of them. $\nu(n{:}T).C$ (read "new name n of type T in C") denotes the component where the name n is hidden from the outside, as it is new and thus different from all names outside. The ν acts as binder for the name n with C as its current *scope* i.e., the components are considered up-to renaming of their bound names. The scope is *dynamic,* especially communication can enlarge the scope. The mechanism of dynamic scoping and scope extrusion is taken from the π-calculus, where the names here refer to the the dynamically generated entities of the calculus, i.e., references to objects (and later names of threads).

A method $\varsigma(n{:}T).\lambda(x_1{:}T_1, \ldots, x_k{:}T_k).t$, often abbreviated as $\varsigma(n{:}T).\lambda(\vec{x}{:}\vec{T}).t$, contains the code of the method body abstracted over the formal parameters of the method. The name parameter n plays a specific role: It is the "self" parameter bound to the identity of the object upon method call. The type system later assures that the type T of the self-parameter refers to the class containing the method. The body itself is a sequential piece of code, i.e., a *thread.*

At the level of components, one thread of code is being executed, the active entity of a running program. In particular, objects are passive. To distinguish the running thread from the threads being kept in the method bodies of the classes, we denote it by $\natural\langle t\rangle$. Unlike the other entities at component level, it is unnamed.[3] We assume a single thread present and active from the start, either inside the component or in the environment. In *Java*, for instance, this initial thread is put into one specific method of one specific class, the static `main`-method of the main class; $C^{\#}$ chooses `Main` as the name of that method.

A thread t is either a value v, or a sequence of expressions, where the *let*-construct is used for local declarations and sequencing; $stop$ stands for the deadlocked or terminated thread. Besides threads, expressions comprise conditionals (including a definedness-check for fields) and method calls, furthermore object creation via class instantiation and the creation of new threads. Values, finally, are either variables x or names n (and $true, false, 0, 1, \ldots$, when

[1] Relaxing the first restriction would not change the theory much. To allow hiding classes inside ν-binders, one would have to relax the corresponding typing rules accordingly (the T-Nu-rules from Table 2.3). Without the possibility to communicate class names, the scopes for class names would be *static*, and their scope would never escape across the interface. When additionally sending class names one needs to extend scope extrusion to class names.

[2] Fields in classes contain $\bot_c$ indicating that the field is yet uninitialized. But $\bot_c$ is not a value.

[3] Or rather: We use $\natural$ as constant symbol. Later, in the presence of multithreading, different threads are distinguished by name. Moreover, when threads can be created dynamically, their names will incorporated into the dynamic scoping mechanism, which in the single-threaded setting is used only for object names.

convenient). For the names, we will generally use o for objects and c for classes (plus their syntactic variants $o_1, o', \ldots$, resp., $c_2, \tilde{c}, \ldots$). The abstract syntax is displayed in Table 2.2.

C	$::=$	$\mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid n[\![O]\!] \mid n[n, F] \mid \natural\langle t \rangle$	component
O	$::=$	F, M	object
M	$::=$	$l = m, \ldots, l = m$	method suite
F	$::=$	$l = f, \ldots, l = f$	fields
m	$::=$	$\varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
f	$::=$	$\varsigma(n{:}T).\lambda().\bot_c \mid fv$	field
fv	$::=$	$\varsigma(n{:}T).\lambda().v$	defined field
t	$::=$	$v \mid \mathit{stop} \mid \mathit{let}\ x{:}T = e\ \mathit{in}\ t$	thread
e	$::=$	$t \mid \mathsf{if}\ v = v\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{if}\ \mathit{undef}(v.l)\ \mathsf{then}\ e\ \mathsf{else}\ e$	expression
	$\mid$	$v.l(v, \ldots, v) \mid v.l := fv \mid \mathit{new}\ n$	
v	$::=$	$x \mid n$	value

Table 2.2: Abstract syntax

We further use the following syntactic abbreviations and conventions. The sequential composition $t_1; t_2$ of two threads stands for $\mathit{let}\ x{:}T = t_1\ \mathit{in}\ t_2$, where x does not occur free in t_2. Instance variables or fields are seen as specific methods, namely of empty parameter list. Besides values v, we allow as content of a field the "value" $\bot_c$, abbreviating $\varsigma(x{:}T).\lambda().\bot_c$, which represents an undefined field value of type c. We abbreviate $l = \varsigma(n{:}T).\lambda().v$, resp., $l = \varsigma(n{:}T).\lambda().\bot_c$ by $l = v$, resp., $l = \bot_c$. Field access $v.l()$ is written shorter as $v.l$.

The important distinction between methods and fields is the one between "code" and "data", i.e., fields do not have side-effects. An operation available for fields, only, is field update $v.l \Leftarrow \varsigma(n{:}T).\lambda().v'$, which we abbreviate by $v.l := v'$; we do not allow general method update $v.l \Leftarrow \varsigma(o{:}T).\lambda().t$, as often featured by object-based calculi. Note that it is not possible to set a field back to undefined, using $v.l := \bot_c$, since $\bot_c$ is not a value. As usual and as for the corresponding types, we assume for the method suites and the "record" of fields, that the used labels are all different, and that the order in which they are listed, is irrelevant.

A further distinction between the syntactical elements of the calculus is between *static* and *dynamic* code. Static code is what is allowed to appear in *classes*, i.e., it forms the syntactical material the user can work with, whereas the (additional) dynamic code describes the entities created at run-time, in particular references to objects.[4] Especially, a field of type c as declared in classes contains $\bot_c$, as this is the only well-typed, static syntactic construct available. For simplicity we do not introduce $\bot$ or $\bot_c$ as proper value, side-stepping the question whether one can pass the undefined reference as argument, or what happens when invoking a method on $\bot$, etc.[5]

[4] [21], e.g., distinguish user vs. run-time syntax in their operational semantics of *Java* with remote method invocation.

[5] Indeed, the latter point cannot be completely avoided: It is possible, to invoke a method using a yet uninitialized field. As there is no operational rule covering that, the semantics just stops.

As said, we distinguish between fields, which are included into the objects and are updateable, and methods, which remain in the class, introducing fields syntactically as sub-category of methods. For simplicity, we adopt the convention, that when writing $c[\![F, M]\!]$ for a class, F contains the fields as *all* members of the required form, and the proper methods M none. It would be straightforward to generalize this scheme, i.e., to declare syntactically some zero-parameter members as fields and others as proper methods, which remain in the classes. We additionally disallow (read and write) references to fields across object boundaries.[6]

2.3 Type system

The type system or static semantics characterizes the well-typed programs. The system is layered into typing for *components* (in the sense of the corresponding clause in the abstract syntax of Table 2.2), and, at the second layer, rules for the syntactic sub-constituents of the components (objects, methods, expressions, ...). The two parts of the type system work on judgments of the forms

$$\Delta \vdash C : \Theta \tag{2.1}$$

for components (Table 2.3) and judgments of the form

$$\Gamma; \Delta \vdash t : \Theta \quad \Gamma; \Delta \vdash e : \Theta \quad \ldots \tag{2.2}$$

for threads, expressions, ... in Table 2.4. The type system for components from Table 2.3 recursively "calls" the one for the sub-constituents from Table 2.4, when interpreting the rules in a goal-directed manner, i.e., interpreting the rules a the specification of a recursive type checking procedure.[7] The type system is rather standard and also quite similar to the one in [82].

Table 2.3 defines the typing at the level of components or global configurations, i.e., for "sets" of objects and classes, all named, together with a single thread. As said, the typing judgments are of the form $\Delta \vdash C : \Theta$, where Δ and Θ are finite mappings from names to types. In the judgment, Δ plays the role of the typing assumptions about the environment, and Θ the commitments of the configuration, i.e., the names offered to the environment. Sometimes, the words *required* and *provided* interface are used to describe the dual roles. Anyway, Δ contains at least all external names referenced by C and Θ mentions the names offered by C.

We call a context $\Delta = n_1{:}T_1, \ldots n_k{:}T_k$ *well-formed*, written $\vdash \Delta$, if all names n_i are different and if furthermore the following holds: If $\Delta = \Delta_1, o{:}c, \Delta_2$, then $\Delta = \Delta_1', c{:}[\![l_1{:}U_1, \ldots, l_m{:}U_m]\!], \Delta_2'$, i.e., if Δ contains the binding for an object, it must provide also the type of the corresponding class. The order of the bindings in a context does not play a role. Considering Δ as a finite function from names to types, we write $\Delta(n)$ for the type of n as declared in Δ, i.e., $\Delta(n) = T$

[6] The paper [82] is slightly more general in this respect: It only forbids write-access —including method update— across component boundaries, introducing the semantic notion of *write closedness.* The theory does not depend on this difference. Therefore we content ourselves here with the simpler syntactic restriction which completely disallows field access across object boundaries.

[7] Apart from allowing a simple form of subtyping, the derivation system is goal-directed and can indeed be understood as specification of a deterministic, recursive function, with the conclusion as the argument and the premises as the recursive call.

when $\Delta = \Delta_1, n{:}T, \Delta_2$. Furthermore we write $dom(\Delta)$ for the domain of Δ. Alternatively we write $\Delta \vdash n : T$ for $\Delta(n) = T$ and $\Delta \vdash n$ for $n \in dom(\Delta)$. When writing Δ_1, Δ_2 or synonymously $\Delta_1 + \Delta_2$, we mean the disjoint combination of Δ_1 and Δ_2. The definitions are used correspondingly for commitment contexts Θ. We call a pair Δ and Θ of assumption and commitment context to be *well-formed*, written $\Delta \vdash \Theta$, when Δ and Θ are well-formed, and furthermore the domains of Δ and Θ are disjoint. We do not formalize the (straightforward) formation rules for well-formed contexts.

The empty component **0** is well-typed in any context and exports no names (cf. rule T-EMPTY). Two configurations in parallel can refer mutually to each other's commitments, and together offer the union of their names (cf. rule T-PAR). It will be an invariant of the operational semantics that the identities of parallel entities are disjoint. Therefore, Θ_1 and Θ_2 in that rule are merged disjointly, likewise for Δ, Θ_1, resp., Δ, Θ_2.

The ν-binder hides the bound name (cf. the rules T-NU$_i$ and T-NU$_e$). The two variants of the rule distinguish whether the bound object name o is an instance of an internal or an external class. As the instance of a class always belongs to the part of the system, where its class resides, the new name is added in the first case (cf. rule T-NU$_i$) to the commitment context; otherwise, to the assumption context. In both cases, the ν-construct does not only introduce a local scope for its bound name but asserts something stronger, namely the *existence* of a likewise named entity. This highlights a difference of let-bindings for variables and the introduction of names via the ν-operator: The construct to introduce and create names is the new-operator, which opens a new local scope and a named component running in parallel. We call the fact that object references of external objects can be introduced but instantiated only later when first used, *lazy instantiation;* see Section 2.4 for their operational behavior.

Let-bound variables are *stack*-allocated and checked in a stack-organized variable context Γ (see Table 2.4 below). Names created by new are *heap* allocated and thus checked in a "parallel" context (cf. again the assumption-commitment rule T-PAR). The instantiated object $o[c, F]$ will be available in the exported context Θ by rule T-NOBJ. The rules for the named entities introduce the name and its type into the commitment (cf. rules T-NOBJ and T-NCLASS). The premise $; \Delta, c{:}T \vdash [\![O]\!] : c$, resp., $; \Delta, o{:}c \vdash [F] : c$ of T-NCLASS, resp., of T-NOBJ is a judgment of the form covered in Table 2.4, with the variable context Γ empty.

Since the active thread does not have a name and cannot be referred to in the programming language, the presence of $\natural\langle t\rangle$ does not extend the context. Rule T-THREAD also requires that the thread t in $\natural\langle t\rangle$ is well-typed in its premise.[8] In the single-threaded setting, the name of the sole thread $\natural$ is treated as constant and is not covered or checked by the type system. Throughout, we assume, that a component contains one thread, only. In particular, we disallow by convention the parallel composition of $\natural\langle t_1\rangle \parallel \natural\langle t_2\rangle$; the type system does not prevent that. In the multithreaded setting, threads will carry names

[8]For the thread in T-THREAD, the type $none$ can be introduced only by $stop$. Compound threads may also carry $none$, e.g., the expression if $v_1 = v_2$ then $stop$ else $stop$, but ultimately, the type $none$ is introduced only by $stop$. Since $none$ is not a user type, in particular variables cannot be declared as carrying the type $none$. Later, two augmentational pieces of syntax will be introduced, which may also carry $none$.

and will have a type, to distinguish thread names from names of objects, for instance, and in that setting, it is the type system that prevent $n\langle t_1\rangle \parallel n\langle t_2\rangle$ (but of course allow$n_1\langle t_1\rangle \parallel n_2\langle t_2\rangle$ for two different threads), in the same way as the type system here disallows, for instance, $o[c_1, F_1] \parallel o[c_2, F_2]$. In the simpler setting here, we decided not to burden the type system with this task.

The last rule is a rule of *subsumption*. We make use of a simple form of subtyping: We allow that an object, respectively, a class contains *more* members than the interface requires. This corresponds to width subtyping. Note, however, that each object has exactly one type, its class. A name context Δ_2 imposes *less* restrictions than a context Δ_1, written $\Delta_1 \leq \Delta_2$, if it contains *fewer* classes and if the types of the common names are in subtype relation. Weakening thus allows to hide classes. Note that we do not allow weakening wrt. object names. Technically, the development could do without hiding of classes. We nonetheless allows this flexibility, as it allows an intuitive definition of a closed component: C is closed, when being typeable in $() \vdash C : ()$.

Definition 2.3.1 (Subtyping and context weakening)**.** *The relation $\leq$ on types is defined as identity for all types except for class interfaces where we have:*

$$[\![l_1{:}T_1, \ldots, l_k{:}T_k, l_{k+1}{:}T_{k+1}, \ldots]\!] \leq [\![l_1{:}T_1, \ldots l_k{:}T_k]\!] \,.$$

For well-formed name contexts $\vdash \Delta_1$ and $\vdash \Delta_2$, we write in abuse of notation $\Delta_1 \leq \Delta_2$, if the following holds. For all class names c, if $\Delta_2 \vdash c$, then $\Delta_1 \vdash c$. For object names, $\Delta_1 \vdash o$ iff. $\Delta_2 \vdash o$. For all names n with $\Delta_2 \vdash n$, we have $\Delta_1(n) \leq \Delta_2(n)$.

The $\leq$ relations are obviously reflexive, transitive, and antisymmetric. The subtyping relation on the interface types allows two forms of *hiding* via the subsumption rule, namely hiding of classes and hiding of methods of a public class.

$$\frac{}{\Delta \vdash \mathbf{0} : ()}\ \text{T-EMPTY} \qquad \frac{\Delta, \Theta_2 \vdash C_1 : \Theta_1 \qquad \Delta, \Theta_1 \vdash C_2 : \Theta_2}{\Delta \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2}\ \text{T-PAR}$$

$$\frac{\Delta \vdash C : \Theta, o{:}c \qquad \Theta \vdash c : [\![\ldots]\!]}{\Delta \vdash \nu(o{:}c).C : \Theta}\ \text{T-NU}_i \qquad \frac{\Delta, o{:}c \vdash C : \Theta \qquad \Delta \vdash c : [\![\ldots]\!]}{\Delta \vdash \nu(o{:}c).C : \Theta}\ \text{T-NU}_e$$

$$\frac{;\, \Delta, c{:}T \vdash [\![O]\!] : c}{\Delta \vdash c[\![O]\!] : (c{:}T)}\ \text{T-NCLASS} \qquad \frac{\Delta \vdash c : [\![T_F, T_M]\!] \qquad ;\, \Delta, o{:}c \vdash [F] : c}{\Delta \vdash o[c, F] : (o{:}c)}\ \text{T-NOBJ}$$

$$\frac{;\, \Delta \vdash t : none}{\Delta \vdash \natural\langle t\rangle : ()}\ \text{T-THREAD} \qquad \frac{\Delta' \leq \Delta \qquad \Theta \leq \Theta' \qquad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \text{T-SUB}$$

Table 2.3: Static semantics (components)

The typing rules of Table 2.4 formalize typing judgments for threads and objects and their syntactic sub-constituents. Besides assumptions about the names of the environment kept in Δ as before, the typing is done relative to assumptions about occurring free variables. They are kept separately in a variable context Γ, a finite mapping from variables to types.

The typing rules are rather straightforward and in many cases equivalent to the ones from [82]. Different from the object-based setting are the ones dealing with objects and classes. To formulate the open semantics, the syntax will be *augmented* later by two auxiliary constructs, namely denoting a thread returning a value to the outside, resp., a thread being blocked and waiting for a return value from the outside. Thus, for open systems, Table 2.4 will be extended by rules dealing with the new constructs (cf. Table 2.9).

The similar rules T-CLASS and T-OBJ deal with checking the members of a class, resp., the fields of an object, using the interface type of the respective class c. Furthermore it is checked whether the type of self-parameters s_i of the members equals the class c in which the members reside. Members (fields or methods of a class or fields of an object) are dealt with by rule T-MEMB. Recall the meta-mathematical notation $T.l$ from Section 2.2.1, use to select the entry labeled l from a type of the forms $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$ or $[l_1{:}U_1, \ldots, l_k{:}U_k]$. The body t of the member is checked with the contexts extended by the formal parameters. Note that the self-parameter extends the name context Δ, whereas the formal parameters $x_1, \ldots, x_k$ extend Γ. The interface type of the class the member belongs to is consulted to extract the expected return type, the T' in the rule, against which the body t is checked.

The type of a method call is the return type of the method being called, and the rule T-CALL checks compliance of the actual parameters $v_1, \ldots, v_k$ against the expected argument types. The rule applies to methods and for field lookup. A field update $v.l := v'$ invoked on an object reference leaves the class type of the object unchanged. The corresponding rule T-FUPDATE checks availability of the field being updated (indirectly by stipulating that $T.l$ is defined) and furthermore that the new value v' matches the type as declared for the field.[9] The expression $new\ c$ carries the name c as type, provided c is the name of a class (cf. rule T-NEWC). The rules for local variable declarations, for conditionals, and for testing for definedness of a reference are fairly standard (cf. rule T-LET, T-COND, and T-UNDEF). Note that the type rule for the let-binding extends the variable context Γ, not the name context Δ. The terminated thread $stop$ has any type (see rule T-STOP), highlighting the fact that control never reaches the point after $stop$. The last three rules deal with the basic syntactic constructs, variables, names, and the special "constants" $\bot_c$. For variables and names they type is looked up in the respective context, i.e., in Γ resp., in Δ.

Example 2.3.2. *Assume a class $c[\![l = \varsigma(s{:}c).\lambda().v]\!]$ with one member labeled l. The corresponding type derivation looks as follows, abbreviating $T_1 = [\![l : U_2]\!] = [\![l : \mathsf{Unit} \to T_2]\!]$:*

$$\dfrac{\dfrac{\dfrac{}{;\Delta, c{:}T_1 \vdash c : T_1} \qquad \dfrac{;\Delta, c{:}T_1, s{:}c \vdash v : T_2 \quad ;\Delta \vdash c : T_1}{;\Delta, c{:}T_1 \vdash \varsigma(s{:}c).\lambda().v : U_2}\,\text{T-MEMB}}{;\Delta, c{:}T_1 \vdash [\![l = \varsigma(s{:}c).\lambda().v]\!] : c}\,\text{T-CLASS}}{\Delta \vdash c[\![l = \varsigma(s{:}c).\lambda().v]\!] : (c{:}T_1)}\,\text{T-NCLASS}$$

Example 2.3.3. *Assume the following abbreviations: Let $[\![F, M]\!] = [\![O]\!] = [\![l_1 = f_1, \ldots, l_{k'} = f_{k'}, l_{k'+1} = m_{k'+1}, \ldots, l_k = m_k]\!]$ and furthermore $T = [\![T_F, T_M]\!] =$*

[9] Remember that later we abbreviate $v.l \Leftarrow \varsigma(s{:}c).\lambda().v'$ by $v.l := v'$. This is not a restriction, type-wise; $v.l \Leftarrow \varsigma(s{:}c).\lambda().s$ can be equivalently expressed by $v.l := v$ (and not by $v.l := s$, of course).

$$\frac{\Gamma;\Delta \vdash c : [\![(l_1{:}U_1, \ldots, l_k{:}U_k)]\!] \quad \Gamma;\Delta \vdash m_i : U_i \quad m_i = \varsigma(s_i{:}c).\lambda(\vec{x}_i{:}\vec{T}_i).t_i}{\Gamma;\Delta \vdash [\![(l_1 = m_1, \ldots, l_k = m_k)]\!] : c}\ \text{T-CLASS}$$

$$\frac{\Gamma;\Delta \vdash c : [\![(l_1{:}U_1, \ldots, l_k{:}U_k)]\!] \quad \Gamma;\Delta \vdash f_i : U_i \quad f_i = \varsigma(s_i{:}c).\lambda().v_\perp}{\Gamma;\Delta \vdash [l_1 = f_1, \ldots, l_k = f_k] : c}\ \text{T-OBJ}$$

$$\frac{\Gamma, x_1{:}T_1, \ldots, x_k{:}T_k;\Delta, n{:}c \vdash t : T' \quad \Gamma;\Delta \vdash c : T \quad T = [\![(\ldots, l{:}T_1 \times \ldots \times T_k \to T', \ldots)]\!]}{\Gamma;\Delta \vdash \varsigma(n{:}c).\lambda(x_1{:}T_1, \ldots, x_k{:}T_k).t : T.l}\ \text{T-MEMB}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![(\ldots, l{:}T_1 \times \ldots \times T_k \to T, \ldots)]\!] \quad \Gamma;\Delta \vdash v_1 : T_1 \ \ldots\ \Gamma;\Delta \vdash v_k : T_k}{\Gamma;\Delta \vdash v.l(v_1, \ldots, v_k) : T}\ \text{T-CALL}$$

$$\frac{\Gamma;\Delta \vdash v : c \qquad \Gamma;\Delta \vdash c : T \qquad \Gamma;\Delta \vdash \varsigma(s{:}c).\lambda().v' : T.l}{\Gamma;\Delta \vdash v.l \Leftarrow \varsigma(s{:}c).\lambda().v' : c}\ \text{T-FUPDATE}$$

$$\frac{\Gamma;\Delta \vdash c : [\![(T)]\!]}{\Gamma;\Delta \vdash \mathit{new}\, c : c}\ \text{T-NEWC} \qquad \frac{\Gamma;\Delta \vdash e : T_1 \qquad \Gamma, x{:}T_1;\Delta \vdash t : T_2}{\Gamma;\Delta \vdash \mathit{let}\, x{:}T_1 = e\ \mathit{in}\, t : T_2}\ \text{T-LET}$$

$$\frac{\Gamma;\Delta \vdash v_1 : T_1 \qquad \Gamma;\Delta \vdash v_2 : T_1 \qquad \Gamma;\Delta \vdash e_1 : T_2 \qquad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2}\ \text{T-COND}$$

$$\frac{\Gamma;\Delta \vdash v : c \qquad \Gamma;\Delta \vdash c : [\![(\ldots, l{:}() \to T', \ldots)]\!] \qquad \Gamma;\Delta \vdash e_1 : T_2 \qquad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\ \mathit{undef}(v.l)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2}\ \text{T-UNDEF}$$

$$\frac{}{\Gamma;\Delta \vdash \mathit{stop} : T}\ \text{T-STOP} \qquad \frac{\Gamma(x) = T}{\Gamma;\Delta \vdash x : T}\ \text{T-VAR} \qquad \frac{\Delta(n) = T}{\Gamma;\Delta \vdash n : T}\ \text{T-NAME} \qquad \frac{\Gamma;\Delta \vdash c : [\![(T)]\!]}{\Gamma;\Delta \vdash \perp_c : c}\ \text{T-UNDEF}$$

Table 2.4: Static semantics (2)

$[\![l_1{:}T_1, \ldots, l_k{:}T_k]\!]$. *Additionally we write* $o{:}c{:}T$ *for the two bindings* $o{:}c, c{:}T$ *in the contexts.*

$$\dfrac{\dfrac{\dfrac{;\, o{:}c{:}T \vdash c : T \quad ;\, o{:}c{:}T \vdash m_i : T_i \ldots ;\, o{:}c{:}T \vdash f_j{:}T_j}{;\, o{:}c{:}T \vdash [\![(F, M)]\!] : c}\,\text{T-CLASS}}{o{:}c \vdash c[\![(F, M)]\!] : (c{:}T)}\,\text{T-NCLASS} \quad \dfrac{\dfrac{;\, o{:}c{:}T \vdash c : T \quad ;\, o{:}c{:}T \vdash f'_j : T_j}{;\, o{:}c{:}T \vdash [F'] : c}\,\text{T-OBJ}}{c{:}T \vdash o[c, F'] : (o{:}c)}\,\text{T-NOBJ}}{\vdash c[\![(F, M)]\!] \parallel o[c, F'] : (o{:}c{:}T)}\,\text{T-PAR}$$

In the premises of rule T-CLASS *and* T-OBJ*, it is additionally checked that the methods* m_i *and the fields* f_j*, resp.,* f'_j *are of the form* $\varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).t$*. In the leaves of the derivation,* j *ranges over* $1, \ldots, k'$ *(fields) and* i *over* $k' + 1, \ldots, k$ *(proper methods).* □

Remark 2.3.4 (Polymorphism)**.** *The type system is not monomorphic, is allows a simple form of* subtyping*, more precisely* width *subtyping as far as the "interface types" of classes are concerned. There is no* subclassing*, however. This allows to* hide *methods and classes from outside use: A class or an object can have more methods than advertised in the commitment context, and, furthermore, there might be internal classes. This will later be needed in the completeness proof, which involves the implementation of a given behavior. The implementation uses certain methods for observation, which must not be visible from outside the component. Note that we do not have hiding of classes via the* ν*-binder. The rule of subsumption* T-SUB*, however, allows to hide component classes from the environment.* □

2.4 Operational semantics

Next the operational semantics for closed systems in the form of a small-step semantics formalizing the component *internal* steps. Later, we add rules which additionally describe the component-environment interaction (Section 2.6.4).

2.4.1 Internal steps

The internal steps are given in Table 2.5, where we distinguish between confluent steps, written $\rightsquigarrow$, and other internal transitions, written $\xrightarrow{\tau}$.[10]

The first 7 rules deal with the basic sequential constructs, all as $\rightsquigarrow$-steps. The basic evaluation mechanism is substitution. The corresponding rule RED requires that the leading let-bound variable of a thread can be replaced only by *values.* This means the redex (if any) is uniquely determined within the thread, rendering the reduction strategy deterministic. The LET rule re-organizes two nested let-expression, putting the expression e_1 at the front position to be reduced next. As a side condition in that rule, x_1 must not occur free in t, to avoid variable capture. The four rules for conditionals branch appropriately depending on the result of the comparison of two values, resp., depending on the result of the definedness check on a field. Rule COND_2 has as side condition, that $v_1 \neq v_2$. The *stop*-thread terminates for good, i.e., the rest of the thread will never be executed (cf. rule STOP).

[10] In the single-threaded setting, the distinction is not too important, as at any time at most one reduction step is enabled. It nevertheless enhances the understanding to conceptually distinguish side-effect free steps from those that may lead to race conditions when executed in the presence of other threads.

$$\natural\langle \mathit{let}\ x{:}T = v\ \mathit{in}\ t\rangle \rightsquigarrow \natural\langle t[v/x]\rangle \qquad \textsc{Red}$$

$$\natural\langle \mathit{let}\ x_2{:}T_2 = (\mathit{let}\ x_1{:}T_1 = e_1\ \mathit{in}\ e)\ \mathit{in}\ t\rangle \rightsquigarrow$$
$$\natural\langle \mathit{let}\ x_1{:}T_1 = e_1\ \mathit{in}\ (\mathit{let}\ x_2{:}T_2 = e\ \mathit{in}\ t)\rangle \qquad \textsc{Let}$$

$$\natural\langle \mathit{let}\ x{:}T = (\mathsf{if}\ v = v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathit{in}\ t\rangle \rightsquigarrow \natural\langle \mathit{let}\ x{:}T = e_1\ \mathit{in}\ t\rangle \qquad \textsc{Cond}_1$$

$$\natural\langle \mathit{let}\ x{:}T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathit{in}\ t\rangle \rightsquigarrow \natural\langle \mathit{let}\ x{:}T = e_2\ \mathit{in}\ t\rangle \qquad \textsc{Cond}_2$$

$$\natural\langle \mathit{let}\ x{:}T = (\mathsf{if}\ \mathit{undef}(\varsigma(s{:}c)\lambda().\bot_{c'})\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathit{in}\ t\rangle \rightsquigarrow$$
$$\natural\langle \mathit{let}\ x{:}T = e_1\ \mathit{in}\ t\rangle \qquad \textsc{Cond}_1^{\bot}$$

$$\natural\langle \mathit{let}\ x{:}T = (\mathsf{if}\ \mathit{undef}(\varsigma(s{:}c)\lambda().v)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathit{in}\ t\rangle \rightsquigarrow$$
$$\natural\langle \mathit{let}\ x{:}T = e_2\ \mathit{in}\ t\rangle \qquad \textsc{Cond}_2^{\bot}$$

$$\natural\langle \mathit{let}\ x{:}T = \mathit{stop}\ \mathit{in}\ t\rangle \rightsquigarrow \natural\langle \mathit{stop}\rangle \qquad \textsc{Stop}$$

$$c[\![F, M]\!] \parallel \natural\langle \mathit{let}\ x{:}c = \mathit{new}\ c\ \mathit{in}\ t\rangle \rightsquigarrow$$
$$c[\![F, M]\!] \parallel \nu(o{:}c).(o[c, F] \parallel \natural\langle \mathit{let}\ x{:}c = o\ \mathit{in}\ t\rangle) \qquad \textsc{NewO}_i$$

$$c[\![O]\!] \parallel o[c, F'] \parallel \natural\langle \mathit{let}\ x{:}T = o.l(\vec{v})\ \mathit{in}\ t\rangle \xrightarrow{\tau}$$
$$c[\![O]\!] \parallel o[c, F'] \parallel \natural\langle \mathit{let}\ x{:}T = O.l(o)(\vec{v})\ \mathit{in}\ t\rangle \qquad \textsc{Call}_i$$

$$o[c, F'] \parallel \natural\langle \mathit{let}\ x{:}T = o.l()\ \mathit{in}\ t\rangle \xrightarrow{\tau}$$
$$c[\![O]\!] \parallel o[c, F'] \parallel \natural\langle \mathit{let}\ x{:}T = F'.l(o)(\vec{v})\ \mathit{in}\ t\rangle \qquad \textsc{FLookup}$$

$$o[c, F] \parallel \natural\langle \mathit{let}\ x{:}T = o.l \Leftarrow \varsigma(s{:}c).\lambda().v\ \mathit{in}\ t\rangle \xrightarrow{\tau}$$
$$o[c, F.l \Leftarrow \varsigma(s{:}c).\lambda().v] \parallel \natural\langle \mathit{let}\ x{:}T = o\ \mathit{in}\ t\rangle \qquad \textsc{FUpdate}$$

Table 2.5: Internal steps

The step $\textsc{NewO}_i$ describes the creation of an instance of a component *internal* class $c[\![F, M]\!]$, i.e., a class whose name is contained in the configuration. Note that instantiation is a confluent step. The fields F of the class are taken as template for the created object, and the identity of the object is new and local — for the time being— to the instantiating thread; the new named object and the thread are thus enclosed in a ν-binding. Rule $\textsc{Call}_i$ treats an internal method call, resp., a field look-up. In the step, $O.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, where the method suite $[O]$ equals $[\ldots, l = \varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$. The rule FLookup does with field look-up and works similarly with the difference that it does not refer to the class, but the fields of the object to extract the value. In the rule, $F'.l(o)()$ in the steps stands, in analogy to the method look-up in $\textsc{Call}_i$, for $\bot_c[o/s] = \bot_c$, resp., for $v[o/s]$, where $[c, F'] = [c, \ldots, l = \varsigma(s{:}c).\lambda().\bot_c, \ldots]$ (if the field is yet undefined), resp., $[c, F'] = [c, \ldots, l = \varsigma(s{:}c).\lambda().v, \ldots]$. Unlike the situation for $\textsc{Call}_i$, there will later be not an external variant of the rule for field look-up in the semantics of open systems, since we do not allow field access across component boundaries. The same restriction will hold for field update in rule FUpdate for field update, where

$$[c, (\ l_1 = f_1, \ldots, l_k = f_k, l = \varsigma(s{:}c).\lambda().v'\ ).l \Leftarrow \varsigma(s{:}c).\lambda().v]$$

$$\mathbf{0} \parallel C \equiv C$$
$$C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$$
$$C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2)$$
$$\nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$$

Table 2.6: Structural congruence

stands for $[c, l_1 = f_1, \ldots, l_k = f_k, l = \varsigma(s{:}c).\lambda().v]$. We write shorter $o[c, F] \parallel \natural\langle let\ x{:}T = o.l := v\ in\ t\rangle \xrightarrow{\tau} o[c, F.l := v] \parallel \natural\langle let\ x{:}T = o\ in\ t\rangle$ for the update.

Note also that the steps given by FUPDATE, FLOOKUP, and CALLI$_i$ are $\xrightarrow{\tau}$-steps, not a confluent ones. Note further that instances of a component class invariantly belong to the component and not to the environment. This means that an instance of a component class resides after instantiation in the component, and named objects will never be exported from the component to the environment or vice versa; of course, *references* to objects may well be exported.

The reduction relations from above are used modulo *structural congruence*, which captures the algebraic properties of parallel composition and the hiding operator. The basic axioms for $\equiv$ are shown in Table 2.6 where in the fourth axiom, the name n does not occur free in C_1. The congruence relation is imported into the reduction relations in Table 2.7. Note that all syntactic entities are tacitly understood modulo α-conversion. We write $=_\alpha$ for equality up to renaming, and $\Longrightarrow$ for the reflexive and transitive closure of the internal steps from Table 2.7.

2.5 Notion of observation

We next fix a (standard) notion of semantic equivalence or rather semantic implication —one program allows at least the observations of the other. Being put into an observing context, the component, together with the context, reaches a defined point, which counts as the successful observation. A context $\mathcal{C}[\_]$ is a program "with a hole". In our setting, the hole is filled with a program fragment consisting of a *component* C in the syntactical sense, i.e., consisting of the

$$\frac{C \equiv \rightsquigarrow \equiv C'}{C \rightsquigarrow C'} \qquad \frac{C \rightsquigarrow C'}{C \parallel C'' \rightsquigarrow C' \parallel C''} \qquad \frac{C \rightsquigarrow C'}{\nu(n{:}T).C \rightsquigarrow \nu(n{:}T).C'}$$

$$\frac{C \equiv \xrightarrow{\tau} \equiv C'}{C \xrightarrow{\tau} C'} \qquad \frac{C \xrightarrow{\tau} C'}{C \parallel C'' \xrightarrow{\tau} C' \parallel C''} \qquad \frac{C \xrightarrow{\tau} C'}{\nu(n{:}T).C \xrightarrow{\tau} \nu(n{:}T).C'}$$

Table 2.7: Reduction modulo congruence

parallel composition of (named) classes, named objects, and the thread, and the context is the rest of the programs such that $\mathcal{C}[C]$ gives a well-typed program. More precisely, we assume c_b:barb $\vdash \mathcal{C}[C] : ()$, where c_b is an external class with a particular success reporting method and the type barb abbreviates $[\![succ : \mathsf{Unit} \to none]\!]$. A component C *strongly barbs on* c_b, written $C \downarrow_{c_b}$, if

$$C \equiv \nu(\vec{n}{:}\vec{T}, b{:}c_b).C' \parallel \natural\langle let\ x{:}none = b.succ()\ in\ t\rangle \tag{2.3}$$

for some C'. Furthermore, C *barbs on* c_b, written $C \Downarrow_{c_b}$, if it can reach a point which strongly barbs[11] on c_b, i.e., $C \Longrightarrow C' \downarrow_{c_b}$. The observable preorder [68] is defined similar as in [82]. Since the programs are deterministic, the distinction between a "may" and a "must" success disappears.

Definition 2.5.1 (Observable preorder)**.** *Assume* $\Delta \vdash C_1 : \Theta$ *and* $\Delta \vdash C_2 : \Theta$. *Then* $\Delta \vdash C_1 \sqsubseteq_{obs} C_2 : \Theta$, *if*

$$(C_1 \parallel C) \Downarrow_{c_b} \quad \textit{implies} \quad (C_2 \parallel C) \Downarrow_{c_b} \tag{2.4}$$

for all Θ, c_b:barb $\vdash C : \Delta$. *We will apply the definition only on components in their initial state, only, i.e. consisting only of classes plus potentially one thread.*

Technically, the definition of barbing slightly deviates from the one used in [82]. For the observation, there must be some visible piece of information shared between the program and the outside world, otherwise, there is nothing to observe. Whereas [82] uses an external *object* for this purpose, in our setting an external class is more appropriate, but the choice is not very crucial as far as the resulting theory is concerned.

2.6 External behavior

A component exchanges information with the environment via *calls* and *returns* (cf. Table 2.8). Note that there are no separate labels for object instantiation: Externally instantiated objects are created only at the point when they are actually accessed for the first time, which we call "*lazy instantiation*". Note further that the identity of the caller is not part of the label.

Definition 2.6.1. *Given a label* $\nu(\Phi).\gamma'$ *where* Φ *is a name context, i.e., a sequence of single* $(n{:}T)$ *bindings (whose names are assumed all disjoint, as usual) and where* γ' *does not contain any binders, we call* γ' *the* core *of the label. As we communicate only object names via scope extrusion but neither thread names nor class names,* Φ *contains only bindings of the form* $o{:}c$. *We refer with* $\lfloor\gamma\rfloor$ *to the core of a label* γ. *We write* $fn(\gamma)$ *and* $bn(\gamma)$ *for the free, resp., bound (object and thread names of label* γ, *and* $names(\gamma)$ *refers to all object names; in the multithreaded setting later, we include also thread names in this set.*

Note that class names, which occur in the label as the types of object names, are not counted among the names carried by a label; we are interested only in the names carried as argument in the label, not their types (and the calculus does not allow to communicate class names). A call label is abbreviated by γ_c and a return label by γ_r. The definitions are used analogously for send and receive labels.

[11] The notion of barbing was first introduced for the π-calculus in [104]. For an early citation for a "testing" based semantics in the context of the λ-calculus see [107].

γ	::=	$\langle call\ o.l(\vec{v})\rangle \mid \langle return(v)\rangle \mid \nu(n{:}T).\gamma$	basic labels
a	::=	$\gamma? \mid \gamma!$	receive and send labels

Table 2.8: Labels

Remark 2.6.2 (Lazy instantiation)**.** *As mentioned, (cross-border) object* creation *is not a separate label. However, when considering incoming communication, for instance, not just identities referring to environment objects are received by scope extrusion, but also new references referring to component objects. As reaction, the objects are created and thus the ν-syntax for those objects can be understood as label indicating an instantiation request.* □

2.6.1 Augmentation

To formulate the external semantics, we augment the syntax by two additional expressions,

$$o_1\ blocks\ for\ o_2 \quad \text{and} \quad o_2\ returns\ v\ to\ o_1\ .$$

The first one denotes a method body in o_1 waiting for a return from o_2, and dually the second expression returns v from o_2 to o_1. The corresponding typing rules are shown in Table 2.9. Note that the return expression is of arbitrary type, reflecting the fact that the control flow never reaches the point after the return (see also the typing for $stop$, for which the same argument applies, and which also carries any type).

$$\frac{}{\Gamma;\Delta \vdash o_1\ blocks\ for\ o_2 : T}\ \text{T-BLOCK} \qquad \frac{\Gamma;\Delta \vdash v : T}{\Gamma;\Delta \vdash o_2\ returns\ v\ to\ o_1 : T'}\ \text{T-RETURN}$$

Table 2.9: Static semantics (3)

Furthermore, we augment the syntax of the method definitions, such that method calls to external methods are preceded by an annotation of the caller; i.e., instead of $\varsigma(self{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots x.l(\vec{y})\ldots)$ we write

$$\varsigma(self{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots self\,x.l(\vec{y})\ldots)\ , \tag{2.5}$$

where x is of type c of an external class.

One particular thing we point out in connection with the treatment of $stop$ in connection with the block-return augmentation: The internal rule STOP for the deadlocked thread is interpreted on the new syntax insofar that it does *not* remove a $o_s\ returns\ v\ to\ o_r$-statement. I.e., the thread of the form $\natural\langle let\ x{:}T = stop\ in\ t_1; o_1\ returns\ x\ to\ o_2; t_2\rangle$ does not reduce to (a) $\natural\langle stop\rangle$, but to (b) $\natural\langle let\ x{:}T = stop\ in\ o_1\ returns\ x\ to\ o_2; t_2\rangle$ and deadlocking there, assuming the t_1 does not contain a further return-statement, i.e., assuming that t_1 is the rest of the top-most stack-frame, terminated by the shown return. Basically, we do not reduce indiscriminatingly to $stop$, as later we want to distinguish the situation (a) from (b); situation (a) indicates that the thread has started at the component side and

the thread has been return to the component, with all calls worked off. In contrast, (b) means, the execution of an incoming call has hit $stop$ and got stuck, i.e., there is at least one pending outgoing return, expected by the environment which is blocked waiting for the answer, which will, however, never occur.

2.6.2 Connectivity contexts and cliques

An important condition in the rules of the external semantics concerns which *combinations* of names can occur in communications. This phenomenon does not occur in the object-based setting and merits a closer discussion before we embark on the formalization in the following section. See also Section 1.4.

For a simple example, assume the component creates an instance of an environment class. Similar to the internal steps as given in Table 2.5, this will be done by the thread of the component executing a new-statement, with the difference that the instantiated class does not occur inside the component as in rule NEWO$_i$, but is listed in the assumption context Δ. With the class as part of the environment and thus in the hand of the observer, it can be used to make observations via its instances. Consequently, its instance belongs to the environment, as well, and communication from and to this object will be part of the interface behavior. Even if occurring likewise at the interface, however, the *instantiation itself* cannot be used by the context to make any observations about the component. This is a consequence of two facts. First, our language does not support *constructors* which, in the hand of the environment, could be used to make distinguishing observations. Secondly, exchanging a class by another and thus exchanging its instances does not make a difference in the overall behavior *unless* the component communicates with the instances; the pure existence of one object or another does not make any difference.[12]

Assume now that the component creates *two* instances of an external class or of two different external classes; the class types of the two objects do not play a role. As just explained, the objects named o_1 and o_2, say, are themselves part of the environment. Is it possible in this situation that a communication occurs where o_1 issues a call to an object of the component with o_2 as argument? Clearly the answer is no, unless the component has *given away* the identity of o_2 to o_1, since otherwise there is no means that o_1 could have learned about the existence of o_2! Therefore, such a communication must be deemed illegal. (Cf. also the informal discussion in the introductory section, especially Figure 1.1).

Therefore, for an exact representation, the semantics must *keep track* of which identities the component gives away to which object to exclude situations as just described.

For the book-keeping, a well-typed component thus takes into account the *relation* of objects from the assumption context Δ amongst each other, and the knowledge of objects from Δ about those exported by the component, i.e., those from Θ. The *connectivity contexts* E_Δ and E_Θ overapproximate the heap structure, i.e., the pointer structure of the objects among each other, divided into the component part and the environment part.

[12]The attentive reader will have noticed that there is another assumption underlying the non-observability of instantiation, namely that there is no bound on the number of objects in the system, i.e., there is no "out-of-heapspace" situation.

Definition 2.6.3 (Connectivity contexts)**.** *The semantics of an open component is given by labeled transitions between judgments of the form* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$*, where* Δ *and* Θ *are* name contexts *containing name bindings of the form* $n{:}T$*. The* connectivity *context* E_Δ*, a relation on object names, satisfies*

$$E_\Delta \subseteq \Delta \times (\Delta + \Theta)\,, \tag{2.6}$$

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$*. We write* $o_1 \hookrightarrow o_2$ *("*o_1 *may know* o_2*") for pairs from these relations.*

Note that the class names do not play a role the connectivity information; their names are global knowledge. In analogy to the name contexts Δ and Θ, E_Δ expresses assumptions about the environment, and E_Θ commitments of the component. For the formulation of the semantics itself, the commitments E_Θ are not really needed: It is unnecessary to advertise the approximated E_Θ-commitments to exclude impossible behavior with the code of the component at hand. Nevertheless, a symmetric situation is advantageous, for instance, if we come to characterize the possible traces of a component independent from its implementation (cf. Section 3.3.2).

As mentioned, the component has to over-approximate via E_Δ which environment objects are potentially connected, and, symmetrically, for its own objects via E_Θ. The worst case assumptions about the actual situation is represented using the reflexive, transitive, and symmetric closure of the $\hookrightarrow$-relation:

Definition 2.6.4 (Acquaintance)**.** *Given* Δ*,* Θ*, and* E_Δ*, we write* $\leftrightharpoons$ *for the* reflexive, transitive, *and* symmetric *closure of the* $\hookrightarrow$*-pairs of objects from the domain of* Δ*, i.e.,*

$$\leftrightharpoons \triangleq\ (\hookrightarrow\!\downarrow_{\Delta\times\Delta} \cup \hookleftarrow\!\downarrow_{\Delta\times\Delta})^* \subseteq \Delta \times \Delta\,, \tag{2.7}$$

where we write shorter $\Delta \times \Delta$ *for* $dom(\Delta) \times dom(\Delta)$*. Similarly, we write* $\Theta + \Delta$ *for* $dom(\Theta) + dom(\Delta)$*, etc.*

We also need the union $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq\ \Delta \times (\Delta + \Theta)$*, for which we will write* $\leftrightharpoons\!\hookrightarrow$ *(in the definition, ";" denotes relational composition).*

Note that we close $\hookrightarrow$ concerning environment objects, only, but not wrt. objects at the *interface*, i.e., the part of $\hookrightarrow \subseteq\ \Delta \times \Theta$. As judgment, we use

$$\Delta; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta \quad \text{, respectively,} \quad \Delta; E_\Delta \vdash o_1 \leftrightharpoons\!\hookrightarrow o_2 : \Theta\,. \tag{2.8}$$

For Θ, E_Θ, and Δ, the definitions are applied dually. To make explicit whether we are interested in the clique structure of the component or the environment, we add sometimes Δ, resp., Θ as subscript to the $\vdash$-symbol, i.e., write $\Delta; E_\Delta \vdash_\Delta o_1 \leftrightharpoons o_2 : \Theta$ and $\Delta; E_\Delta \vdash_\Delta o_1 \leftrightharpoons\!\hookrightarrow o_2 : \Theta$ for the judgments of equation (2.8), and use $\vdash_\Theta$ for the dual situation. Strictly speaking, the subscript is not needed; whether component or environment connectivity is meant is *determined* by the class type of o_1 whether $\Theta \vdash o_1$ or whether $\Delta \vdash o_1$ (and only one of the two conditions can apply). Note also that the subscript in $\vdash_\Delta$, resp., of $\vdash_\Theta$, is just a "binary flag", the Δ and Θ is not meant as the contexts mentioned in the judgments of equation (2.8).

The fact that we close in the judgment on the right-hand side of equation (2.8) wrt. environment, only, but not wrt. component objects can be understood as follows. The transitivity and symmetry of $\leftrightharpoons$ expresses the fact that the

corresponding code is abstracted away. Equation (2.8) takes the perspective of the component, in that it is the code of the environment which is considered absent and which is abstractly represented by E_Δ.

As illustration: If, for three *environment* objects o_1, o_2, and o_3, it is the case that $o_1 \hookrightarrow o_2$ (read: "it is according to the components abstract reckoning, kept in the assumptions E_Δ, possible that o_1 knows o_2, i.e., o_1 contains a reference to o_2") and $o_2 \hookrightarrow o_3$, then it is also possible that $o_2 \hookrightarrow o_1$ and $o_1 \hookrightarrow o_3$, etc., because they might contact each other and exchange their identities. This exchange of information would be possible by *internal* steps of the environment and hence would go unnoticed by the component.

If we change the scenario insofar that o_2 is no longer an environment object but belongs to the component, then we still have $o_1 \leftrightharpoons\!\hookrightarrow o_2$ implied by E_Δ, since $o_1 \hookrightarrow o_2 \in E_\Delta \subseteq \Delta \times (\Delta + \Theta)$, but it does not mean that o_1 can contact o_3 or vice versa, using transitivity and symmetry. The only way that the environment object o_1 can contact the environment object o_3 in this situation (assuming that we have described the situation completely) is *via the component object* o_2, for instance, o_2 could send the identity of o_1 to o_3 (provided, that o_2 in turn actually knows o_1, which is overapproximated by E_Θ). But this would involve an interface interaction between environment and component and would not remain unnoticed. Being "noticed" means that sending o_1 to o_3 at the interface would update E_Δ in such a way that afterwards, E_Δ it contains additionally $o_3 \hookrightarrow o_1$, and by symmetric closure on the domain of Δ we could then conclude that $o_1 \hookrightarrow o_3$.

As an aside: If closing under both the equations of E_Θ and of E_Δ, the situation would collapse into one single clique, i.e., all objects would be acquainted with each other. The reason is that each new object, created by cross-border instantiation is *at least* known to its creator.

To facilitate the following development notationally, we use the following conventions.

Notation 2.6.5 (Contexts)**.** *We abbreviate the pair of name contexts Δ, Θ as Φ, and the pair $\Delta; E_\Delta$ and $\Theta; E_\Theta$ of both assumption and commitment context by Ξ, i.e., we write $\Xi \vdash C$ for $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$. The Ξ_Δ refers to the assumption context $\Delta; E_\Delta$, and Ξ_Θ to the commitment context $\Theta; E_\Theta$. Furthermore we understand $\acute{\Delta}, \acute{\Theta}$ as $\acute{\Phi}$, $\acute{\Xi}$ as consisting of $\acute{\Delta}; \acute{E}_\Delta$ and $\acute{\Theta}; \acute{E}_\Theta$, etc.*

Thus we write the judgments (2.8) shorter as $\Xi \vdash o_1 \leftrightharpoons o_2$ and $\Xi \vdash o_1 \leftrightharpoons\!\hookrightarrow o_2$. Note that the shorter notation is unambiguous concerning whether the connectivity context E_Δ or E_Θ is meant, since the domains of Δ and Θ are disjoint. The relation $\leftrightharpoons$ is an equivalence relation on the objects from Δ and partitions them in equivalence classes. As a manner of speaking, we call a set of object names from Δ (or dually from Θ) such that for all objects o_1 and o_2 from that set, $\Delta; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta$, a *clique*, and if we speak of *the* clique of an object we mean the whole equivalence class. Given Ξ and $\Xi \vdash o$, write $[o]_{/_\Xi}$ for the clique of o wrt. the connectivity information of Ξ, or shorter $[o]$, where Ξ is clear from the context.

With E_Δ and E_Θ as part of the judgment, we must clarify what it "means", i.e., when does $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ hold? Besides the typing part, which remains unchanged, this concerns the commitment part E_Θ. The relation E_Θ asserts about C that the connectivity of the objects from the component C is

not larger the connectivity entailed by E_Θ. Given a component C and two object names o_1 from Θ and o_2 from $\Theta + \Delta$, we write $C \vdash o_1 \hookrightarrow o_2$, if $C \equiv C' \parallel o_1[\ldots, l = o_2, \ldots]$, i.e., o_1 contains in one of its fields l a reference to o_2.

Definition 2.6.6. *The judgment $\Delta \vdash C : \Theta; E_\Theta$ holds, if*

1. *$\Delta \vdash C : \Theta$, and if*
2. *$C \vdash o_1 \hookrightarrow o_2$ implies $E_\Theta; \Theta \vdash o_1 \leftrightharpoons o_2 : \Delta$.*

We often simply write $\Delta \vdash C : \Theta; E_\Theta$ to assert that the judgment is satisfied. Note again that the pairs listed in a commitment context E_Θ do not require the *existence* of connections in the components, it is rather the contrapositive situation: If E_Θ does *not* imply that two objects are in connection, possibly following the connection of other objects, then they must not be in connection in C. Thus, a larger relation E_Θ means a weaker specification.

2.6.3 Check and update of contexts

The semantics is formulated as transitions between judgments:

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \text{ or shorter } \Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C} \,. \quad (2.9)$$

The assumption contexts $\Delta; E_\Delta$ are an abstraction of the (absent) environment, consulted to *check* whether an *incoming* action is currently possible, and *updated* in an outgoing communication. The commitments play a dual role, i.e., they are updated in incoming communication. With the code of the component present, the commitment contexts are not used for checks for outgoing communication.

The check, whether the current assumptions are met in an incoming communication, is formalized as follows:

Definition 2.6.7 (Connectivity check)**.** *An incoming core label a is* well-connected *with sender o_s and wrt. a context $\acute{\Xi}$, written*[13] *$\acute{\Xi} \vdash o_s \xrightarrow{a} \_ : wc$ if*

$$\acute{\Delta}; \acute{E}_\Delta \vdash o_s \leftrightharpoons fn(a) : \acute{\Theta} \,. \quad (2.10)$$

Note that for incoming *call* labels, $fn(a)$ includes the receiver o_r, but that sender and receiver in general are not part of the label a itself (except o_r in the call label), but given as additional *argument* to the check. I.e., the check is to per interpreted as checking whether o_s is acquainted with the free names of a, where o_s is best understood as the sender of the label. In the rules later, indeed, the check will be consulted in such a way, that o_s is indeed the sender of the label. Note further that the definition assumes that a is the core of a label, i.e., it contains only free names.

Besides *checking* the connectivity assumptions before a transition, the contexts are *updated* by a step, reflecting the change of knowledge. In first approximation, an incoming communication updates the commitment contexts, but not the assumption context, and, dually, for outgoing communication. More

[13] The definition uses contexts named $\acute{\Xi}$, resp., $\acute{\Delta}$, $\acute{\Theta}$, and $\acute{E}_\Delta$ as reminder that in the rules the check will be done after the contexts have been appropriately updated.

precisely, however, incoming communication, for instance, updates *both* contexts, namely in connection with references exchanged under a ν-binder. All external transitions may exchange *bound* names in the label, i.e., bound references to objects, but not to classes since class names cannot be communicated.

For an incoming communication with binding part $\Phi' = \Delta', \Theta'$, the Δ' contains object references transmitted by *scope extrusion*, and Θ' the references to the *lazily instantiated* objects. The distinction is based on the class types which are never transmitted. I.e., in a binding Φ, which is of the form $o_1{:}c_1, \ldots o_k{:}c_k$, we can consult the classes c_i to determine whether the corresponding instance o_i belongs to the environment or to the component. The distinction uses the fact that the binding $c_i : T$ for a class cannot be contained in both the assumption and the commitment context, as their domains are disjoint. Furthermore, c_i must be declared in the commitment or the assumption context, since otherwise label would not be well-typed. And finally, classes are never communicated, i.e., whether a class belongs to the environment or to the component is fixed from the initial communication, and this fixes whether all of its instances belong to either the environment or the component. In the incoming step, Δ' extends the assumptions Δ and Θ' extends the commitments Θ.

Definition 2.6.8 (Name context update)**.** *The* update $\acute{\Phi}$ *of an assumption-commitment name context Φ by an incoming label $a = \nu(\Phi')\lfloor a \rfloor$ is defined as:*

$$\acute{\Theta} = \Theta + \Theta' \quad \textit{and} \quad \acute{\Delta} = \Delta + \Delta' \,, \tag{2.11}$$

where $+$ stands for the disjoint union of the name contexts. We write

$$\Phi + a \tag{2.12}$$

for the update. For outgoing labels, equation (2.11) applies, as well. (The situation of incoming and outgoing labels is dual in the sense that in the first case, Δ' refers to the references transmitted by scope extrusion and Θ' to the ones lazily instantiated, whereas in the latter case, the interpretation of Δ' and Θ' is reversed.

Next we consider the update of *connectivity*. We concentrate again on incoming communication; the situation for outgoing communication is dual. Communication may bring objects in connection which had been separate before, i.e., it may merge object cliques. For the commitment context, this can be directly formulated by adding the fact that the receiver of the communication now is acquainted with all transmitted arguments. See part 1 of Definition 2.6.9 below. For the update of *assumption* connectivity context E_Δ, we add that the sender knows all of the names which are transmitted boundedly (cf. part 2 of Definition 2.6.9). No update occurs wrt. names already known.

Definition 2.6.9 (Connectivity context update)**.** *The update $(\acute{E}_\Delta, \acute{E}_\Theta)$ of context (E_Δ, E_Θ) wrt. an incoming label $a = \nu(\Phi')\lfloor a \rfloor?$ with sender o_s and receiver o_r is defined as:*

1. $\acute{E}_\Theta = E_\Theta + o_r \hookrightarrow \mathit{fn}(\lfloor a \rfloor)$.
2. $\acute{E}_\Delta = E_\Delta + o_s \hookrightarrow \mathit{dom}(\Phi')$.

We write $(E_\Delta, E_\Theta) + o_s \xrightarrow{a} o_r$ for the update.

The name context and the connectivity context update are used in general together. Thus, combining Definition 2.6.8 and 2.6.9, we write

$$\Xi + o_s \xrightarrow{a} o_r \tag{2.13}$$

when updating the name and the connectivity contexts at the same time.

Remark 2.6.10 (Identity of communication partners)**.** *The identity of the communication partners is in general not part of the transmitted label. At least not "officially"; of course the sender or the receiver may be mentioned in the argument position of the communication label. The only exception is the receiver of a method call, where the callee o_r is part of the label $\langle call\ o_r.l(\vec{v})\rangle$.*

In the semantics, however, it will be the case that, even if not being mentioned as part of the label, the communication partners will be uniquely *determined, at least up-to the identity of the clique. In a multithreaded setting, this knowledge will no longer be available in those cases where a new thread crosses the environment-component border for the first time. In the single-threaded case, which is simpler in this respect, only in the very first external step, the (only) thread crosses the interface, in which case the originating clique is known to be the "initial clique".* □

Besides Definition 2.6.7, which checks whether the connectivity assumptions are met, we must check also the *static* assumptions, i.e., whether the transmitted values are of the correct types. Labels consist of a binding part and of the "core" of the label, written $a = \nu(\Phi).\lfloor a \rfloor$. In the binding part $\nu(\Phi)$, the Φ is a name context. Unlike the name context used in the type system from Section 2.3, the Φ, conventionally consisting of Δ (environment objects) and Θ (component objects), does not contain bindings for class names, as the language cannot send around thread names.

Definition 2.6.11 (Well-formedness and well-typedness of a label)**.** *We call a label $a = \nu(\Phi).\lfloor a \rfloor$* well-formed*, written*

$$\vdash a\ , \tag{2.14}$$

if $dom(\Phi) \subseteq fn(\lfloor a \rfloor)$ and if Φ is a well-formed *name-context for object names, i.e., no name bound in Φ' occurs twice. The part of the well-formedness condition for name contexts Δ and Θ from page 21 concerning class names does not apply to Φ', as we do not communicate class names.*

The assertion

$$\acute{\Delta}, \acute{\Theta} \vdash o.l? : \vec{T} \to T \tag{2.15}$$

("an incoming call to o of the method l expects arguments of type $\vec{T}$ and gives back a result of type T") is given by the following implication:

$$\frac{;\acute{\Theta} \vdash o : c \qquad ;\acute{\Delta}, \acute{\Theta} \vdash c : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]}{\acute{\Delta}, \acute{\Theta} \vdash o.l? : \vec{T} \to T}$$

Note that the receiver o of the call is checked in the commitment context $\acute{\Theta}$, only, to assure that o is a component object. Note further that to check the interface type of the class c, both $\acute{\Theta}$ and $\acute{\Delta}$ are consulted, since the argument types $\vec{T}$ or the result type T may refer to both component and environment classes. For outgoing calls,

$\acute{\Delta}, \acute{\Theta} \vdash o.l! : \vec{T} \to T$ *is defined dually. In particular, in the first premise,* $\acute{\Theta}$ *is replaced by* $\acute{\Delta}$.

Well-typedness *of an incoming core label* a *with receiver* o_r*, resp., with sender* o_r*, with expected type* $\vec{T}$*, resp.,* T*, and relative to* $\acute{\Delta}, \acute{\Theta}$ *is asserted by*

$$\acute{\Delta}, \acute{\Theta} \vdash a : \vec{T} \to \_ \quad \textit{resp.,} \quad \acute{\Delta}, \acute{\Theta} \vdash a : \_ \to T \tag{2.16}$$

(for calls and returns, respectively), as given by Table 2.10. We use $\Delta, \Theta \vdash a : wt$ *as notation to assert well-typedness. We write*

$$\acute{\Xi} \vdash o_s \xrightarrow{a} o_r : \vec{T} \to \_ \quad \textit{resp.} \quad \acute{\Xi} \vdash o_s \xrightarrow{a} o_r : \_ \to T \tag{2.17}$$

to combine the connectivity check from Definition 2.6.7 with asserting well-typedness.

$$\frac{;\acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \quad a = \langle call\ o_r.l(\vec{v})\rangle?}{\acute{\Delta}, \acute{\Theta} \vdash a : \vec{T} \to \_}\ \text{LT-CALLI}$$

$$\frac{;\acute{\Delta}, \acute{\Theta} \vdash v : T \quad a = \langle return(v)\rangle?}{\acute{\Delta}, \acute{\Theta} \vdash a : \_ \to T}\ \text{LT-RETI}$$

Table 2.10: Checking static assumptions

2.6.4 External steps

The operational rules of the semantics are given in Table 2.11. For the formulation of the connectivity contexts, we need one additional syntactical entity, $\odot$, representing the *initial clique,* i.e., the clique in which the thread starts to execute initially. The symbol $\odot$ can be mentioned in the contexts Δ, resp., Θ, i.e., for instance Δ can be of the form $\Delta', \odot$. Since the thread starts initially either in the component or the environment, $\odot$ is contained exactly in Δ or in Θ. Unlike ordinary objects mentioned in Δ or Θ, the $\odot$ does not have a type. The $\odot$ is not only contained in Δ, resp., Θ, but can consequently also be mentioned in the connectivity contexts E_Δ and E_Θ. As for our notational conventions: When writing o and its syntactical variants, we mean in the following proper object references or $\odot$.

Initially, the component contains no objects (not even hidden) and neither the initial context Ξ_0 contains bindings for object references. We write $\Xi_0 \vdash C : static$ to assert that the judgment contains no dynamically generated names, yet. In general, when writing $\Xi_0 \vdash C$, we indicate that the component is in its initial, static state. We use $\Xi \vdash static$ to assert that the context Ξ is static, i.e., Ξ consist of Δ, Θ, where Δ and Θ contain only bindings for classes, but not for objects. Δ or Θ contains, however, the symbol $\odot$. In the multithreaded setting, later, the thread names count as dynamic entities, as well, and are not allowed in a static context.

Remark 2.6.12 (Initial clique)**.** *In the terminology of arena games, a popular game-theoretical model for semantics of programming languages,* $\odot$ *plays the role of a* hereditary justifier. *In the single-threaded setting, there is exactly one such entity.*

Often, only games are considered where the opponent (= environment in our terminology) does the first move, here specified by $\Delta_0 \vdash \odot$. Games, where the player (= component) performs the first step are sometimes called positive *games [97].* □

The rules of Table 2.11 formalize the external steps as labeled transitions between judgments, transforming not only the code of the component, but also the assumption and the commitment contexts. The rules are grouped into those for incoming communication and those for outgoing. A further distinction is whether the communication is done by a method call or by a method return.

$$\frac{a = \nu(\Phi').\langle call\ o_r.l(\vec{v})\rangle?\quad \Delta_0 \vdash \odot \qquad \acute{\Xi} = \Xi_0 + \odot \xrightarrow{a} o_r\quad \acute{\Xi} \vdash o_r.l? : \vec{T} \to T\quad \acute{\Xi} \vdash \odot \xrightarrow{\lfloor a \rfloor} o_r : \vec{T} \to \_}{\Xi_0 \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Theta') \parallel \natural\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ x\ to\ \odot; stop\rangle}\ \textsc{CallI}_0$$

$$\frac{a = \nu(\Phi').\langle call\ o_r.l(\vec{v})\rangle?\quad t_{blocked} = let\ x'{:}T' = o\ blocks\ for\ o_s\ in\ t \qquad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r\quad \acute{\Xi} \vdash o_r.l? : \vec{T} \to T\quad \acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : \vec{T} \to \_}{\begin{array}{l}\Xi \vdash \nu(\Phi).(C \parallel \natural\langle t_{blocked}\rangle) \xrightarrow{a} \\ \acute{\Xi} \vdash \nu(\Phi).(C \parallel C(\Theta') \parallel \natural\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ x\ to\ o_s; t_{blocked}\rangle)\end{array}}\ \textsc{CallI}_1$$

$$\frac{a = \nu(\Phi').\langle call\ o_r.l(\vec{v})\rangle?\quad \Delta \vdash \odot \qquad \acute{\Xi} = \Xi + \odot \xrightarrow{a} o_r\quad \acute{\Xi} \vdash o_r.l? : \vec{T} \to T\quad \acute{\Xi} \vdash \odot \xrightarrow{\lfloor a \rfloor} o_r : \vec{T} \to \_}{\Xi \vdash C \parallel \natural\langle stop\rangle \xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Theta') \parallel \natural\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ x\ to\ \odot; stop\rangle}\ \textsc{CallI}_2$$

$$\frac{a = \nu(\Phi').\langle call\ o_r.l(\vec{v})\rangle!\quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi\quad \acute{\Phi} = \Phi \setminus \Phi'\quad \acute{\Delta} \vdash o_r\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r}{\begin{array}{c}\Xi \vdash \nu(\Phi).(C \parallel \natural\langle let\ x{:}T = o_s\ o_r.l(\vec{v})\ in\ t\rangle) \xrightarrow{a} \\ \acute{\Xi} \vdash \nu(\acute{\Phi}).(C \parallel \natural\langle let\ x{:}T = o_s\ blocks\ for\ o_r\ in\ t\rangle)\end{array}}\ \textsc{CallO}$$

$$\frac{a = \nu(\Phi').\langle return(v)\rangle?\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r\quad \acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : \_ \to T}{\Xi \vdash \nu(\Phi).(C \parallel \natural\langle let\ x{:}T = o_r\ blocks\ for\ o_s\ in\ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Phi).(C \parallel \natural\langle t[v/x]\rangle)}\ \textsc{RetI}$$

$$\frac{a = \nu(\Phi').\langle return(v)\rangle!\quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi\quad \acute{\Phi} = \Phi \setminus \Phi'\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r}{\Xi \vdash \nu(\Phi).(C \parallel \natural\langle let\ x{:}T = o_s\ returns\ v\ to\ o_r\ in\ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Phi}).(C \parallel \natural\langle t\rangle)}\ \textsc{RetO}$$

$$\frac{\Delta \vdash c}{\Xi \vdash \nu(\Phi').(C' \parallel \natural\langle let\ x{:}c = new\ c\ in\ t\rangle) \rightsquigarrow \Xi \vdash \nu(\Phi', o{:}c).(C' \parallel \natural\langle let\ x{:}c = o\ in\ t\rangle)}\ \textsc{NewO}_{lazy}$$

Table 2.11: External steps

The two rules $\textsc{CallI}_1$ and $\textsc{CallI}_2$ deal with incoming calls. In both cases (as for all incoming communication steps), the contexts Ξ before the step are *updated* to $\acute{\Xi}$ by setting $\acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r$, resp., using $\odot$ as sender in $\textsc{CallI}_2$. The update add the information concerning new objects and new connectivity transmitted in that step (cf. equation (2.13) for the context update). Furthermore, it is *checked* whether the label statically type-checks and that the step is

possible according to the (updated) connectivity assumptions $\acute{\Xi}$. The check is done in two stages. First, in the premise $\acute{\Xi} \vdash o_r.l? : \vec{T} \to T$, the expected types for the transmitted values as determined (cf. Definition 2.6.11, in particular equation (2.15)). The $\vec{T}$ is needed for the compliance check of the values transmitted in the current incoming call label. The return type T, in contrast, is not needed for type checking the label *now,* the type is needed to check the return value later, in case the method call should happen to return.[14] The return type T is used in form of the $let\, x : T = o_r.l(\vec{v})\, in \ldots$-syntax of the thread. The last premise $\acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\to} o_r : \vec{T} \to \_$, resp., $\acute{\Xi} \vdash \odot \overset{\lfloor a \rfloor}{\to} o_r : \vec{T} \to \_$ (cf. equation (2.17)) does the mentioned type check plus the check whether the sender object, i.e., the caller o_s, resp., $\odot$, is acquainted with all arguments of the call and with the callee o_r (cf. Definition 2.6.7). The identity of the sender of the call —o_s in case of CALLI$_1$ and $\odot$ in case of CALLI$_2$— is determined by form of the thread. In the first case, the identity is taken from the block-syntax in $t_{blocked}$. In case of CALLI$_2$, $\odot$ is taken instead.

The two discussed rules for incoming calls cover two different situations as to when an incoming call may happen: A reentrant call[15] vs. a call where the thread is already contained in the component. In the post-configuration, $C(\Theta')$ are the lazily instantiated objects mentioned in Θ': For $\Theta' = o_1{:}c_1, \ldots, o_k{:}c_k$, $\Theta = c_1{:}T_1, \ldots c_k{:}T_k, \Theta''$, and the component in the pre-configuration of the form $C' \parallel c_1[\![F_1, M_1]\!] \parallel \ldots \parallel c_k[\![F_k, M_k]\!]$, $C(\Theta')$ is given as $o_1[c_1, F_1] \parallel \ldots \parallel o_k[c_k, F_k]$. By convention, Θ' of the binding part in the incoming communication label contains the references to the lazily instantiated object, i.e., object references o_i whose class c_i belongs to the component, i.e., $\Theta \vdash c_i$. The type system thus assures that the classes $c_i[\![F_i, M_i]\!]$ are actually present in the component and that $C(\Theta')$ is well-defined.

For reentrant method calls (cf. rule CALLI$_1$), the thread is blocked, i.e., it has left the component previously via an outgoing call. The object that had been the target of the call is remembered as part of the augmented block syntax. In the rule it is referred to as o_s, as it is the sender of the current incoming call. Three points are worth mentioning: First, o_s needs not be the *actual* caller, which remains anonymous, since the callee cannot observe who really calls. The reference o_s, however, can be taken as representative of the environment clique from which the call is being issued: The call must originate from the clique where it has previously left into since it cannot enter a disjoint environment clique, at least not without detour via the component which would have been observable and recorded in the connectivity contexts. Secondly, note that the object o_s stored in the block-syntax is not necessarily the callee of the call the thread did *immediately prior* to this incoming call. In the history of the thread, there might have been message exchange in between the blocked outgoing call and the current incoming call, whose code has been popped off the stack. Nonetheless, o_s must (still) be in the clique which sends the current call. Finally, it is impossible that in CALLI$_1$, the used o_s equals $\odot$. The reason

[14] To be precise: The corresponding rule RETO for outgoing returns does *not* check whether the transmitted value has the correct type. Indeed, for outgoing communication, neither type checks nor connectivity checks are done. The checks are not needed in that situation, since the semantics enjoys a subject reduction property. I.e., starting from a well-typed component, well-typedness is preserved under reduction.

[15] Reentrant on the level of the component, not on the level of a single object.

is, that an outgoing call (via CALLO), which introduces the block-statements, stores the concrete callee identity, not an arbitrary representative of the clique of the callee, and $\odot$ cannot be called.

Rule CALLI_2 treats a non-reentrancy situation, where the thread is already present in the component nonetheless. As a consequence, the component contains the entity $\natural\langle stop\rangle$. Unlike in rule CALLI_1, the program code contains no indication as to the origin of the call. The premise $\Delta \vdash \odot$ assures that $\natural$ had started its life on the environment side. This bit of information is important as otherwise one could mistake the code $\natural\langle stop\rangle$ for the code of a (deadlocked) outgoing call. If $\Delta \vdash \odot$ and $\natural\langle stop\rangle$ is part of the component code, it is assured that the thread has left the component to the environment by some last outgoing return. I.e., the incoming call is possible now, and we can use $\odot$ as representative of the caller's identity.

Calling an external object leaves the local execution in a blocked state, waiting for the matching return carrying the returned value (cf. rules CALLO and RETI). Note that the name context $\acute{\Delta}$ is used to distinguish an external call in rule CALLO from an internal one which is covered by the corresponding rule from Table 2.5. Note that the identity o_r may be contained in the bound names Δ' of the label, i.e., the callee o_r may be lazily instantiated by the outgoing call. The contexts are updated dually to the treatment for incoming communication.

Outgoing communication is simpler wrt. type checking: Assuming that we start with a well-typed component, there is no need in re-checking now that only values of appropriate types are handed out, since the operational steps preserve well-typedness ("subject reduction").

The rule CALLI_0 is a variant of CALLI_1 and in particular of CALLI_2, describing the initial situation, where the thread starts in the environment, stipulated by $\Delta_0 \vdash \odot$, and enters the component for the first time. Note that there is no special rule dealing with the dual situation of an initial outgoing call (when $\Theta_0 \vdash \odot$) as this is subsumed by CALLO. Depending on the two mutually exclusive cases that $\Theta \vdash \odot$ or $\Delta \vdash \odot$, i.e., depending on whether the thread starts in the component or the environment, the initial step can either be an incoming call (justified by CALLI_0) or an outgoing call (by CALLO).[16] Note that in the case of an initial outgoing call, the sender does not need to be the initial clique. The first outgoing environment interaction is not necessarily caused by the initial code fragment; the component might start with internal method calls. Object creation across the component boundary is not immediately visible (cf. rule NEWO_{lazy}). The reason is that without constructor methods, instantiation alone cannot be used by an observer. The only way to do observations is by method calls. Consequently, objects are incorporated only at the point when they are first communicated to the other side or used from the other side.

The remaining rules of Table 2.11 deal with the return actions and lazy instantiation of objects. When the activity of the thread returns to the environment (cf. rule RETO), the return-statement is "popped-off" the thread; in combination with the rules for incoming calls we see that the remaining part of the thread remains blocked or is stopped. Note further in this context, that the let-bound variable x in rule RETO does not occur free in the remainder of the

[16] In the degenerated case that no classes are mentioned in Ξ or that the interface types of the classes do not offer methods, no step is possible and the component shows no external behavior at all.

thread t. Note that when the thread returns, the callee is already known. Returns are simpler than calls in that only one value is communicated, not a tuple (as we don't have compound types). To avoid case distinctions and to stress the parallel with the treatment of the calls, we denote the binding part of the label by $\nu(\Phi')$, resp., $\nu(\Delta', \Theta')$, as before, even if at least one of the name contexts are guaranteed to be empty (recall Notation 2.6.5). Rule NEWO$_{lazy}$ deals with lazy instantiation. Rule NEWO$_{lazy}$ describes the local instantiation of an external class. Instead of exporting the newly created name of the object plus the object itself immediately to the environment, the name is kept local until, if ever, it gets into contact with the environment. When this happens, the new instance will not only become known to the environment, but the object will also be instantiated in the environment. Note that the instantiation is a confluent step. Nevertheless, it is part of the external semantics in that it references the assumption context.

Remark 2.6.13 (Anonymous caller)**.** *The caller is* not *transmitted in the label which reflects the fact that it remains anonymous to the callee.*[17] *Even if anonymous, information about the caller is important to adjust the book-keeping about the connectivity appropriately, for instance when returning later.*

The antecedent of the call-rules requires that the caller o_s is acquainted with the callee o_r and with all of the arguments. In case of the very first call, we take $\odot$ as the source of the call, which is assumed to be resident in the environment. Besides the obvious fact that the caller must know the callee, there is a further aspect of connectivity to be considered: the incoming call can only be issued from an object of the clique the thread has left previously. When leaving the component by an outgoing call, the semantics remembers that as part of the block-syntax. If on the other hand the thread has left by a return, the environment clique of the last call *is* not *remembered; the corresponding stack frame is popped off. In this case, the thread must have left into the initial clique again and we take $\odot$ as representative.*

For the book-keeping, the actual identity of the caller is not needed; it suffices to know the clique *of the caller. As representative for the clique, an equivalence class of object identities, we simply pick the one remembered.*

Later, in the concurrent setting with dynamic thread creation, the problem of not knowing the sender of certain messages gets harder: In case a new *thread crosses the border, not even the originating clique may be known.* □

The assumption and commitments contexts play, not surprisingly, dual roles in the semantics. For instance, in case of incoming communication (cf. for instance the CALLI-rules), the assumption context is checked as premise, the commitment context is updated. In connection with the exchange of bound names and lazy instantiation, however, also the assumption context is updated. However, this does not lead to new information about names already known. Again in the situation for incoming communication: Since E_Δ is maintained as a worst-case assumption about the connectivity of the known external objects, learning about the existence of a fresh object must not invalidate this assumption. Intuitively, by creating new objects, initially unknown to the component, the environment cannot contact objects it could not contact otherwise. The fact

[17] Of course, the caller may transmit its identity to the callee as part of the arguments, but this nevertheless does not reveal to the callee who "actually" called.

that no new information is learnt about already known objects ("no surprise") in the assumptions can be phrased using the notion of conservative extension.

Definition 2.6.14 (Conservative extension). *Given two pairs* (Φ, E_Δ) *and* $(\acute{\Phi}, \acute{E}_\Delta)$*, of name and connectivity context, i.e.,* $E_\Delta \subseteq \Phi \times \Phi$ *(and analogously for* $(\acute{\Phi}, \acute{E}_\Delta)$*), we write* $(\Phi, E_\Delta) \vdash (\acute{\Phi}, \acute{E}_\Delta)$ *if the following two conditions holds:*

1. $\acute{\Phi} \vdash \Phi$ *and*
2. $\acute{\Phi} \vdash n_1 \leftrightharpoons n_2$ *implies* $\Phi \vdash n_1 \leftrightharpoons n_2$*, for all* n_1, n_2 *with* $\Phi \vdash n_1, n_2$*.*

In 1, $\acute{\Phi} \vdash \Phi$ *is meant as: (for all names* n*),* $\Phi \vdash n$ *implies* $\acute{\Phi} \vdash n$*.*

Lemma 2.6.15 (No surprise). *Let* $\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}$ *for some incoming label* a*. Then* $\Delta; E_\Delta \vdash \acute{\Delta}; \acute{E}_\Delta$*. For outgoing steps, the situation is dual.*

CHAPTER 3

Full abstraction

In this chapter we address the full abstraction problem for the sequential case of the calculus. Section 3.1 defines the notion of traces, intended to match the notion of observation from Section 2.5. The definition takes especially the evolving clique structure into account. This is done by defining a clique-local view on a trace, using an appropriate notion of projection, which captures the *tree-like* structure of the semantics. Sections 3.2 and 3.3 afterwards deal with soundness and completeness. As intermediate step for completeness, we characterize those traces, which are possible as interface behavior, the legal ones, in Section 3.3.2. Section 3.3.3 in particular covers in overview one key to completeness, the construction of an observer from a given, legal trace.

For both soundness and completeness we show the main top-level proofs here; minor lemmas and ancillary definitions are relegated to the appendix. In particular, Chapter B is devoted to most of the actual realization of the observer, the constructive part of the completeness proof.

3.1 Traces, cliques, and projection

The observational semantics for well-typed components takes sequences of external steps of the program fragment as starting point.

Not surprisingly, a major complication concerns the connectivity of objects. The (hypothetical) connectivity of the environment influences what is observable and the fact that the observer falls into a number of independent cliques increases the "uncertainty of observation". We can point to two reasons responsible for this effect. One is that separate observer cliques cannot determine the absolute *order* of events. Secondly, separate observers cannot cooperate to *compare identities.* This means, as long as not in contact, the observers cannot find out whether identities sent to each of them separately are the same or not. In terms of projections to the observing clique it means that local projections are considered up to α-conversion, only. However, observers can merge which means that identities, separate and local prior to the merge, become comparable and the now joint clique can find out whether local interaction of the past used the same identities or not (cf. also the discussion in Section 1.4).

3.1.1 Traces

A trace of a well-typed component is a sequence of external steps; we write $\Xi_1 \vdash C_1 \stackrel{t}{\Longrightarrow} \Xi_2 \vdash C_2$ when the component $\Xi_1 \vdash C_1$ evolves to $\Xi_2 \vdash C_2$ by executing the trace t. The corresponding rules are given in Table 3.1. For $\Xi_1 \vdash C_1 \stackrel{\epsilon}{\Longrightarrow} \Xi_2 \vdash C_2$, we write shorter $\Xi_1 \vdash C_1 \Longrightarrow \Xi_2 \vdash C_2$, where ϵ denote the empty trace. We use t, s, r, ... and their syntactic variants for traces.[1] We write $names(t)$ for the set of object names occurring in a trace t (i.e., in accordance with the corresponding definition for single labels, we do not care about the names of the classes which occur mentioned as types of the object references), $bn(t)$ for the bound object names, and $fn(t)$ for the free object names. Clearly, for a trace of a component starting from an initial configuration, there are no names occurring free, i.e., $names(t) = bn(t)$.[2] By $names_\Theta(t)$, we refer to all names referring to *component* objects (and analogously for $bn_\Theta(t)$ and $fn_\Theta(t)$), and dually for environment names, when replacing Θ by Δ. We use also $\Phi(t)$ for the bindings mentioned in t.

Later, for proving soundness and completeness, we need to dualize a trace in the following sense: Given a trace t, the dual or *complementary* trace $\bar{t}$ equals t but with all labels $\gamma!$ dualized to $\gamma?$, and vice versa.

The evolution of the cliques, both those of the component and of the environment, is tree structured, i.e., it forms a *forest,* since there may exist more than one clique of objects at the end of the trace. In the following, we need a few properties and auxiliary definitions about the evolving clique structure. First we generalize the connectivity judgment $\Xi \vdash o_1 \leftrightharpoons o_2$ (cf. equation (2.8)) to express acquaintance *after* executing some trace. As mentioned, to make explicit whether we are interested in the clique structure of the component or the environment, we use Θ, resp., Δ as subscript to the $\vdash$-symbol.

[1] The t stands also for threads in the abstract syntax; the context of usage will disambiguate the two notions.

[2] Of course, for partial traces, i.e., for $t = r\ s$, s may contain free names.

$$\frac{C_1 \Longrightarrow C_2}{\Xi_1 \vdash C_1 \overset{\epsilon}{\Longrightarrow} \Xi_2 \vdash C_2}\ \textsc{Internal} \qquad \frac{\Xi_1 \vdash C_1 \overset{a}{\rightarrow} \Xi_2 \vdash C_2}{\Xi_1 \vdash C_1 \overset{a}{\Longrightarrow} \Xi_2 \vdash C_2}\ \textsc{Base}$$

$$\frac{\Xi_1 \vdash C_1 \overset{t_1}{\Longrightarrow} \Xi_2 \vdash C_2 \qquad \Xi_2 \vdash C_2 \overset{t_2}{\Longrightarrow} \Xi_3 \vdash C_3}{\Xi_1 \vdash C_1 \overset{t_1 t_2}{\Longrightarrow} \Xi_3 \vdash C_3}\ \textsc{Conc}$$

Table 3.1: Traces

Definition 3.1.1 (Acquaintance). *Assume $\Xi \vdash C$. We write $\Xi \vdash_\Theta t \rhd o_1 \leftrightharpoons o_2$, if $\Xi \vdash C \overset{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ and $\acute{\Xi} \vdash_\Theta o_1 \leftrightharpoons o_2$. The notation is used analogously for $\leftrightharpoons\!\hookrightarrow$, and dually for $\vdash_\Delta$.*

Note that the assertions $\acute{\Xi}$ after the trace are *determined* by t and the pre-assertions Ξ, since the communication partners are determined by the trace, which in turn determine the update of the contexts. The $\acute{\Xi}$ can be seen as the strongest postcondition concerning the name and the connectivity context after t, given Ξ as precondition. By communication partners, we mean the objects referred to by the meta-variables o_s (or $\odot$) and o_r in the rules for external steps from Table 2.11. Note further that the component C is treated as black box in the definition, i.e., the $\acute{\Xi}$ is determined *only* by Ξ and the interface behavior t; the component C plays a role only insofar as it generates the trace. Indeed, later we will present an independent characterization, which traces are possible by *some* component, the legal traces. Once we have this characterization, we can define $\Xi \vdash t \rhd o_1 \leftrightharpoons o_2$ without referring to a concrete component (Definition A.5.3).

3.1.2 Projection

The linear trace t of a component $\Xi \vdash C$ describes the *global* behavior of C. This neglects the fact that the component may fall into separate cliques (as does the environment) such that *locally*, per clique of objects, the global, linear order cannot be realized (from the standpoint of the component cliques) or observed (from the standpoint of environment cliques). Therefore, we define next a local view on the global trace via the notion of *projection*. The projection is done on a clique of objects. In the definition we have to take into account that the clique structure is dynamic, i.e., when one clique is merged with a second one, interaction with objects from the previously separate clique becomes part of the common behavior after the merge and must appear in the projection onto each of the clique from that point on.[3]

The simplest form of clique is one object in isolation and we define the projection onto a single object, before we generalize the definition onto clique. The projection of t to o, written ${}_o\!\downarrow t$ can be understood as the interaction history of o in t. The local behavior of o starts from the point when o appears fresh in t, i.e., where it is introduced by a ν-binder, and takes the evolving clique structure

[3]To be precise: The merging action is the first common interaction and therefore contained in the projection of all cliques being merged.

into account. The projection plays an important role in the definition of the semantics and furthermore the completeness proof later, namely the constructive part, where we have to come up with a program that realizes the semantics. In the constructed program, each object, resp., clique, will be equipped with a *static* variant of its potential future behaviors, which correspond to the projection. By static we mean that the structure will be encoded in the programming constructs available to the user, namely classes, methods, and fields. We call the definition future or forward projection since it calculates the behavior of a clique or object, taking into account possible future clique mergings.[4]

The labels of a trace, in most cases, do not by themselves carry enough information to determine with which clique(s) the label interact. In case of the receiver of a call, the callee is mentioned in the label, but not the caller. For returns, neither the sender nor the receiver is mentioned. To facilitate the definition of projection, we *augment* the traces with the missing information.

Definition 3.1.2 (Sender and receiver augmentation)**.** *An* augmented *trace uses labels, which are augmented by information about the sender and receiver and which are of the following form in case of incoming communication (cf. also Table 2.8 for the unaugmented labels):*

$$\nu(\Phi').\langle[o_s]call\ o_r.l(\vec{v})\rangle? \quad and \quad \nu(\Phi').\langle o_s\ return\ o_r(v)\rangle? \,, \tag{3.1}$$

and dually with ! *instead of* ? *for outgoing communication.*

An augmented *trace of component C in context Ξ_0 is given by the rules of Table 2.11 where the additional sender and receiver information is added according to the respective reduction rule. I.e., in case of* CALLI_0 *and* CALLI_2*, the sender in the call label of equation (3.1) is $\odot$, for* CALLI_1 *it is o_s as mentioned in $t_{blocked}$ in the premise of the rule. Likewise for* RETI, CALLO, *and* RETO*, o_s and o_r are determined by the form of the thread before the reduction step.*

Given an augmented label a, $sender(a)$ and $receiver(a)$ pick out the sender and receiver identity from the augmentation.

Note that the identities added in the augmentation do *not* change the binding part of the label, mentioned in equation (3.1) as Φ'. In the case of call-labels, both incoming and outgoing, the receiver o_r does not belong to the augmentation, it is already contained in the unaugmented label. In case of CALLI_0 and CALLI_2, the sender is $\odot$ and is not transmitted boundedly. In CALLI_2, the sender o_s is a real (environment) object name, i.e., unequal to $\odot$. Since the thread is being blocked on o_s, the o_s is already known, i.e., $\Delta \vdash o_s$ before the step, and hence o_s in the augmentation is not *fresh.* For incoming returns, and outgoing communication, the argument is similar.

Definition 3.1.3 (Future projection)**.** *Assume a trace t with $\Xi_0 \vdash C_0 \stackrel{t}{\Longrightarrow} \Xi \vdash C$ with a component object reference o after t, i.e., $\Theta \vdash o : c$. Then the* forward projection *of t onto o, written ${}_o\!\downarrow t$, is defined as follows: $t' = {}_o\!\downarrow t$ if there exists a derivation according to Table 3.2, with $\Xi_0 \vdash \epsilon \rhd_o t$ at the bottom and with $\Xi \vdash t' \rhd_o \epsilon$ as axiom, and where o_s and o_r in the rules are determined by $o_s = sender(a)$ and $o_r = receiver(a)$ (with the exception of rule* P-EMPTY*, where no label a is involved). The update of contexts $\Xi + o_s \stackrel{a}{\rightarrow} o_r$ is used from Definition 2.6.9 and 2.6.8 (cf. equation (2.13)). The projection onto an environment clique is defined dually.*

[4] Later, for the proofs, we use also a different projection, which collects all interaction in the past of a clique (see Definition A.3.1).

$$\frac{}{\Xi \vdash r \rhd_o \epsilon}\ \text{P-EMPTY}$$

$$\frac{a = \nu(\Phi_1').\gamma? \qquad \acute{\Xi} \not\vdash o \leftrightharpoons o_r \quad \acute{\Xi} = \Xi + o_r \xleftarrow{a} o_s \quad \acute{\Xi} \vdash r \rhd_o s}{\Xi \vdash r \rhd_o a\ s}\ \text{P-IN}_1$$

$$\frac{\begin{array}{c} a = \nu(\Phi_1').\gamma? \qquad \acute{\Xi} \vdash o \leftrightharpoons o_r \qquad \acute{\Xi} = \Xi + o_r \xleftarrow{a} o_s \\ \Phi_2' = \{o_i{:}c_i \in \Xi \mid o_i \in fn(a), \acute{\Xi} \not\vdash o \leftrightharpoons o_i\} \quad a' = \nu(\Phi_1', \Phi_2').\gamma? \quad \acute{\Xi} \vdash r\ a' \rhd_o s \end{array}}{\Xi \vdash r \rhd_o a\ s}\ \text{P-IN}_2$$

$$\frac{a = \nu(\Phi'').\gamma! \quad \acute{\Xi} \not\vdash o_s \leftrightharpoons o \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash r \rhd_o s}{\Xi \vdash r \rhd_o a\ s}\ \text{P-OUT}_1$$

$$\frac{a = \nu(\Phi'').\gamma! \quad \Xi' \vdash o_s \leftrightharpoons o \quad \acute{\Xi} = \acute{\Xi} + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash r\ a \rhd_o s}{\Xi \vdash r \rhd_o a\ s}\ \text{P-OUT}_2$$

Table 3.2: Future projection onto a component clique

Given a set O of component objects, the definition of projection ${}_O\!\downarrow t$ is defined as the pointwise lifting of ${}_o\!\downarrow t$ to all object names from o. We will use the lifting mostly when O is given as a clique $[o]$ of acquainted component objects. We use also the following generalization, applying the definition not on the complete interaction of a given object, but only partially: Assume a trace $t = r\ s$ with $\Xi_0 \vdash C_0 \overset{r}{\Longrightarrow} \Xi \vdash C \overset{s}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$, with a component object reference o after r, i.e., $\Theta \vdash o : c$. Then the forward projection of s onto the clique $[o]_{/\Xi}$ (or $[o]$ for short) is written as $r \rhd {}_{[o]}\!\downarrow s$ and denotes the part of ${}_o\!\downarrow r\ s$ starting at s. For environment objects/cliques, the definition is used dually.

Before we explain the rules in bit more detail, we illustrate the intention on an example.

Example 3.1.4 (Future projection and merging)**.** *Consider the following trace t:*

$$\begin{array}{l} \nu(o_1, o_3{:}c).\langle call\ o_1.l_1(o_3)\rangle?\langle return()\rangle! \\ \nu(o_2{:}c).\langle call\ o_2.l_2()\rangle?\langle return()\rangle! \\ \langle call\ o_1.l(o_2)\rangle?\ . \end{array} \tag{3.2}$$

We do not make explicit the augmentation in this example, as it is not the focus and does not change the example. After the first 4 labels, there are two component cliques, one consisting of o_1 and o_3, the second one consisting of o_2. The subsequent incoming call $\langle call\ o_1.l(o_2)\rangle?$ merges both cliques by adding the pair $o_1 \hookrightarrow o_2$ to E_Θ in the post-configuration. The behavior is shown schematically in Figure 3.1 (without showing the type c). Note, however, that the merging action $\langle call\ o_1.l(o_2)\rangle?$ is represented in the figure as (without the type/class c)

$$\nu(o_2{:}c).\langle call\ o_1.l(o_2)\rangle? \quad \textit{resp.} \quad \nu(o_1{:}c).\langle call\ o_1.l(o_2)\rangle? \tag{3.3}$$

when seen from the perspective of o_1, resp., from o_2's perspective. This captures the fact that the identity o_2 is new *for the clique of o_1, and conversely, o_1 is new to the clique of o_2.*

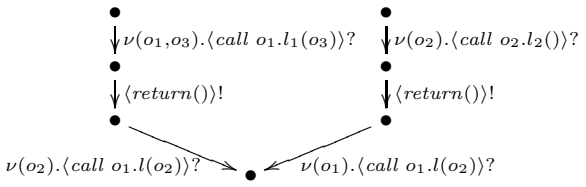


Figure 3.1: Trace of equation (3.2), schematical representation

In the specification from Table 3.2, the situation of the merge correspond to rule P-IN$_2$. *After the first 4 labels, the component consists, of the two cliques* $[o_1, o_3]$ *and* $[o_2]$, *and the connectivity commitment context* E_Θ *consists just of the pair* $o_1 \hookrightarrow o_3$. *When applying Definition 3.1.3 to object* o_1, *i.e., to invoke the rules from Table 3.2 to* $\Xi_0 \vdash \epsilon \rhd_{o_1} t$, *the state after having "executed" the first 4 labels is*

$$\Xi \vdash \nu(o_1, o_3{:}c).\langle call\ o_1.l_1(o_3)\rangle?\langle return()\rangle! \rhd_{o_1} \langle call\ o_1.l(o_2)\rangle? \,, \tag{3.4}$$

where Ξ *in particular contains the connectivity information* $o_1 \hookrightarrow o_2$. *In the history, left of* $\rhd_{o_1}$, *the third and the fourth label of trace* t *from equation (3.2) are not recorded; they have been skipped by rule* P-IN$_1$, *resp.,* P-OUT$_1$, *since the receiver* o_2 *of the incoming call of the third label, resp., the sender of the return in the fourth labels, do not (yet) belong to the clique of* o_1 *we project onto. The state of (3.4) is also when projecting onto* o_3. *In contrast, projecting onto* o_2 *and "executing" the first four labels gives the following situation,*

$$\Xi \vdash \nu(o_2{:}c).\langle call\ o_2.l_2()\rangle?\langle return()\rangle! \rhd_{o_2} \langle call\ o_1.l(o_2)\rangle? \,, \tag{3.5}$$

i.e., this time, the first two interactions of the trace of (3.2) are not recorded.

Both for (3.4) and (3.5), the next label to process is $\langle call\ o_1.l(o_2)\rangle?$, *which contains no* ν*-binder, as both* o_1 *and* o_2 *have already been encountered previously in the global trace, and in both situations, rule* P-IN$_2$ *is used. Continuing in the projection to* o_1 *in (3.4), uses* Ξ *to check that* o_1 *is acquainted with the receiver of the label (which is* o_1 *itself) and calculates* Φ_2' *as the binding context* $o_2{:}c$, *since* $\Xi \nvdash o_1 \leftrightharpoons o_2$, *i.e., from the local perspective of* o_1*'s clique, the label "looks" as* $\nu(o_2{:}c).\langle call\ o_1.l(o_2)\rangle?$, *since the* o_2 *is new to the clique. Analogously, the projection of the last label onto* o_2*'s clique is* $\nu(o_1{:}c).\langle call\ o_1.l(o_2)\rangle?$. □

Basically, considering the rules of Table 3.2 in a goal-directed manner (the premises as sub-goals to derive the conclusion, i.e., as recursive call), the projection recursively walks down the trace s, collecting all labels that concerns the clique in question and omitting the others. The recursive function given by the rules uses the additional argument Ξ to keep track of names known to the clique. The transformation of Ξ into $\acute{\Xi}$ when working off one label a is analogous to the treatment of the context in the external steps from Table 2.11 (and later for the check for legality from Table 3.5). A difference to the treatment in Table 2.11 is that here we do not use the context to *check* whether the step is possible.

That the rules of Table 3.2 give rise to a *function* rests on the observation that the premises are mutually exclusive: The label a is either an input or an

output. The cases P-IN$_1$ and P-IN$_2$ are mutually exclusive, since either $\acute{\Xi} \vdash o \leftrightharpoons o_r$ or not; the same argument separates P-OUT$_1$ and P-OUT$_2$. Furthermore, these four mentioned cases cover all possible combinations, and the premises are determined by the form of the conclusion (in particular, $\acute{\Xi}$ is determined). Finally, when starting with the goal $\Xi \vdash \epsilon \rhd_o s$, the generation of the subgoals terminates, as t is finite. The fact that we apply the projection onto a trace t generated by $\Xi_0 \vdash C_0 \stackrel{t}{\Longrightarrow} \Xi \vdash C$ guarantees that those checks had been successfully done when executing the component using Table 3.2. So we are interested for the projection only in the context update part, keeping track of the evolving clique structure.

The rule P-EMPTY covers the situation, when there is no future right of $\rhd_o$ left, at which point the generation of subgoals terminates, and the result of the projection is kept in r right of $\rhd_o$. The P-IN-rules for incoming labels distinguish whether the next label a pertains to the clique of the object o we are projecting onto. To do so it determines the receiver of a (see Definition 3.3.3) and checks whether it belongs to the clique or not, according to the *current* connectivity information, i.e., the connectivity after trace s, as given by the recursive call in the premise of the rule. If the receiver does not belong to the clique, a is not part of the projection (cf. rule P-IN$_1$). Otherwise, a is included in the projection. However, not literally, in extending the prior projection s' by a. Instead, *locally new* labels, Φ_2' in the rule, are mentioned as ν-bound. The rules for outgoing communication take the sender of the label as distinction. If the sender does not belong to the clique of interest, a is omitted, otherwise, it is added. Unlike in P-IN$_2$, the label is added literally: In the outgoing communication, all locally new labels are also *globally* new, since they are freshly created by the clique.

It will be convenient, especially when considering the encoded program, to view the rooted forest as the collection of local linear traces, one for each object in t, which form the paths of the forest from the leaves to the roots.

Definition 3.1.5 (Tree paths and subtrees)**.** *Let t be a legal trace. We write $\underline{t}$ for the representation of t as set of traces:*

$$\underline{t} \triangleq \{o \mapsto {}_o{\downarrow} t \mid o \in names(t)\} \,. \tag{3.6}$$

We furthermore need the subtree *of a trace t, given by a local end-trace s_o. Let t be a legal trace, and $r\ s = t$ for some r and s. Furthermore, let $s_o = {\downarrow_{[o]}}\ s$, for some component clique $[o]$ after r. Assume further $s_o \neq \epsilon$. Then $t - s_o$ is defined as follows:*

$$\underline{t} - s_o \triangleq \{r_o \mid r_o\ s_o \in \underline{t}\} \,. \tag{3.7}$$

Equation (3.7) defines (in linearized form) the subtree of t, accessed from one of its roots via s_o. Note the order of the linear traces forming the branches of the trees: Unlike more common representations, they do not represent the access *from the root to the nodes* of the tree, but are written inversely, describing the paths from the leaves to the roots of the forest. That, of course, is not a crucial difference. Note furthermore, that the representation from equation (3.6) of the tree describes the paths from the *leaves* to the roots, and not from each node —leaf or internal node— to the roots (from left to right). In other words, the set of linear paths is not closed under suffix.

One point for the representation is worth stressing, related to the fact that $\underline{t}$ represents a rooted forest, not a rooted tree. The roots of the forest (at the end of the trace, not the beginning) correspond to the different *cliques* of objects after the end of the trace, either from the perspective of the component or of the observer. The clique structure thus *partitions* the set of component, resp, the environment objects. This implies that the labels constituting the edges of the forest and the alphabet of the traces disambiguate which of the roots of the forest are meant. This means, the subtree defined in equation (3.7) is uniquely defined: the trace s_o, being non-empty, uniquely identifies a node in the tree, i.e., the subtree.

Example 3.1.6. *For the trace t of equation (3.2), there exists three component objects after t, i.e., $\underline{t}$ is the following set/tree:*

$$\begin{array}{ll} \{ & o_1 \mapsto \nu(o_1, o_3{:}c).\langle call\ o_1.l_1(o_3)\rangle?\ \langle return()\rangle!\ \nu(o_2).\langle call\ o_1.l(o_2)\rangle?, \\ & o_3 \mapsto \nu(o_1, o_3{:}c).\langle call\ o_1.l_1(o_3)\rangle?\ \langle return()\rangle!\ \nu(o_2).\langle call\ o_1.l(o_2)\rangle?, \\ & o_2 \mapsto \nu(o_2{:}c).\langle call\ o_2.l_2()\rangle?\ \langle return()\rangle!\ \nu(o_1).\langle call\ o_1.l(o_2)\rangle? \quad \} . \end{array} \tag{3.8}$$

Note that there are three projections, one for each object. Now, $\underline{t}{-}\nu(o_2).\langle call\ o_1.l(o_2)\rangle?$, e.g., yields

$$\begin{array}{ll} \{ & o_1 \mapsto \nu(o_1, o_3{:}c).\langle call\ o_1.l_1(o_3)\rangle?\langle return()\rangle!, \\ & o_3 \mapsto \nu(o_1, o_3{:}c).\langle call\ o_1.l_1(o_3)\rangle?\langle return()\rangle! \quad \} . \end{array} \tag{3.9}$$

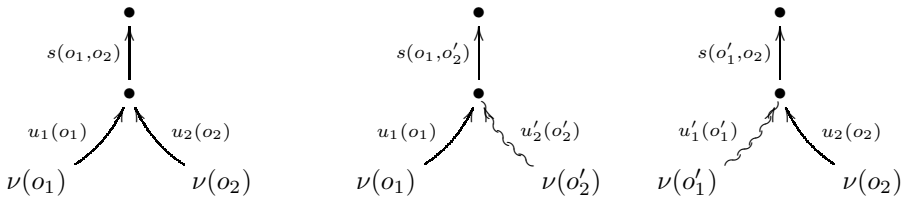
It still can be seen as tree, where the leaves correspond to o_1 and to o_3, which are then immediately merged. □

We use the projections and Definition 3.1.5 for one important equivalence on traces, namely when they are equal when considered as as tree, i.e., when projected to the behavior of all component objects in the two traces (or dually for environment objects). Considering the traces projected onto objects and the evolving clique structure ignores the linear order of certain[5] labels occurring in different cliques, i.e., they can occur in the traces under comparison in commuted or swapped order. We call the relation in traces the *swapping* relation. Later, in the proofs, we provide an equational characterization of that relation (cf. Section A.2.2 and in particular Definition A.2.22).

Definition 3.1.7 (Swapping)**.** *Assume two traces in the same context Ξ_0, i.e., $\Xi_0 \vdash C_1 \stackrel{t_1}{\Longrightarrow}$ and $\Xi_0 \vdash C_2 \stackrel{t_2}{\Longrightarrow}$, for some components C_1 and C_2. The we write $\Xi_0 \vdash t_1 \asymp_\Theta t_2$ when $\underline{t_1} = \underline{t_2}$.*

The next definition fixes an important ingredient of the semantics, capturing the generative nature of classes: Two instances of the same class are identical, and thus behave identically, up to their name. We call the fact that a second instance of a class repeats the behavior of another instance of the same class in a trace (with the identities appropriately renamed), *replay* (cf. also Section 1.4.3 for a discussion). The notion is formalized as a relation on traces below. Intuitively, t_1 is "replay-smaller" than t_2, written $t_1 \preccurlyeq_\Theta t_2$, if the complete behavior of all objects from t_1, i.e., starting from instantiation, is covered by the behavior of objects of t_2. In some sense, it is a complicated version of *prefixing*,

[5] Just occurring in different branches of the tree does not guarantee that two interactions can be swapped. This is possible only under additional side conditions. As an easy example of such a condition is that the alternating nature of calls and returns must not be violated by swapping.



(a) t_1 (b) t_2

Figure 3.2: Replay

taking into account the tree-like clique structure and the replay-phenomenon. One needs to be careful, however, with the identities of objects. As usual, the names occurring in the traces are relevant only up to renaming. To respect the clique structure and especially the merge of cliques, however, one cannot simply compare each linear object behavior in isolation. The renaming has to be done for whole cliques, not individually per behavior of an object; after a merge, the names are "coupled" and cannot be renamed independently. Cf. Example 3.1.9, and also the example from Figure 1.5 in the introduction, illustrating the same phenomenon, albeit from the dual perspective of the environment, not the component.

Definition 3.1.8 (Replay)**.** *Let $\preccurlyeq$ denote the prefix relation. Assume two traces t_1 and t_2 in the same context Ξ_0, i.e., $\Xi_0 \vdash C_1 \stackrel{t_1}{\Longrightarrow}$ and $\Xi_0 \vdash C_2 \stackrel{t_2}{\Longrightarrow}$ for some components C_1 and C_2. Let Ξ_1 be the context after t_1, and analogously for Ξ_2 and t_2. We write*

$$\Xi_0 \vdash t_1 \preccurlyeq_\Theta t_2, \tag{3.10}$$

if for all component cliques $[o_1]_{/_{\Xi_1}}$, there exists an α-renaming t_2' of t_2 s.t., for all names $o \in [o_1]_{/_{\Xi_1}}$, ${}_o\!\downarrow t_1 \preccurlyeq {}_o\!\downarrow t_2'$. We write $\Xi_0 \vdash t_1 \asymp_\Theta t_2$, if $\Xi_0 \vdash t_1 \preccurlyeq_\Theta t_2$ and vice versa. The definition for $\preccurlyeq_\Delta$ is dual.

Note that for each component clique $[o_1]$ at the end of t_1, one can choose a different α-variant of t_2 for a match.

Example 3.1.9 (Replay)**.** *The example illustrates that for comparing two traces t_1 and t_2, we cannot consider the behavior of each object in isolation in relating the objects of the two traces t_1 and t_2. In other words, the simpler definition, stipulating that each projection ${}_{o_1}\!\downarrow t_1$ is, up-to renaming, a prefix of ${}_{o_2}\!\downarrow t_2$ for some object o_2, would be incorrect.*

Consider the behavior of Figure 3.2. Scenario 3.2(a) shows a trace t_1 which ends in one single clique, which during the run is merged from two separate cliques; for simplicity, we assume that the cliques before the merge consist of one object each, namely o_1, resp., o_2. The projections ${}_{o_1}\!\downarrow t_1$ and ${}_{o_2}\!\downarrow t_1$ are of the form $u_1\ s$ and $u_2\ s$, where s is the common postfix. Scenario 3.2(b) on the right shows the interaction of four objects, resulting in two *end-cliques, the roots of the two trees. Both projected traces $u_1\ s$ and $u_2\ s$ of t_1 have, up to renaming, a counterpart in t_2; not in the same clique, however. Indeed, the scenarios of 3.2(a) and 3.2(b) are observably* different, *if the tree reflects the branching structure of the observer: At the point of merging, the observer cannot determine the exact past order of events. However, it can distinguish, obviously, whether*

u_1 and u_2 has happened in the cliques being merged, or u_1 and u_2', for instance, as in the left tree of 3.2(b). □

The external semantics from Section 2.6.4 used the assumption context as an abstract representation of the environment, intended to capture the behavior of any possible concrete environment. There is, however, one requirement left out from the assumptions, namely the knowledge that we are dealing with single-threaded programs which consequently (in absence of any operator for internal choice) behave *deterministically* (cf. Section 1.4.3 in the introduction).

In the single-threaded setting, it must be *assumed* that the environment reacts in a deterministic way.[6] Abstractly, an environment object or more generally a clique of objects, reacts deterministically, if the following holds: When confronted with an equivalent stimulus, its reaction is equivalent. Equivalent in particular means, up-to renaming of identities.

Note in particular, that the determinism-requirement is already a condition on a *single* trace, not on the behavior of a *set* of traces. If the programming language had not the properties that (1) the behavior of a piece of code could be repeated within single trace (here as different instances of the same class) and that (2) there are unrelated parts of the program or observer without influence on each other (as here the cliques), a repetition would not be possible, since there would never be a fresh start of a behavior already seen. Since this applies to a situation, when a class is instantiated more than once, the requirement is characteristic of the class-based setting and *absent* in an object-based one.

Definition 3.1.10 (Deterministic extension)**.** *Given the label output $a = \gamma!$ and a trace ra with $\Delta \vdash r \rhd a : \Theta$. The trace r can be extended deterministically by a, written $\Xi \vdash r \rhd a : det_\Theta$, if the following holds:*

$$\begin{array}{l} \Xi \vdash r\ a \preccurlyeq_\Theta r \quad \textit{or} \\ \textit{there does not exist a label } b \textit{ with} \quad \Xi \vdash r\ b \preccurlyeq_\Theta r\ . \end{array} \tag{3.11}$$

The definition for incoming labels is dual and especially refers to $\asymp_\Delta$ instead of $\asymp_\Theta$.

So, according to (3.11), a trace r can be extended without violating the assumption of determinism, of the new label a has already happened before in r (in a different clique and with different identities), or if a is really new behavior (the second line of (3.11)). Since $\Xi_0 \vdash r\ c \succcurlyeq_\Theta r$, the two asymmetric conditions in equation (3.11) are equivalent to requiring the symmetric $\asymp_\Theta$ instead of $\preccurlyeq_\Theta$.

Note that condition (3.11) does not in itself guarantee determinism for the trace; if the shorter r is deterministic, it preserves determinism when extending the trace, which is the way the check is used in the legal trace system later. We use the judgment $\Xi \vdash r \rhd a : det_\Theta$ to combine enabledness and the output determinism requirement for the next action in a single assertion. Dually we use det_Δ for input determinism for incoming communication. We write also $\Xi \vdash t : det_\Delta$, resp., $\Xi \vdash t : det_\Theta$, when the whole trace is deterministic wrt. the environment, resp., wrt. the component. We write $\Xi \vdash t : det_{\Delta,\Theta}$, when t is deterministic wrt. both environment and component.

Based on the above definitions, we define the order on traces as follows.

[6] That the component itself behaves deterministic needs not be imposed on the external behavior, because the steps of the program are deterministic.

Definition 3.1.11 ($\sqsubseteq_{trace}$)**.** *We write* $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$*, if for all* $\Xi_0 \vdash C_1 \stackrel{t}{\Longrightarrow}$ *with* $\Xi_0 \vdash t : det_\Delta$*, there exists* $\Xi_0 \vdash C_2 \stackrel{s}{\Longrightarrow}$ *with* $\Xi_0 \vdash s : det_\Delta$ *such that* $\Xi_0 \vdash s \asymp_\Delta t$*.*

The definition basically states that all behavior of C_1 can be done by C_2, as well, up-to replay and taking the evolving tree-like structure into account. As mentioned shortly above, we need only to impose determinism wrt. the environment to t, resp., s, but not det_Θ wrt. the component since this is "automatically" ensured by the semantics. Note that tree-structure and the replay are considered from the standpoint of the *observer* not of the component. It is the observer's clique structure which determines the discriminating power. The clique structure of the components does not play a role in the definition of $\sqsubseteq_{trace}$.

3.2 Soundness

Soundness means that the semantics and the notion of $\sqsubseteq_{trace}$ is not "too abstract", i.e., $\sqsubseteq_{trace}$ implies $\sqsubseteq_{obs}$.

Proposition 3.2.1 (Soundness)**.** *If* $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$*, then* $\Xi_0 \vdash C_1 \sqsubseteq_{obs} C_2$*.*

The proof is given on page 205 in Section A.4.4 of the appendix. As one main ingredient of the soundness proof (as well as for completeness) is the ability to *decompose* the joint behavior of the environment or observer C_O together with the component C_1, resp., with C_2 into two *complementary traces* t and $\bar{t}$, and conversely, a *composition* property, which allows to put together a component and an observer, both engaging in complementary traces. Complementary means, that each outgoing label of t is replaced in $\bar{t}$ by the corresponding incoming label, and vice versa. Composition and decomposition are shown in Section A.4.2 and A.4.3.

3.3 Completeness

3.3.1 Outline

Completeness is dual to soundness and states that the semantics is not too "concrete" wrt. the notion of observation, i.e., in our setting, that $\sqsubseteq_{obs}$ implies $\sqsubseteq_{trace}$. Formulated contra-positively, it means that if $C_1 \not\sqsubseteq_{trace} C_2$, then there *exists* an environment or observer which reports success for C_1, but fails to do so for C_2. Spelled out, $C_1 \not\sqsubseteq_{trace} C_2$ means that there exists a trace that C_1 can do but not C_2 (modulo $\asymp_\Delta$). Therefore, the core of completeness is to show that, whenever a trace exists discriminating between C_1 and C_2, there exists an observer which observes the difference in that it reports success for C_1 but refuses to do so for C_2.

So one key to completeness is constructive: Given a trace t, program in the calculus an observer C_t for t, enjoying the properties that $C_t \stackrel{t}{\Longrightarrow}$ and furthermore, that C_t basically can do nothing else than t (up to the unavoidable imprecision of the semantics). The observer C_t is a program of the calculus and in particular adheres to the syntactic and context-free restrictions of the language; in particular, C_t must be *well-typed* in a given context, i.e., the observer

is $\Xi_0 \vdash C_t$ rather than C_t alone. Related to that, it is programmed with the user-available syntactic material, namely classes, methods, fields (as advertised in Ξ_0), plus a single thread.

It should be clear that the construction of the observer $\Xi_0 \vdash C_t$ is impossible, if t is just an arbitrary sequence of method labels. To make $\Xi_0 \vdash C_t$ actually realizable in the calculus at hand, t must conform to a number of restrictions which reflect the semantical nature of the calculus (and the chosen way of composition; here the "parallel" composition of classes). In overview, we need to capture the following restrictions:

- *balance:* Method calls and returns must be parenthetic.
- *typing:* The calls and returns must carry values consistent with the declared interfaces.
- *connectivity:* No communication can transmit references to objects which are guaranteed to be unconnected.
- *determinism:* In the single-threaded setting, two instances of the same class must react to equivalent stimulus in an equivalent way (up-to renaming).

Apart from balance, these points have been already addressed in the design of the external semantics from Section 2.6.4.[7] So the next Section 3.3.2 distills the mentioned language restrictions into a characterization of *legal* traces.

Afterwards we present, at some level of abstraction, the coding of the observer $\Xi_0 \vdash C_t$ from a given legal trace t (Section 3.3.3). The lower-level details of the encoding are relegated to Appendix B; Section 3.3.3 conveys, however, the idea of the encoding by way of examples.

Section 3.3.4 finally contains the completeness argument. Besides the characterization of the legal traces, this additionally requires to account for the imprecision of the semantics, resp. the notion of observation, yielding certain *closure conditions* on the set of traces: If a program exhibits a trace t, then unavoidably it also performs some other traces t'. The reasons for this imprecision have been partly discussed already. Indeed, the imprecision formalized in the closure conditions on the behavior of the observer determines the limits of observation on the behavior of the component. Replay and the tree-like clique structure are thus (the two major) ingredients of the closure conditions.

3.3.2 Legal traces

In this section we characterize which traces, the "legal" ones, can occur at all at the interface of a program acting together with an environment; again the crucial difference with the object-based case is the connectivity of objects. We need furthermore to filter out non-deterministic ones —we have done this already wrt. determinism of the environment— and the calls and returns of the thread must be "parenthetic", i.e., each return must have a matching call prior in the trace and we must take into account whether the thread is resident inside the component or outside. If the thread is currently active inside, it cannot at the same time issue a call from outside.

[7] Implicitly, balance was dealt with by the stack-like structure of the thread $\natural\langle t\rangle$. E.g., an incoming call is possible only when the stack is empty or blocked from an previously outgoing call.

Balance conditions

We start with auxiliary definitions concerning the parenthetic nature of calls and returns of a sequence of interactions. The definition is similar to the one in [82]. The calls and returns of a trace form some sort of Dyck-language [137]. See also Lemma A.2.7. However, input and output are distinguished, and furthermore, the trace must start with a call. The requirement is also reminiscent of the well-bracketed condition on strategies in dialogue games ("every answer is justified by the last-asked open question") [77][13]. See also [91].

A trace (of a single thread) is balanced, if every return is an answer to a previous matching call, and vice versa, each call is answer by some later matching return. For traces in the multithreaded setting we use the definition analogously for the projection of the trace to the interactions of a given thread.

Definition 3.3.1 (Balance, pop)**.** *The assertion that a trace of labels is* balanced*, is given by the rules of Table 3.3. We write* $\vdash t : balanced$ *for* $\vdash t : balanced^+$ *or* $\vdash t : balanced^-$*.*

$$\frac{}{\vdash \epsilon : balanced^+}\ \text{B-EMPTY}^+ \qquad \frac{}{\vdash \epsilon : balanced^-}\ \text{B-EMPTY}^-$$

$$\frac{\vdash t_1 : balanced^+ \qquad \vdash t_2 : balanced^+ \qquad t_1, t_2 \neq \epsilon}{\vdash t_1\ t_2 : balanced^+}\ \text{B-II}$$

$$\frac{\vdash t_1 : balanced^- \qquad \vdash t_2 : balanced^- \qquad t_1, t_2 \neq \epsilon}{\vdash t_1\ t_2 : balanced^-}\ \text{B-OO}$$

$$\frac{\vdash t : balanced^+}{\vdash \gamma_c?\ t\ \gamma_r! : balanced^-}\ \text{B-IO} \qquad \frac{\vdash t : balanced^-}{\vdash \gamma_c!\ t\ \gamma_r? : balanced^+}\ \text{B-OI}$$

Table 3.3: Balance

The (partial) function pop on traces is defined as follows (see Lemma A.2.13 for the argument that it indeed is a partial function):

1. $pop\ (t_1 a t_2) = t_1\ a$*, if* $a = \gamma_c?$ *and* $\vdash t_2 : balanced^+$*.*
2. $pop\ (t_1 a t_2) = t_1\ a$*, if* $a = \gamma_c!$ *and* $\vdash t_2 : balanced^-$*.*

The "polarity" of $balanced^-$, resp. $balanced^+$, expresses whether the thread in the situation after as well as before the trace, resides inside the component ($+$), or outside ($-$). The rules B-IO and B-OI directly express that each return must have a matching call, and vice versa. The association of a call with the corresponding return is uniquely determined (see the lemmas below), i.e., each return has exactly one matching call, namely a call picked by an instance of B-OI (or B-IO). The concatenation of two balanced traces is balanced again, provided the two traces fit together as far as their polarity is concerned (cf. rule B-II and B-OO). In these two rules, we require, mainly for technical reasons, that the two traces are non-empty. In that way, the traces in premises of

each rule are proper subsequences of the trace from the conclusion. Note that the derivation system from Table 3.3 is not deterministic in the sense that the conclusion *determines* the subgoals in the premises. The reason for that indeterminacy are the two rules B-II and B-OO, since it may be possible to split a balanced trace in different ways into balanced subsequences. Besides that, the empty trace is covered by the two axioms B-EMPTY$^-$ and B-EMPTY$^+$. The derivation system is coherent, however, in that the mentioned indeterminacy does not influence the outcome of the derivation (with the only exception of the empty trace, which is both $balanced^-$ and $balanced^+$, depending on the choice of the axiom). Besides that, it would be straightforward to render the rules B-II and B-OO deterministic, for instance, by requiring that in the mentioned two rules, the subsequence t_1 is the *shortest* balanced prefix of t, or by similar conditions.

In a balanced trace, each call is answered by a return. Obviously, not all traces of a component nor all legal traces are balanced. The conditions on the parenthetic nature of calls and returns is rather that there are no returns without prior matching calls, which corresponds to *prefixes* of a balanced trace. We call such prefixes *weakly balanced.* For the sake of proving properties about balanced and weakly balanced traces, we provide, however, a direct characterization of weak balance. By definition, it is a weaker notion than balance, i.e., $\vdash t : balanced^+$, then $\vdash t : wbalanced^+$ (and analogously for $balanced^-$). For emphasis, we sometimes call balanced traces also strictly balanced.

Definition 3.3.2 (Weak balance)**.** *Let t be a trace (i.e., a sequence) of labels. The rules for judgments of the form $\vdash t : {}^{p_1}wbalanced^{p_2}$ are given in Table 3.4, where p_1 and p_2 (and the respective alphabetic variants) range over $+$ and $-$, which we call* polarities*. We write $\vdash t : wbalanced^+$ as abbreviation of the disjunction $\vdash t : {}^+wbalanced^+$ or $\vdash t : {}^-wbalanced^+$. I.e., the judgment $\vdash t : wbalanced^-$ states that t is weakly balanced and that the last interaction of t had been outgoing (or t is empty), and leaves it unspecified whether t starts with an incoming or an outgoing interaction.*

Analogous conventions apply to $wbalanced^-$, ${}^+wbalanced$, and ${}^-wbalanced$, i.e., we omit the polarity superscript when leaving it unspecified.

In particular, $\vdash t : wbalanced$ asserts $\vdash t : wbalanced^+$ or $\vdash t : wbalanced^-$, and we call the trace t weakly balanced *in this case.*

In order to compose (weakly) balanced traces into longer ones, the rules use assertions of the form $\vdash t : {}^{p_1}wbalanced^{p_2}$, where p_1 and p_2 refer to whether the thread resides inside or outside the component *before*, resp., *after* the trace. For instance, $\vdash t : {}^-wbalanced^+$ stipulates that the thread before t traces is active in the environment, and after executing t, it is active in the component. For the definition of strictly balanced traces, is suffices to use only *one* polarity: A balanced trace is of even length, incoming and outgoing communication occur in strict alternation (see Lemma A.2.1 below), and hence, the polarity before the balanced traces equals the one afterwards.

Now, a balanced trace is weakly balanced, as well, by WB-B. The rules WB_1 and WB_2 allow to extend a weakly balanced trace by a balanced prefix or postfix, provided the two sequences fit together as far as their polarity is concerned. The last two rules WB-CALL$^+$ and WB-CALL$^-$ allow to extend a weakly balanced trace by an *unanswered* call. This latter rule captures the difference between strictly balanced traces and weakly balanced ones.

$$\frac{\vdash t : balanced^p}{\vdash t : {}^p wbalanced^p}\ \text{WB-B}$$

$$\frac{\vdash t_1 : balanced^{p_1} \qquad \vdash t_2 : {}^{p_1} wbalanced^{p_2} \qquad t_1, t_2 \neq \epsilon}{\vdash t_1\ t_2 : {}^{p_1} wbalanced^{p_2}}\ \text{WB}_1$$

$$\frac{\vdash t_2 : {}^{p_1} wbalanced^{p_2} \qquad \vdash t_3 : balanced^{p_2} \qquad t_2, t_3 \neq \epsilon}{\vdash t_2\ t_3 : {}^{p_1} wbalanced^{p_2}}\ \text{WB}_2$$

$$\frac{\vdash t : {}^{+} wbalanced^{p}}{\vdash \gamma_c?\ t : {}^{-} wbalanced^{p}}\ \text{WB-CALL}^{+} \qquad \frac{\vdash t : {}^{-} wbalanced^{p}}{\vdash \gamma_c!\ t : {}^{+} wbalanced^{p}}\ \text{WB-CALL}^{-}$$

Table 3.4: Weak balance ($p \in \{+, -\}$)

Note that we do not have rules which allow to join two weakly balanced traces to obtain a larger one; the concatenation rules WB_1 and WB_2 require one of the subsequences to be strictly balanced. We have based the formalization on this more restrictive set of rules, to render the system more deterministic. Later, however, we prove, that the concatenation of two weakly balanced traces yields a weakly balanced result (provided, of course, they fit together as far as their polarity is concerned; see Lemma A.2.4).

Let us call a trace *alternating*, if incoming and outgoing actions of the thread occur alternatingly in it. In the multithreaded setting, the condition must apply to each thread in isolation.

Enabledness and communication partners

Input enabledness stipulates whether, given a sequence of past communication labels, an incoming call is possible in the next step; analogously for output enabledness. To be input enabled, one checks against the last matching communication. If there is no such label, enabledness depends on where the thread started.

Definition 3.3.3 (Enabledness)**.** *Given a method call γ_c. Then call-*enabledness *of γ_c after the history r and in the contexts Δ and Θ is defined as:*

$$\Delta \vdash r \rhd \gamma_c? : \Theta \textit{ if } \begin{array}{ll} pop\ r = \bot & \textit{and} \quad \Delta \vdash \odot \quad \textit{or} \\ pop\ r = r'\gamma_c'! & \end{array} \tag{3.12}$$

$$\Delta \vdash r \rhd \gamma_c! : \Theta \textit{ if } \begin{array}{ll} pop\ r = \bot & \textit{and} \quad \Theta \vdash \odot \quad \textit{or} \\ pop\ r = r'\gamma_c'? & \end{array} \tag{3.13}$$

For return labels γ_r, $\Delta \vdash r \rhd \gamma_r! : \Theta$ abbreviates $pop\, r = r'\gamma_c?$, and dually for incoming returns $\gamma_r?$.

We also say, the thread is *input-call enabled* after r if $\Delta \vdash r \rhd \gamma? : \Theta$ for some incoming call label, respectively *input-return* enabled in case of an incoming return label. The definitions are used dually for output-call enabledness and output-return enabledness. When leaving the kind of communication

unspecified we just speak of input-enabledness or output-enabledness. Note that return-enabledness implies call-enabledness, but not vice versa.

Based on a balanced past, the following definition formalizes the notion of source and target of a communication event at the end of a trace by mutual recursion and with the help of the function pop.

Definition 3.3.4 (Sender and receiver)**.** *Let $r\ a$ be a weakly balanced trace. Sender and receiver of a after history r are defined by mutual recursion, using pattern matching over the following cases:*

$$\begin{aligned} sender(\gamma_c!) &= \odot \\ sender(r'\ a'\ \gamma_c!) &= receiver(r'\ a') \\ sender(r'\ a'\ \gamma_r!) &= receiver(pop(r'\ a')) \end{aligned}$$

$$\begin{aligned} receiver(r\ \nu(\Phi).\langle call\ o_r.l(\vec{v})\rangle!) &= o_r \\ receiver(r\ \gamma_r!) &= sender(pop(r)) \end{aligned}$$

For $a = \gamma_c?$ and $a = \gamma_r?$, the definition is dual.

Note that source and target are well-defined. In particular, the recursive definition terminates. Furthermore, Lemma A.2.13 guarantees that each call of pop yields a well-defined result, as in the definition it is applied to non strictly balanced traces, only. The definition of sender and receiver (as well as balance) is given wrt. one single thread. In the multi-threaded setting, we apply the definitions on the projections of the common trace onto one thread.

For enabledness, the connectivity does not play a role. Nonetheless, we write often shorter $\Xi \vdash r \rhd a$ for $\Delta \vdash r \rhd a : \Theta$.

The next definition determines the type *expected* for the transmitted values in a label. To do so, in the case of return labels, it needs to look up the matching call from the history (for calls, all information is already contained locally in the call label). In the operational semantics, the function of Definition 3.3.5 is not needed, since the expected return type is stored as part of the block-syntax *let* $x : T = o_1$ *blocks for* o_2 *in* t.

Definition 3.3.5 (Expected typing)**.** *Assume a weakly balanced trace r and a label a. The* expected type *for the transmitted values of a after r, asserted by $\Xi \vdash r \rhd a : \vec{T} \to T$ is given as follows:*[8]

$$\frac{a = \nu(\Phi').\langle call\ o_r.l(\vec{v})\rangle? \qquad \acute{\Xi} = \Xi + \Phi(r\ a) \qquad \acute{\Delta}, \acute{\Theta} \vdash o_r.l? : \vec{T} \to T}{\Xi \vdash r \rhd a : \vec{T} \to T}$$

$$\frac{\begin{array}{c} a = \gamma_r? \qquad pop(r\ a) = r'\ \nu(\Phi').\langle call\ o_r.l(\vec{v})\rangle! \\ \acute{\Xi} = \Xi + \Phi(r\ a) \qquad \acute{\Delta}, \acute{\Theta} \vdash o_r.l! : \vec{T} \to T \end{array}}{\Xi \vdash r \rhd a : \vec{T} \to T}$$

In the rules, $\Phi(r\ a)$ refers to the name context consisting of all the bindings mentioned in trace $r\ a$. Note that o_r in the first rule is the receiver of the call label a, whereas in the second rule, it is the sender *of the return label a.*

[8] Note: In the second rule for returns, it would suffice to check the type of o_s in the potentially smaller context $\Xi_0 + \Phi(r)$, since the sender of the return cannot be transmitted boundedly by $\gamma_r?$.

In general, we do not need the type $\vec{T}$ of the arguments and the return type T and at the same time. I.e., we use the definition in most cases in the form of

$$\Xi_0 \vdash r \rhd \gamma_c? : \vec{T} \to \_ \quad \textit{for calls and} \quad \Xi_0 \vdash r \rhd \gamma_r? : \_ \to T$$

for returns. The definition is applied analogously for outgoing *calls and returns.*

Cf. also Definition 2.6.11, and in particular equation (2.15), checking well-typedness when given the expected type.

We combine the enabledness check (Definition 3.3.3), the calculation of the sender and receiver cliques from Definition 3.3.4), and the determination of the expected type as follows:

Notation 3.3.6 (Enabledness, communication partners, expected type)**.** *We write*

$$\Xi \vdash r \rhd o_s \xrightarrow{a} o_r : \vec{T} \to T \tag{3.14}$$

(reading "after r, the next label a is enabled, has sender o_s and receiver o_r, and the transmitted value is expected to be of type $\vec{T}$ for a call, resp., of type T for a return") if the following three conditions hold: (1) $\Xi \vdash r \rhd a$ (enabledness), (2) $sender(r\ a) = o_s$ and $receiver(r\ a) = o_r$ (communication partners), and (3) $\Xi \vdash r \rhd a : \vec{T} \to T$ (typing) When not interested in the type, we write[9]

$$\Xi \vdash r \rhd o_s \xrightarrow{a} o_r\ , \tag{3.15}$$

reading "after r, the label a is enabled with sender o_s and receiver o_r". To enhance readability, we sometimes write $\Xi \vdash r \rhd o_s \xleftarrow{a} o_r$ for $\Xi \vdash r \rhd o_s \xrightarrow{a} o_r$ in case of incoming *communication, and use the arrow to the right for* outgoing *communication.*

Checking for legality

The legal traces are specified by a system for judgments of the form

$$\Xi \vdash r \rhd s : trace\ , \tag{3.16}$$

stipulating that under the type and relational assumptions Δ and E_Δ and with the commitments Θ and E_Θ, the trace s is a possible future, given the (already checked) past r of the trace (remember the conventions from Notation 2.6.5). The rules are shown in Table 3.5. We omit three dual rules: Since the situation, unlike for the open, operational semantics, is now completely symmetric, the three omitted rules for *outgoing* communication (L-CALLO$_{1,2}$, L-RETO, and L-CALLO$_0$) correspond to their shown counterparts for incoming communication just by changing the labels from incoming to outgoing labels. The premises checking and updating the context Ξ remain unchanged. However, with *using* the dual label, the premise using Δ and E_Δ for the check for incoming communication refers in the dual variant to Θ and E_Θ, etc.

Distinguishing according to the first action a of the future trace, the rules check whether a is possible, i.e., whether it is well-typed and adheres to the restrictions imposed by the connectivity contexts. Furthermore, the contexts

[9] Strictly speaking, one does not need the full context Ξ for the judgment determining sender and receiver, i.e., the connectivity contexts E_Δ and E_Θ are not needed, only Δ and Θ are relevant (actually only whether $\Delta_0 \vdash \odot$ or else $\Theta_0 \vdash \odot$).

$$\frac{}{\Xi \vdash r \rhd \epsilon : trace}\ \text{L-EMPTY}$$

$$\frac{\Xi \vdash r \rhd o_s \xrightarrow{a} o_r : \vec{T} \to \_ \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : \vec{T} \to \_ \quad \Xi \not\vdash static \quad a = \nu(\Phi').\ \langle call\ o_r.l(\vec{v}) \rangle ? \quad \acute{\Xi} \vdash r\ a \rhd s : trace}{\Xi \vdash r \rhd a\ s : trace}\ \text{L-CALLI}_{1,2}$$

$$\frac{\Xi_0 \vdash \epsilon \rhd \odot \xrightarrow{a} o_r : \vec{T} \to \_ \quad \acute{\Xi} = \Xi + \odot \xrightarrow{a} o_r \quad \acute{\Xi} \vdash \odot \xrightarrow{\lfloor a \rfloor} o_r : \vec{T} \to \_ \quad \Xi_0 \vdash static \quad \Delta \vdash \odot \quad a = \nu(\Phi').\ \langle call\ o_r.l(\vec{v}) \rangle ? \quad \acute{\Xi} \vdash a \rhd s : trace}{\Xi_0 \vdash \epsilon \rhd a\ s : trace}\ \text{L-CALLI}_0$$

$$\frac{\Xi \vdash r \rhd o_s \xrightarrow{a} o_r : \_ \to T \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : \_ \to T \quad a = \nu(\Phi').\ \langle return(v) \rangle ? \quad \acute{\Xi} \vdash r\ a \rhd s : trace}{\Xi \vdash r \rhd a\ s : trace}\ \text{L-RETI}$$

Table 3.5: Legal traces (dual rules omitted)

are updated appropriately, and the rules recur checking the tail of the trace. The derivation system also resembles the one for projection from Table 3.2. In the projection, the check for well-typedness and well-connectedness is not needed: The projection definition does not *check* whether the trace is possible, it is given a possible trace and filter out the projection onto the object in question.

Concerning the rules from Table 3.5: The empty trace is always legal. An incoming call (cf. rule L-CALLI$_{1,2}$) updates the contexts and checks typing and connectivity the same way as the operational CALLI-rules from Table 2.11 did (using the judgment $\acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : \vec{T} \to \_$ from Definition 2.6.7 and 2.6.11). The rule L-CALLI$_{1,2}$ corresponds to the two situations described by CALL$_1$ and CALLI$_2$ in the operational semantics.

As slight difference between the L-CALLI-rules here and the CALLI-rules from the operational semantics concerns the determination of the sender and receiver and the treatment the expected types. In the operational semantics, sender and receiver were determined referring to the code of the component. With the code not available in the legal trace system, we refer to the history r to determine the communication partners and the expected type. This is done in the premise $\Xi_0 \vdash r \rhd o_s \xrightarrow{a} o_r : \vec{T} \to \_$ here. Unlike in the "corresponding" premise $\acute{\Xi} \vdash o_r.l? : \vec{T} \to T$ in the CALLI-rules, we do not need the return type T here at that point. The reason is, that at the point where we need it, i.e., when (and if) the matching return happens later, we consult the (then longer) history again to look up the return type. The CALLI-rules of the operational semantics use the *return type* to correctly add the method body which consists of a typed let-statement.

Back to the premises of the L-CALLI-rules here. Besides determining the type of the arguments and the communication partners, the premise $\acute{\Xi} \vdash o_s \xrightarrow{a} o_r : \vec{T} \to \_$ checks whether an incoming call is possible in a next step at all, i.e., whether, given the history r, the thread is *input enabled,* (cf. Definition 3.3.3 for the definition of enabledness), and determines the expected typing for the

parameters of the call.

Rule L-CALLI$_0$ is similar and deals with the initial situation, where the interaction starts with an incoming call. This is allowed only if the thread starts in the environment, as stated by $\Delta \vdash \odot$. Initially, there are no objects contained in either the assumptions or the commitments. Furthermore, the history left of the $\triangleright$-symbol is empty in the conclusion.

For incoming returns in L-RETI, the context update and the check works similarly, and also similar to the treatment in RETI in the semantics. An obvious difference between L-RETI and the L-CALLI-rules is that here we need the return type, not the argument type, for checking the transmitted value. This, the corresponding premise reads $\Xi \vdash o_s \xrightarrow{a} o_r : \_ \to T$ instead of $\Xi \vdash o_s \xrightarrow{a} o_r : \vec{T} \to T$ (see again Definition 3.3.5 and Notation 3.3.6).

3.3.3 Definability

At the heart of the completeness result lies *definability*: Construct a program C_t that realizes as exact as possible a given legal trace t. We start by sketching the line of argument, before we provide the construction in more detail.

Overview and illustration

Before we embark on the construction of C_t itself, it is instructive to abstractly think of which requirements the commitments express and how the legality rules manipulate them, since Θ and E_Θ must be implemented by some *"data structures"* and their changes by some *"algorithms"*.

The context Θ lists all named components which have to be present and publicly visible in C_t. As named components we have classes and objects, and as active component the only thread.

Concentrating for now on the objects, Θ is changed by scope extrusion when internally created objects get exposed to the environment, i.e., their scope opens across the component boundary. This sure can be the case for outgoing communication, but in the lazy instantiation scheme we employ, also incoming communication may make the component aware that the environment has created instances of internal classes.

The connectivity context $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$ stipulates for each component object from Θ, which other objects from Θ and from Δ it is expected to know and which it should be able to contact, when necessary. In principle, there are various ways to implement E_Θ. We adopt a "*distributed*" implementation, by which we mean that each object contains in its instance state its share of E_Θ.[10] Being kept distributed over the members of the clique, changes to E_Θ must be propagated to all members.

We describe the implementation and the propagation not yet in concrete ν-calculus terms and postpone the problem to ultimately *encode* the solution into classes and objects (cf. Chapter B).

The fact that E_Θ is implemented in a distributed way does not literally mean that each object has a local copy of the full E_Θ available in its instance state, rather than its *local view* of E_Θ in the following sense: Each object keeps in its instance state the list of objects from Θ as well as those from Δ it is aware

[10] How to actually encode it in the calculus, we will discuss later.

of. In slight abuse of notation we call the respective instance variable Θ; to distinguish it from the abstract context Θ we will always reference it in a qualified manner as $self.\Theta$.[11]

As mentioned, the distributed implementation of E_Θ makes it necessary to broadcast across the clique any change to the connectivity context. This change of E_Θ occurs in the system for legal traces when dealing with *incoming* communication —reception of names may increase the knowledge of the component clique— and for *outgoing* communication —*new* objects previously unknown to the environment are exported.

In any case, the change of knowledge takes place in *some* object, namely, the caller, resp., the callee of the communication. The object therefore has to update both its own knowledge accordingly *and* to inform all members of its clique about the change. Other changes to Θ, resp., E_Θ require to create new, appropriately connected objects.

The pieces of synchronization code in the construction of the component C_t come in two flavors, *input* and *output* synchronization code, and flank the corresponding external transition steps at the interface. Output synchronization code *precedes* the corresponding output, and dually, input synchronization *trails* the input action (cf. equation (3.47) later).

As the commitment contexts of the judgments provide a concise specification of the component, the requirement for the synchronization can be clearly understood by looking at the change of the $\Xi \vdash C$- judgments in external steps (cf. Tables 2.11 and 3.5). The changes are always *additive,* i.e., the contexts only grow larger. To implement the extension of the typing context Θ in an output step, the component must *create* corresponding objects, whose references are then published. Likewise, the component must cater for lazily instantiated objects of the environment, which lead to an extension of E_Δ in an output step. On the other hand, the component is not responsible for extensions of E_Θ by incoming lazy instantiation.

On an abstract level, each object (or clique) needs to implement "exactly" the behavior as given by the prescribed trace. In principle, this can be achieved as follows: Each object is equipped with a representation of its prescribed behavior, its *future*, and the program follows this path step by step.

The major complication in this scheme comes from the fact, that objects cannot be coded *individually*, but arise as instances of a class, i.e., as incarnations of the corresponding code, common to all instances of a class. This means, each class must contain the foreseen behaviors of all their instances. Furthermore, after instantiation, an object of a given class has no a priori way of telling which future it is supposed to play; directly after instantiation, all instances are identical up to their name, i.e., α-equivalent.

Remark 3.3.7 (Object-based setting)**.** *Note that in an object-based setting, for instance in [82], the problem is absent. Objects are instantiated, not from classes, but directly from unnamed pieces of code, for instance (omitting the typing information), in the syntax*

$$let\, x = new[l_1 = \varsigma(s).\lambda(\vec{x})t_1, \ldots]\, in\, t_2\ .$$

[11] Note that $self.\Theta$ corresponds more to a instance-local view of E_Θ rather than Θ, and can also contain objects from Δ.

This means that one has to deal with the problem of generating a fresh identity for instantiation, but not with the fact that two instances possess the same behavior up to their name, since there are never two *instances of the same class. Thus the situation corresponds to the use of classes according to the singleton design pattern.* □

Remark 3.3.8 (Class variables)**.** *With class variables, also called static variables, the incarnation problem for objects simplifies. With (mutable) static variables, two points change, one concerning connectivity, the second one the question of incarnation.*

First, class variables offer a "communication channel", i.e., information can flow from otherwise separate objects via the class variables. Assuming that their visibility is "`private`*" (in the sense of* Java*) means that they cannot be used universally for coordination, but only by the instances of the class in question. As a consequence, all instances of a given class belong to the same clique, in our terminology.*

Concerning the second point, class variables can be used to keep track of the number of instantiations of a class. Consequently, two instantiations of the same class are not longer identical up to their name, as the order of instantiations may influence their behavior. In other words, the phenomenon of replay *vanishes from the closure conditions of the traces, which is a major(!) simplification. See also Section 6.1.2.* □

In a non-deterministic setting, e.g., when considering multiple threads, a possible solution could be that the object simply *guesses* which future it is supposed to play; if it turns out (after some interaction) to be the wrong choice, it simply blocks. In our setting, guessing is not an option.

Instead, each object, resp., each clique maintains a representation of *all* possible futures during the run of the program and while working off the future and based on the past interaction, it narrows the still open options. In other words: Without non-determinism and with the class(es) containing the description of the future for all instances, the implementation must explore all options "in parallel", weeding out those which, during the run, turn out to be inconsistent with the witnessed past behavior. At this point, an example may help.

Example 3.3.9 (Roles and scripts)**.** *Consider the following trace, where o_1 and o_2 are instances of the same class c.*

$$\begin{aligned}&\nu(o_1{:}c).\langle call\ o_1.l()\rangle?\ \langle return(o_1)\rangle!\ \langle call\ o_1.l_1()\rangle?\ \langle return(a)\rangle! \\ &\nu(o_2{:}c).\langle call\ o_2.l()\rangle?\ \langle return(o_2)\rangle!\ \langle call\ o_2.l_2()\rangle?\ \langle return(b)\rangle!\ .\end{aligned} \tag{3.17}$$

In this situation, the class c of the two instances o_1 and o_2 must contain the description of the two possible futures, which, up to the second call, are indistinguishable. The second incoming call l_1 vs. l_2 is the distinguishing interaction in this example. This means, up to this point, the respective instance cannot know whether it must behave according to the given behavior of o_1 or o_2 and, indeed, both behave the same, and the object must "keep both options open" until the distinguishing interaction occurs. Especially, the following reaction, the return, must be equal (up to naming) in the deterministic setting, since the first incoming call does not contain enough information to distinguish the two behaviors so far.

When either l_1 or l_2 occurs, it becomes clear that from now on the object must behave like o_1 in the trace of equation (3.17) or like o_2. Of course, the implementation cannot assure that the object concretely carries the identifier o_1, resp., o_2, as the semantics is invariant under α-renaming. *However, when confronted with, say, l_2 in*

the second step, the object, whatever its identifier, must behave from now on the way, o_2 behaves in the above trace; for instance, it must return an b and not a label a in the next step.[12]

In the given (fixed) trace, we call an identity of an object in the trace a role *(or rather, we will later use the word role for the instance variable used to store that identity). In the above example, a new instance of c is expected to play exactly one of two possible roles, the one as witnessed by o_1 in the given trace, and one for o_2.*

When coding *statically the intended behavior(s) in a class, the roles are represented by (appropriately typed) instance variables. In the trace of equation (3.17), the component thus contains two instance variables, i.e., roles x_1 and x_2 of type c, initially undefined. The* association *of roles with currently known objects is kept in a data structure with typical elements σ, σ'_1, ... Initially, the values in the examples are $\sigma_1 = \bot$ and $\sigma_2 = \bot$ below, i.e., the finite mappings $[x_1 \mapsto \bot, x_2 \mapsto \bot]$. The future behavior of the instances of that class is coded in terms of these instance variables and thus the uninitialized future of instances of this class is represented as follows, where for both scripts the association part is "empty", i.e., $\sigma_1 = \bot$ and $\sigma_2 = \bot$:*

$$\begin{array}{lcl} \check{s}_1 & = & x_1 \mapsto \nu(x_1{:}c).\langle call\ x_1.l()\rangle?\ \langle return(x_1)\rangle!\ \langle call\ x_1.l_1()\rangle?\ \langle return(a)\rangle! \\ \check{s}_2 & = & x_2 \mapsto \nu(x_2{:}c).\langle call\ x_2.l()\rangle?\ \langle return(x_2)\rangle!\ \langle call\ x_2.l_2()\rangle?\ \langle return(b)\rangle!\ . \end{array} \tag{3.18}$$

The values of the two instance variable $script_1$ and $script_2$ consists of the pair $(\sigma_1, \check{s}_1)$, resp., $(\sigma_2, \check{s}_2)$ and can be thought of as (structured) instance variables, as well. □

The coding of the scripts is shown in more detail later; that the coding is possible rests of the fact that all encoded entities (the number of roles, the number and the length of the scripts, the number of different method labels, etc.) are finite, given a finite trace to be represented.

The above example illustrates the *static* representation, i.e., the coding of the intended behavior of a program within classes and in terms of roles and scripts. Next we illustrate the *dynamic* aspects of the data representations, i.e., how they change during the run of the program. As mentioned, the current state of an object (or, conceptually, the clique) is represented by identifying which futures are still possible, which includes which roles are still possible, given the past interaction. Confronted with an input, the object (or clique) checks which scripts in the current state are consistent with the input and *shortens* the respective future, perhaps filling in more roles. We call this "playing the scripts".

Example 3.3.10 (Playing a script)**.** *Consider the trace from equation (3.17), resp., the script coding of equation (3.18). Let us assume that the behavior corresponding to the first script is what actually happens. Wlog., assume further that the actual name of the object in the run is o_1, i.e., the identity of the object happens to equal the identity from the trace of equation (3.17); any other would do as well.*

In the illustration, the shortening of the scripts is directly represented by shortening the future traces in the $scripts_i$ instance variables. Upon instantiation, the object o_1 is in the state as given by equation (3.18); without constructor, the state of the new instance must equal the state as given by the class. To execute the first call

$$a = \langle call\ o_1.l()\rangle?\ ,$$

[12] We assume for sake of the argument that a and b are two different reactions, left unspecified. Also l_1 and l_2 are assumed to be different, of course.

object o_1 in its initial state checks which of the two possible scripts is consistent with the witnessed a; in this case, both. The state after playing a can be represented as (we omit here the association of the two lines to a role, as it is clear from the bindings):

$$\begin{array}{ll} \sigma_1^1 = [x_1 \mapsto o_1] & \check{s}_1^1 = \langle return(x_1)\rangle!\ \langle call\ x_1.l_1()\rangle?\ \langle return(a)\rangle! \\ \sigma_2^1 = [x_2 \mapsto o_1] & \check{s}_2^1 = \langle return(x_2)\rangle!\ \langle call\ x_2.l_2()\rangle?\ \langle return(b)\rangle!\ . \end{array} \tag{3.19}$$

Playing the first script, the incoming communication a associates the role x_1 with the actually witnessed identity o_1 and leaves the role x_2 undefined; in case of $script_2$ in the second line, o_1 takes the role of x_2. Continuing the interaction, the object answers with a return in both situations. Using σ_1^1, resp., σ_2^1, the value returned is identically o_1 in both cases. In the state before the incoming second call $\langle call\ o_1.l_1()\rangle?$, the futures look as follows; playing the return does not add more constraints, i.e., it leaves σ_1^1 and σ_2^1 unchanged:

$$\begin{array}{ll} \sigma_1^2 = [x_1 \mapsto o_1] & \check{s}_1^1 = \langle call\ x_1.l_1()\rangle?\ \langle return(a)\rangle! \\ \sigma_2^2 = [x_2 \mapsto o_1] & \check{s}_2^2 = \langle call\ x_2.l_2()\rangle?\ \langle return(b)\rangle! \end{array} \tag{3.20}$$

Matching $\langle call\ o_1.l_1()\rangle?$ succeeds when playing the first script, but fails with the second. I.e., after playing the second call, there remains only one possible future:

$$\sigma_1^2 = [x_1 \mapsto o_1] \quad \check{s}_1^2 = \langle return(a)\rangle!\ . \tag{3.21}$$

□

The following example illustrates the handling of data structures, when two cliques are merged. In this case, information distributed across two different cliques needs to be combined.

Example 3.3.11 (Merging)**.** *Consider the following trace:*

$$\begin{array}{l} \nu(o_1, o_3{:}c).\langle call\ o_1.l_1(o_3)\rangle?\langle return()\rangle! \\ \nu(o_2{:}c).\langle call\ o_2.l_2()\rangle?\langle return()\rangle! \\ \langle call\ o_1.l(o_2)\rangle?\ . \end{array} \tag{3.22}$$

The trace is the one from Example 3.1.4, equation (3.2). See also Figure 3.1 for a schematic tree representation. Remember, from Example 3.1.4, that the merging action $\langle call\ o_1.l(o_2)\rangle?$ is represented in the figure as

$$\nu(o_2).\langle call\ o_1.l(o_2)\rangle? \quad resp. \quad \nu(o_1).\langle call\ o_1.l(o_2)\rangle?$$

when seen from the perspective of o_1, resp., from o_2's perspective. This captures the fact that the identity o_2 is new *for the clique of o_1, and conversely, o_1 is new to the clique of o_2. These two clique-local views onto the global label $\langle call\ o_1.l(o_2)\rangle?$ corresponds to the treatment of ν-binders for projection (cf. Definition 3.1.3).*

Given the above trace, the set of static scripts contains the following three possible futures, using x_1, x_2, and x_3 as roles:

$$\begin{array}{l} x_1 \mapsto \nu(x_1, x_3).\langle call\ x_1.l_1(x_3)\rangle?\ \langle return()\rangle!\ \nu(x_2).\langle call\ x_1.l_2(x_2)\rangle?\ , \\ x_3 \mapsto \nu(x_1, x_3).\langle call\ x_1.l_1(x_3)\rangle?\ \langle return()\rangle!\ \nu(x_2).\langle call\ x_1.l_2(x_2)\rangle?\ , \\ \nu(x_2).\langle call\ x_2.l_2()\rangle?\ \langle return()\rangle!\ \nu(x_1).\langle call\ x_1.l(x_2)\rangle?\ . \end{array} \tag{3.23}$$

After the first four global actions, i.e., before the merge, the component consists of two separate cliques, each with a separate view concerning the future. In the clique of o_1

(and o_3), only one future is still possible, since the call with method l_1 in the past of o_1's clique invalidated the second script:

$$\sigma_1^2 = [x_1 \mapsto o_1, x_3 \mapsto o_3] \quad \{\nu(x_2).\langle call\ x_1.l(x_2)\rangle?\}\ . \tag{3.24}$$

The script of equation (3.24) is available twice, once for x_1 and once for x_3, but already after the first step, they are identical. For o_2's clique, the possible future presents itself as

$$\sigma_2^2 = [x_2 \mapsto o_2] \quad \{\nu(x_1).\langle call\ x_1.l(x_2)\rangle?\}\ . \tag{3.25}$$

As said, the incoming call $a = \langle call\ o_1.l(o_2)\rangle?$ merges the two cliques. Matching the actual label a against the only possible expected one from equation (3.23), clique o_1's post configuration then is of the form

$$[x_1 \mapsto o_1, x_2 \mapsto o_2, x_3 \mapsto o_3]$$

and the same holds for the post configuration of o_2's clique. Indeed, o_1, o_2, and o_3 belong to the same clique after the merge and must from now on have the same view upon the future. In the example, there is only one future left, corresponding to the empty trace. □

The next example is slightly more complex in that it shows how the possible roles are *narrowed* during a run (as opposed to being cancelled out altogether).

Example 3.3.12 (Merging)**.** *Let the trace t be given as*

$$\begin{array}{rcl} t(o_1,o_2) & \triangleq & \nu(o_1).\langle call\ o_1.l_0()\rangle?\langle return()\rangle! \\ & & \nu(o_2).\langle call\ o_2.l_0()\rangle?\langle return()\rangle! \\ & & \langle call\ o_1.l_1(o_2)\rangle?\langle return()\rangle!\ , \end{array} \tag{3.26}$$

and further $t(o_1', o_2')$ the variant with o_1 and o_2 renamed to o_1' and o_2'. Consider the trace s which performs $t(o_1, o_2)$ and $t(o_1', o_2')$, followed by an interaction which allows to distinguish the cliques $[o_1, o_2]$ and $[o_1', o_2']$ which exist after $t(o_1, o_2)t(o_1', o_2')$:

$$\begin{array}{rcl} s(o_1,o_2,o_1',o_2') & \triangleq & t(o_1,o_2)\ t(o_1',o_2') \\ & & \langle call\ o_1.l_2(o_2)\rangle?\langle return()\rangle! \\ & & \langle call\ o_2'.l_2(o_1')\rangle?\langle return()\rangle!\ . \end{array} \tag{3.27}$$

The distinguishing action is the call of l_2, in the first clique a call to o_1 with o_2 as parameter and in the second clique a call to o_2' with o_1' as argument. This allows to distinguish the two cliques, since at that point, o_1 and o_2, resp., o_1' and o_2', are distinguishable, where the merging action $\langle call\ o_1.l_1(o_2)\rangle?$ prior in the trace separated the behavior of o_1 from that of o_2, resp., $\langle call\ o_1'.l_1(o_2')\rangle?$ for o_1' and o_2'.

A tree representation of the behavior is shown Figure 3.3. We use $a(o_1, o_2)$ as abbreviation for $\langle call\ o_1.l_1(o_2)\rangle?$ and analogously for $a(o_1', o_2')$. Since s contains four different object identities, the static encoding foresees four roles, $x_1, x_2, x_1', $ and x_2'. Furthermore, there are initially four scripts, one for each role (we write the script in an abbreviated form: $r!$ stands for a $\langle return()\rangle!$ and calls are written shorter $x.l(y)$):

$$\begin{array}{l} \{\ \sigma_\perp : \nu(x_1).x_1.l_0()?\ r!\ \nu(x_2).x_1.l_1(x_2)?\ r!\ x_1.l_2(x_2)?\ r! \\ \quad \sigma_\perp : \nu(x_2).x_2.l_0()?\ r!\ \nu(x_1).x_1.l_1(x_2)?\ r!\ x_1.l_2(x_2)?\ r! \\ \quad \sigma_\perp : \nu(x_1')x_1'.l_0()?\ r!\ \nu(x_2').x_1'.l_1(x_2')?\ r!\ x_2'.l_2(x_1')?\ r!) \\ \quad \sigma_\perp : \nu(x_2').x_2'.l_0()?\ r!\ \nu(x_1').x_1'.l_1(x_2')?\ r!\ x_2'.l_2(x_1')?\ r!\ \ \}\ . \end{array}$$

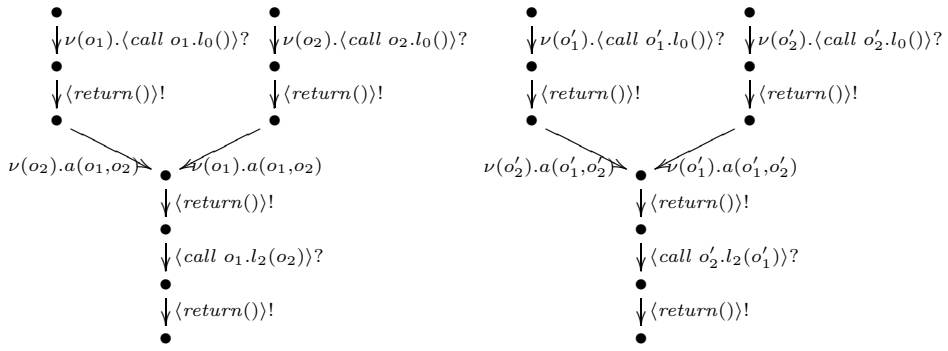


Figure 3.3: Trace of equation (3.27), tree representation

Now assume that the following trace happens, with $t(o_1, o_2)$ given by equation (3.26):

$$t(o_1, o_2)\ o_1.l_2(o_2)?\ r!\ . \tag{3.28}$$

The trailing incoming call $o_1.l_2(o_2)?$ is the interaction which distinguishes between the trees on the left-hand and the right-hand side of Figure 3.3 and it corresponds to the tree on the left. Of course, the exact identities o_1 and o_2 are irrelevant.

After the first call $\nu(o_1).o_1.l_0()?$ and the subsequent return, the potential future behaviors look as follows:

$$\begin{array}{ll}\{ & [x_1 \mapsto o_1] : \nu(x_2).x_1.l_1(x_2)?\ r!\ x_1.l_2(x_2)?\ r! \\ & [x_2 \mapsto o_1] : \nu(x_1).x_1.l_1(x_2)?\ r!\ x_1.l_2(x_2)?\ r! \\ & [x'_1 \mapsto o_1] : \nu(x'_2).x'_1.l_1(x'_2)?\ r!\ x'_2.l_2(x'_1)?\ r! \\ & [x'_2 \mapsto o_1] : \nu(x'_1).x'_1.l_1(x'_2)?\ r!\ x'_2.l_2(x'_1)?\ r!\ \}_{[o_1]}\ .\end{array} \tag{3.29}$$

The subscript $[o_1]$ is meant to indicate that (3.29) describes the value of the data structures in the clique $[o_1]$, currently consisting of o_1 in isolation. In this state, $[o_1]$ is also the only clique which exists. The next (global) input $\nu(o_2).o_2.l_0()?$ does not affect this clique, but creates a new one for o_2, whose data structures after the execution of the input (and after the subsequent return) look analogous to equation (3.29), only that the identity o_2 instead of o_1 is associated with the roles $x_1, \ldots, x'_2$:

$$\begin{array}{ll}\{ & [x_1 \mapsto o_2] : \nu(x_2).x_1.l_1(x_2)?\ r!\ x_1.l_2(x_2)?\ r! \\ & [x_2 \mapsto o_2] : \nu(x_1).x_1.l_1(x_2)?\ r!\ x_1.l_2(x_2)?\ r! \\ & [x'_1 \mapsto o_2] : \nu(x'_2).x'_1.l_1(x'_2)?\ r!\ x'_2.l_2(x'_1)?\ r! \\ & [x'_2 \mapsto o_2] : \nu(x'_1).x'_1.l_1(x'_2)?\ r!\ x'_2.l_2(x'_1)?\ r!\ \}_{[o_2]}\ .\end{array} \tag{3.30}$$

The next step $o_1.l_1(o_2)?$ merges the cliques of o_1 and o_2 and contains enough information to distinguish between o_1 and o_2. More precisely, it contains enough information to distinguish between the roles x_1 and x'_1 on the one hand and x_2 and x'_2 on the other. Before the communication, as witnessed by equation (3.29) and (3.30), o_1 and o_2 can both play the role of x_1 and x_2 or vice versa (cf. also Example 3.3.11). After the merge, two possible futures have turned out inconsistent (the second and the fourth one from the perspective of o_1 in (3.29), and the first and third one from the perspective of o_2 in (3.30) and the post-state is given by:

$$\begin{array}{ll}\{ & [x_1 \mapsto o_1, x_2 \mapsto o_2] : x_1.l_2(x_2)?\ r! \\ & [x'_1 \mapsto o_1, x'_2 \mapsto o_2] : x'_2.l_2(x'_1)?\ r!\ \}_{[o_1,o_2]}\ .\end{array} \tag{3.31}$$

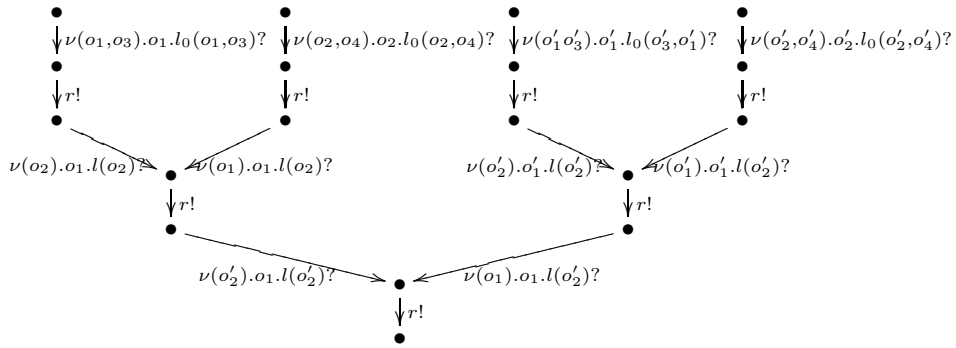


Figure 3.4: Trace of equation (3.34), tree representation

Finally, the communication $o_1.l_2(o_2)?$ is compatible only with the future and the association in the first line of equation (3.31), which distinguishes between the left and the right tree of Figure 3.3. □

The examples so far are simplified in that the merging leads to the same conclusion concerning the future for both cliques participating in the merge. In the next example, the merge must lead also to a *combined* view on the future.

Example 3.3.13 (Merging). *Let the trace t_1 be given as*

$$\begin{aligned} t_1(\vec{o}) = t_1(o_1,o_2,o_3,o_4) \quad \triangleq \quad & \nu(o_1,o_3).o_1.l_0(o_1,o_3)?r! \\ & \nu(o_2,o_4).o_2.l_0(o_2,o_4)?r! \\ & o_1.l(o_2)?r! \end{aligned} \tag{3.32}$$

and further t_2 be defined as

$$\begin{aligned} t_2(\vec{o}') = t_2(o'_1,o'_2,o'_3,o'_4) \quad \triangleq \quad & \nu(o'_1,o'_3).o'_1.l_0(o'_3,o'_1)?r! \\ & \nu(o'_2,o'_4).o'_2.l_0(o'_2,o'_4)?r! \\ & o'_1.l(o'_2)?r!\,. \end{aligned} \tag{3.33}$$

In what follows, we use $\vec{o}$ as short-hand for o_1,o_2,o_3,o_4, similarly $\vec{o}'$ for $o'_1,\ldots,o'_4$ and $\vec{x}$ for $x_1,\ldots,x_4$, etc. Note that the order of the arguments to the first call is reversed, comparing (3.32) and (3.33). Thus, unlike the traces $t(o_1,o_2)$ and $t(o'_1,o'_2)$ from Example 3.3.12, the traces $t_1(o_1,o_2,o_3,o_4)$ and $t_2(o'_1,o'_2,o'_3,o'_4)$ are not α-equivalent. Another difference to Example 3.3.12 is that the two traces continue by merging the two remaining cliques. So consider the trace s which performs $t_1(\vec{o})$ and $t_2(\vec{o}')$ followed by an interaction which merges the cliques $[o_1,o_2,o_3,o_4]$ and $[o'_1,o'_2,o'_3,o'_4]$ which exist after $t(\vec{o})\ t(\vec{o}')$:

$$s(\vec{o},\vec{o}') \quad \triangleq \quad t_1(\vec{o})\ t_2(\vec{o}')\ o_1.l(o'_2)?r! \tag{3.34}$$

A tree representation of the behavior is shown Figure 3.4.

The initial, static configuration looks as follows:[13]

$$\begin{array}{ll}\{ & \sigma_\perp : \nu(x_1,x_3).x_1.l_0(x_1,x_3)?\ r!\ \nu(x_2).x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r! \\ & \sigma_\perp : \nu(x_2,x_4).x_2.l_0(x_2,x_4)?\ r!\ \nu(x_1).x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r! \\ & \sigma_\perp : \nu(x_1',x_3').x_1'.l_0(x_3',x_1')?\ r!\ \nu(x_2').x_1'.l(x_2')?\ r!\ \nu(x_1).x_1.l(x_2')?\ r! \\ & \sigma_\perp : \nu(x_2',x_4').x_2'.l_0(x_2',x_4')?\ r!\ \nu(x_1').x_1'.l(x_2')?\ r!\ \nu(x_1).x_1.l(x_2')?\ r!\ \}\ .\end{array} \tag{3.35}$$

Now assume that the following trace happens, with $t_1(\vec{o})$ given by equation (3.32):

$$t_1(\vec{o})\ o_1.l(o_2)?\ . \tag{3.36}$$

The first call $\nu(o_1,o_3).o_1.l(o_1,o_3)?$ now already distinguishes between the behavior of t_1 and that of t_2. After that call and after the subsequent return, the potential future behavior looks as follows:

$$\begin{array}{ll}\{ & [x_1\mapsto o_1, x_3\mapsto o_3] : \nu(x_2).x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r! \\ & [x_2\mapsto o_1, x_4\mapsto o_3] : \nu(x_1).x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r! \\ & [x_2'\mapsto o_1, x_4'\mapsto o_3] : \nu(x_1').x_1'.l(x_2')?\ r!\ \nu(x_1).x_1.l(x_2')?\ r!\ \}_{[o_1,o_3]}\ .\end{array} \tag{3.37}$$

This means, the third line of equation (3.35) has been invalidated, the other three remain open. The next interaction $\nu(o_2,o_4).o_2.l_0(o_2,o_4)?$ creates a new clique (and leaves the data structures of the clique $[o_1,o_3]$ unchanged). After the call, the state of the new $[o_2,o_4]$ clique looks as follows

$$\begin{array}{ll}\{ & [x_1\mapsto o_2, x_3\mapsto o_4] : \nu(x_2).x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r! \\ & [x_2\mapsto o_2, x_4\mapsto o_4] : \nu(x_1).x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r! \\ & [x_2'\mapsto o_2, x_4'\mapsto o_4] : \nu(x_1').x_1'.l(x_2')?\ r!\ \nu(x_1).x_1.l(x_2')?\ r!\ \}_{[o_2,o_4]}\ ,\end{array} \tag{3.38}$$

i.e., up to the fact that different object identities are stored in the roles, the states of equation (3.37) and (3.38) are equivalent. Note in passing that the "range" of the mappings from roles to identities is identical for each script (in each clique separately). Indeed, the stored identities form the current clique. Another invariant concerns the "domain" of the mapping, i.e., the set of currently chosen roles: In each clique it is the case that for each distinct pair of scripts, the domains are disjoint.

The next incoming label is of the form $o_1.l(o_2)?$ and merges the two cliques $[o_1,o_3]$ and $[o_2,o_4]$. We start with the behavior of the callee clique, i.e., the clique of o_1. The script labels contain the information, which identities are new from the local perspective; for instance, the script in the first line expects identities for x_2, whereas x_1 and x_3 are already filled in. Besides the identity of x_2, the partner clique contributes also the identity for x_4, which is new for clique $[o_1,o_3]$ but not mentioned in the label. Concentrating on the first line of equation (3.37): It is an invariant that from the state of its partner, i.e., the state of o_2's clique in equation (3.38), there is at most *one script that offers the dual of the required roles; in the example, only the second line of equation (3.38) offers x_2 and x_4.*

The second line of $[o_1,o_3]$ matches, as far as the exchange of identities and roles are concerned, with the first line of $[o_2,o_4]$. The last line of $[o_1,o_3]$ does not find a partner: No still open future of $[o_2,o_4]$ offers the required identities for x_1' and x_3' (the corresponding script has "died out").

[13] We show only half of the scripts, we do not separately list the futures corresponding to o_1 and o_3, for instance.

After $[o_1, o_3]$ has received the identities, the (intermediate) state looks as follows (the newly filled roles are underlined, the third line of equation (3.37) has been removed):

$$\{\ [x_1 \mapsto o_1, \underline{x_2} \mapsto o_2, x_3 \mapsto o_3, \underline{x_4} \mapsto o_4] : x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r! \\ [\underline{x_1} \mapsto o_2, x_2 \mapsto o_1, \underline{x_3} \mapsto o_4, x_4 \mapsto o_3] : x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r!\ \}_{[o_1,o_3]}\ . \tag{3.39}$$

From the perspective of $[o_2, o_4]$, the last line of equation (3.38) does not find a partner and is thus removed; the other two are still possible:

$$\{\ [x_1 \mapsto o_2, \underline{x_2} \mapsto o_1, x_3 \mapsto o_4, \underline{x_4} \mapsto o_3] : x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r! \\ [\underline{x_1} \mapsto o_1, x_2 \mapsto o_2, \underline{x_3} \mapsto o_3, x_4 \mapsto o_4] : x_1.l(x_2)?\ r!\ \nu(x_2').x_1.l(x_2')?\ r!\ \}_{[o_2,o_4]}\ , \tag{3.40}$$

The state is intermediate, since so far only the new identities *have been consistently exchanged, which corresponds loosely to the binding part of the label; the core of the communication* label *has not yet been evaluated which is why it is still mentioned in the scripts at the current stage, with the ν-binders removed.*[14]

Evaluating also the core $o_1.l(o_2)$ of the call with the current possible bindings and matching it against $x_1.l(x_2)$ cancels out the second line of equation (3.39) for $[o_1, o_3]$ and the first line of (3.40) for $[o_2, o_4]$. Both cliques have reached agreement and thus the now common state after the merge (and after the subsequent return) is:

$$\{\ [x_1 \mapsto o_1, x_2 \mapsto o_2, x_3 \mapsto o_3, x_4 \mapsto o_4] : \nu(x_2').x_1.l(x_2')?\ r!\ \}_{[o_1,o_2,o_3,o_4]}\ . \tag{3.41}$$

Note that from the original, static arrangement of scripts from equation (3.35), line 1 has survived in equation (3.39) and line 2 in equation (3.40) which are merged into (3.41). In other words: When merging two cliques, one cannot just statically compare scripts as they appear in the original code, for instance match line 1 of (3.35) in one clique against the same line in the partner clique.

Having arrived at (3.41), the clique $[o_1, \ldots, o_4]$ has only one possible future left open. The next possible step from the perspective of $o_1, \ldots, o_4$ with the current role bindings, is $o_1.l(o_2')?$, provided the partner clique, whose derivation is not shown, is able provide appropriate bindings for the roles $x_1', \ldots x_4'$, in particular, $x_2' \mapsto o_2'$ for some o_2' which is new for the clique $[o_1, \ldots, o_4]$. □

The next example illustrates the behavior of newly instantiated objects.

Example 3.3.14 (Initialization)**.** *Consider the following global trace*

$$t = \nu(o_1{:}c, o_2{:}c).o_1.l(o_2)?\ r!\ , \tag{3.42}$$

i.e., two instances of a component class c are created by the same incoming call. Thus, the class contains two roles, x_1 and x_2 and two corresponding scripts. Unlike the previous examples where we did not mention for simplicity the classes of the objects in the traces or in the scripts, now we are explicit about the class, here c, since the class is needed as template for the newly instantiated objects.

$$\begin{aligned} \mathit{init} &= [\ x_1 \mapsto \nu(x_2{:}c).x_1.l(x_2)?\ r! \\ &\quad\ \ x_2 \mapsto \nu(x_1{:}c).x_1.l(x_2)?\ r!\ ]\ ; \\ \mathit{scripts} &= \bot\ ; \end{aligned} \tag{3.43}$$

[14] The actual implementation will not actually first remove the ν-binding part from the scripts stored in the instance state and afterwards the core of the label. Conceptually, when executing the code, the two stages are performed in the mentioned order. When successful, the scripts are shortened by removing the whole label in one step.

Initially, the scripts data structure is undefined. The static code for all scripts is kept in $init$. The two mentioned object references o_1 and o_2 give rise to two roles x_1 and x_2, respectively. Each role is associated with one unique future in terms of their roles.

The "ν"-binders for roles in the script are now interpreted slightly differently from the intuition of ν-binders in traces (such as that from equation (3.42) in this example). In a trace and in the semantics, the ν acts as a binder, and object references are interpreted always up to α-renaming. In the scripts in equation (3.43) or (3.44), which can be seen as a static representation of one fixed trace, (νx) is not understood as a binder for the rest of the script. In particular we are not at liberty to rename the "binding occurrence" in a capture avoiding way in one script.[15]

The incoming call $\nu(o_1{:}c, o_2{:}c).o_1.l(o_2)?$ at the beginning of the trace creates two new instances; that's how the semantics, independently of any encoding, deals with lazy instantiation (as said, c is a component class). The encoding must then initialize the sketched dynamic data structures appropriately. We treat the freshly created two objects o_1 and o_2 as two separate cliques. Separate at least for a short moment, until they are merged by the label in the same way, as cliques existing already for a longer time are merged (cf. the other examples). Object o_1 thus forms a clique consisting only of itself. With no other information evaluated, the self-identity o_1 can play any *of the foreseen roles, in this case x_1 and x_2. Therefore, the state of the clique $[o_1]$ after initialization looks as follows (the value of $init$ does no longer play a role and not shown again; it is only used once to fill in the initial value of $scripts$):*

$$\begin{array}{rcl} init & = & [\,\ldots\,] \\ scripts & = & \{\ [x_1 \mapsto o_1] : \nu(x_2{:}c).x_1.l(x_2)?\ r! \\ & & \ \ [x_2 \mapsto o_1] :\ \nu(x_1{:}c).x_1.l(x_2)?\ r!\ \}_{o_1}\ . \end{array} \tag{3.44}$$

Object o_2 executes the same initialization phase, reaching an equivalent state, where o_1 is replaced by o_2 in the role associations.

After this phase, two properly initialized cliques exist. Indeed, the two cliques are now no longer in a specific "initial" state; for instance, the state of the clique $[o_1]$ contains no information whether it has just been created or whether it has undergone already a number of calls and returns (without contact to other objects), since cliques need not record the history. Consequently, the rest of the behavior follows exactly *the merge protocol as seen in the other examples.* □

Remark 3.3.15 (Initialization and lazy instantiation)**.** *As an aside: The first initialization leading to the code of equation (3.44) can be understood as the coder's perspective on* lazy instantiation*. In absence of constructors, objects created by of cross-border instantiation are actually created only when first accessed by a method call (for instance at that moment in the example). The actual new-statement of the (absent) environment may have been executed earlier. If the language had constructors, any object creation would be observable immediately, and furthermore the constructor could be used to set up the data structures directly after creation. The initialization illustrated in Example 3.3.14 can be interpreted as the "trivial" constructor executed in a lazy manner, i.e., immediately preceding the method call proper, and trivial in the sense that it uses the* only *dynamic piece of information handed over to the object, namely the identity as value of self, i.e., the value of the ς-parameter.* □

[15] One could, of course, program the whole component with a differently chosen set of instance variables, as far as the names in concrete syntax are concerned. In that way —of course— the representation is invariant under renaming. However, the aspect of dynamic scoping is missing.

Data structures and algorithms

Next we describe the data structures contained in each class. Definition 3.3.16 describes only the form or "type", not the exact values; they will be filled in in the construction of Definition 3.3.20. Cf. also Definition B.2.1 in the appendix. In the following, when speaking about *fields* and their *types* we mean collections or compound ensembles of basic fields in the calculus which encodes the data structure appropriately. In general we use *italics* (or mathematical symbols) for the encoded methods and fields. For the encoded types, we use sanserif (or also mathematical notations such as $\times$ for product types).

Definition 3.3.16 (Data structures)**.** *Each class contains the fields $script$ containing the current futures and $init$, containing the initial, static representation of $scripts$. In overview and ignoring "overloading", the interface type for each class is of the form:*

$$\left[\!\!\left(\begin{array}{l} init, scripts : \mathsf{set\ of\ script} \\ step^i : \mathsf{label} \times (\mathsf{set{:}object}) \to \mathsf{Unit} \\ step^o : \mathsf{Unit} \to \mathsf{Unit} \\ \hline l : \vec{T} \to T \\ \vdots \end{array}\right]\!\!\right] .$$

The vertical "..." refer to further methods and fields contained in the class which are independent from the definability construction —only one method l is shown here— but whose presence is required by the given commitment context. For the sets, we use the mathematical notation such as $\{\_\}$, $\cup$, etc. For the lists, we write $[a_1, a_2, \ldots]$, and $a :: s$ for extending the list by the new element at the head. Those notations are used in the construction *of C_t. Thus they have no run-time significance and are used as meta-mathematical notation to describe the constructed program. The concrete implementation of the types and values and those access functions needed at run-time to access and manipulate the data are shown later. Further, we use the following "type" abbreviations:*

$$\begin{array}{rcl} \mathsf{script} & \triangleq & \mathsf{assoc} \times \mathsf{future} \\ \mathsf{scripts} & \triangleq & \mathsf{set\ of\ script}\ . \end{array} \tag{3.45}$$

The methods $step^i$ and $step^o$ (and many more auxiliary methods described in the appendix) are "private" in that they are hidden to the environment using subsumption. $step^i$ and $step^o$ are the top-level method responsible for "playing the scripts", i.e., for shortening the script data structures step by step. $step^i$ does so for incoming labels and $step^o$ for outgoing.

Next we fix the notation and the conventions for the *static* representation of the run-time identities. They form the core of the data representation, similarly as *names* are the simplest entity in the syntax and semantics: In the abstract syntax from Table 2.2, names are one of the two forms of *values* of the calculus (the other form are local variables.) The names are *dynamic* in nature: they are created during the run of the program and they cannot occur in the static code of classes. The names occurring in a trace of the semantics are represented in the code by instance variables:

Definition 3.3.17 (Static representation)**.** *Given the legal trace t to realize. Each object identity o of a component class in the trace is represented by an instance variable*

$\check{o}$ of the corresponding class type. For object identities $o_1, o_2', \ldots$, we also refer to the corresponding instance variable as $x_1, x_2', \ldots$, when the connection is clear. We apply the same convention also to compound entities such as labels and traces, i.e., the static representation of a basic label γ is denoted by $\check{\gamma}$, of a label a as $\check{a}$, and of a trace t as $\check{t}$, where each occurrence of an object reference o is replaced by its static instance variable.

For instance, given a label $a = \nu(o_1{:}c_1, o_2{:}c_2).\langle call\ o_2.l(o_1, o_3)\rangle?$, the static equivalent $\check{a}$ is $\nu(x_1{:}c_1, x_2{:}c_2).\langle call\ x_2.l(x_1, x_3)\rangle?$, where the x_i are instance variables. As a manner of speaking, we call the instance variables for object references, i.e., the static representations of names from the given trace, *roles*.

In the constructed component C_t, each clique contains a static, linear representation of its still open future plus an *association* from roles (instance variables) to actual object identities. Informally, we can associate the type "role $\rightarrow$ object" to the mentioned associations. However, this "function" is given only *implicitly*, in that the roles are nothing else than a statically given set of instance variables, and the (finite) association is given by storing the object identity associated with the role in that instance variable (see also Remark 3.3.19 below).

The association of roles with objects is maintained as an abstraction of the clique's past. We call a pair of role-name association and future a *script.* Since, due to replay, a clique may have more than one potential future, the central data structure $scripts$ is a set of scripts. Initially, the scripts contain empty associations together with the static analogs of all possible futures, as given in equation (3.46) (cf. also Definition 3.1.5 for the definition of linear paths from the local viewpoint of an object, resp., a clique, based on the notion of projection).

We fix a number of notations to facilitate the coding and the reasoning about the component C_t. A more detailed implementation of the data structures used can be found in Appendix B. See also Lemma A.5.13 for some invariants concerning the data structures mentioned in the introduced notations.

Notation 3.3.18. *With the data structures given as in Definition 3.3.16, we define a number of "views" on the state. Given a component object o, $o.scripts$ refers to the set of scripts as value. We use $\sigma, \sigma_i, \ldots$ for "values" of type* assoc*, called associations or substitutions. The range of an association is denoted by $ran(\sigma)$, the domain by $dom(\sigma)$. Furthermore, $ran_\Theta(\sigma)$ denotes the instance variables contained component object references from $ran(\sigma)$ (i.e., those instance variables typed by a component class and not containing $\bot$), and analogously $ran_\Delta(\sigma)$ to the instance variable containing environment object references. By $o.\Sigma$, we refer to the set of scripts as value. Furthermore, $o.\Xi$ refers to $ran(\sigma)$ for some/all σ from $o.script$, $o.\Theta$ to the component objects from $o.\Xi$, and $o.\Delta$ to the environment objects from $o.\Xi$.*[16]

Given $\Xi_0 \vdash C_t \stackrel{r}{\Longrightarrow} \Xi \vdash C$ and a component clique $[o]_{/E_\Theta}$ (or $[o]$ for short) after r, then $[o].l$ means the clique of o agrees on the value of the field l and $[o].l$ is the value of $o'.l$ for some/all $o' \in [o]$.

To avoid confusion: Given $\Xi_0 \vdash C_t \stackrel{t}{\Longrightarrow} \Xi \vdash C = \Delta; E_\Delta \vdash C : \Theta; E_\Theta$, the value $o.\Theta$ does not implement the context Θ. Representing all the objects that o knows, it rather corresponds to or implements the *portion* of the connectivity context E_Θ in acquaintance with the object o. Correspondingly, $o.\Delta$ are the environment objects that o knows, i.e., it corresponds to the objects o' such that

[16] It will be an invariant that all associations σ have identical range.

$\Theta; E_\Theta \vdash o \leftrightharpoons\!\hookrightarrow o' : \Delta$. In particular $o.\Delta$ has nothing to do with the connectivity E_Δ; after all we need to implement the commitments, not the assumptions.

Remark 3.3.19 (Association)**.** *The range of an association σ is a set of object identifiers, the domain a "set of instance variables". The range $ran(\sigma)$ is unproblematic; it can be straightforwardly encoded and the implementation C_t must have access to all objects in the range of σ, since this set represents the connectivity. This means, ran must be implemented as some method. The domain $dom(\sigma)$ as a set of instance variables is not explicitly needed in the program as data structure or method, it is used as meta-mathematical notation to refer to all role instance variables, whose value is different from $\bot$. In particular we do not need a method returning a "set" of instance variable or manipulating such sets; this would amount to implement some sort of reflection.* □

Definition 3.3.20 (Observer for trace t)**.** *Assume $\Xi_0 \vdash t : trace$, i.e., t is a deterministic, legal trace. The observer for t, denoted by C_t is defined as as follows. Let C'_t be the part of C_t without (potentially) the thread, consisting of classes, only. It is given as follows. Each class mentioned in the commitment assertion Θ_0 is equipped, with the data structures typed as given in Definition 3.3.16, with $scripts = \bot$ and*

$$init = \{(\sigma_\bot, \check{t}_o) \mid t_o = {}_o{\downarrow}\, t,\ o \in names(t)\}\ . \tag{3.46}$$

Each public method $l : \vec{T} \to T$ of each component class c is implemented as

$$l \triangleq \varsigma(s{:}c).\lambda(\vec{x} : \vec{T}).t^i_{sync}(\vec{x}); t^o_{sync}\ . \tag{3.47}$$

If $\Delta_0 \vdash \odot$, then $C_t = C'_t$, i.e., C_t does not contain the thread $\natural$. If otherwise $\Theta_0 \vdash \odot$, then C_t is of the form

$$\Xi_0 \vdash C_t \quad \triangleq \quad \Xi_0 \vdash C'_t \parallel \natural\langle let\ x{:}c_i\ in\ new\ c_i\ in\ x.start()\rangle \tag{3.48}$$

for some class c_i with $\Theta \vdash c_i$.

The mentioned data structures are presented in more detail in the appendix. That the encoding is possible rests ultimately on the fact that the given trace t is finite. In particular it contains a *finite* number of object identities, which means that C_t contains also only a finite number of roles, one $\check{o}$ for each name o occurring in t. Also the number of different methods is finite.

Remark 3.3.21 (Start method and hiding)**.** *If the thread starts executing in the component, it starts its activity by invoking the $start$-method of some component class (cf. equation (3.48)). This requires at least* one *class to be present in the component, which further implies that at least one class is mentioned in the interface: $\Theta_0 \vdash c_i$. The reason is that ν-binders cannot hide classes. If we allowed this, Θ_0 could advertise no class to the environment, and in that case, the "initial class" implementing the start method could be hidden. The resulting theory would not change much. Of course, without externally visible component classes, the environment cannot call any component method. Traces with* outgoing *calls and with only incoming returns would be perfectly legal. See also Section 6.1.7.* □

The next definition is helpful to *reason* about the behavior of C_t. It basically expresses the induction hypothesis about the evolving C_t in that it asserts the

still open future of the component. To do so it uses the introduced data structures and the future *projection* from Definition 3.1.3. In particular, it connects the associations or substitutions σ_i as abstractions of the past clique interaction with the still open future scripts of the clique. It is an existential assertion; it simply states that in the current state of the component, each clique has still open a future compatible with the (rest of the) global trace s. The assertion is used for the inductive proof of definability.

Definition 3.3.22 (Future)**.** *The assertion $\Xi \vdash C :: s$ is defined as follows. For all component objects o from Θ, i.e., for all o with $\Theta \vdash o$ we are given*

$$\Xi \vdash C :: [o] \rhd {}_{[o]}\!\downarrow s \,, \tag{3.49}$$

where the component clique $[o]$ abbreviates $[o]_{/\equiv}$ and is defined wrt. E_Θ (since o is a component object). The specification of equation (3.49) is meant as follows:

$$[o].scripts = \{\ldots, (\sigma, \check{s}'), \ldots\} \tag{3.50}$$

such that ${}_{[o]}\!\downarrow s \lesssim \check{s}'\sigma$ and $ran(\sigma) = [o]$, and where $\check{s}'\sigma$ denotes the application of the substitution σ to $\check{s}'$ and $\lesssim$ denotes matching (cf. Definition B.4.13 in the appendix). We furthermore write

$$\Xi \vdash C :: [o] \rhd \bot \tag{3.51}$$

when $o'.scripts = \bot$ for all component objects o' from $[o]$.

3.3.4 Completeness argument

The next lemma states that, given a trace t, the corresponding observer C_t can indeed perform the trace. Considering t as the specification of the observer, we call the lemma "total correctness" of C_t wrt. t.

Lemma 3.3.23 (Total correctness)**.** *Let t be a legal trace and $\Xi_0 \vdash C_t$ given by Definition 3.3.20. Then $\Xi_0 \vdash C_t \stackrel{t}{\Longrightarrow}$.*

Whereas the total correctness Lemma 3.3.23 stipulates that C_t can perform the trace t, we show next that it can perform nothing else (up to unavoidable variations). We can consider the corresponding exactness Lemma 3.3.24 also as partial correctness property.

Lemma 3.3.24 (Exactness/partial correctness)**.** *Let t be a legal trace, i.e., $\Xi_0 \vdash t : trace$, and $\Xi_0 \vdash C_t$ given by Definition 3.3.20. If $\Xi_0 \vdash C_t \stackrel{r}{\Longrightarrow}$, then $\Xi_0 \vdash r \preccurlyeq_\Theta t$.*

The next lemma is the last step towards completeness. It performs the completeness proof without exploiting the knowledge that a closed program —the component together with the environment— behaves deterministic. In this respect it corresponds to the completeness proof in the multithreaded setting where programs behave non-deterministically. With concurrency, however, the completeness proof is more complex in another respect, caused by the inability to atomically observe interaction in the presence of race condition.

First we define a variant of the $\sqsubseteq_{trace}$ relation from Definition 3.1.11, which ignores the fact that closed programs are deterministic. This definition will be helpful in proving completeness where the effects of determinism are factored out. Besides being helpful in the proof, the definition is instructive insofar,

as it captures the generalization of $\sqsubseteq_{trace}$ needed if we dropped the assumption that the programs are *deterministic*, for instance if we introduced a non-deterministic choice operator to the language. Later, in the concurrent setting, $\sqsubseteq_{trace}$ will resemble the version from Definition 3.3.25, since in the presence of concurrency and race conditions, programs behave non-deterministically (cf. Definition 5.1.6).

Definition 3.3.25. *Let t_1 be a legal trace. We write $\Xi_0 \vdash t_2 \preccurlyeq^{\bullet}_{\Delta} t_1$, if:*

1. $\Xi_0 \vdash {}_o\!\downarrow t_2 = {}_o\!\downarrow t_1$ *for all environment objects $o \in [o_1]$, where $[o_1]$ is the environment clique of the last action of t_1.*
2. $\Xi_0 \vdash t_2 \preccurlyeq_{\Delta} t_1$

If t_1 is empty, the first condition is omitted. We write $\Xi_0 \vdash C_1 \sqsubseteq^{nondet}_{trace} C_2$, if for all $\Xi_0 \vdash C_1 \stackrel{t_1}{\Longrightarrow}$, there exists a t_2 with $\Xi_0 \vdash C_2 \stackrel{t_2}{\Longrightarrow}$ with $\Xi_0 \vdash t_1 \preccurlyeq^{\bullet}_{\Delta} t_2$.

Lemma 3.3.26. *If $\Xi_0 \vdash C_1 \sqsubseteq_{obs} C_2$, then $\Xi_0 \vdash C_1 \sqsubseteq^{nondet}_{trace} C_2$.*

Definition 3.1.10 characterizes deterministic traces from the perspective of the component, captured in the assertion $\Xi_0 \vdash t : det_{\Theta}$, and dually from the perspective of the observer $\Xi_0 \vdash t : det_{\Delta}$. Traces at the component-observer interface and of a closed program (consequently legal traces) are deterministic in both respects, for which we use $det_{\Delta,\Theta}$. Note that requiring for $\Xi_0 \vdash C \stackrel{t}{\Longrightarrow}$, that the t is deterministic, i.e., impose $\Xi_0 \vdash t : det_{\Delta,\Theta}$ as restriction,[17] does *not* mean that $\Xi_0 \vdash C$ can perform only t (plus all its prefixes and up to renaming)! For each single environment, which closes C, there exists exactly *one* behavior (up to prefixing and renaming), but $\Xi_0 \vdash C \stackrel{t}{\Longrightarrow}$ describes the behavior of C as *open* program, with the environment program being abstracted away.

The following definition captures the intuition, that in the deterministic setting a *closed* program can do basically only one trace. I.e., in the setting of component and observer, in the parallel composition $\Xi_0 \vdash C_1 \parallel C_O$, the trace at the interface between c_1 and C_O is basically fixed. There is not literally exactly one trace possible, of course. Assume

$$\Xi_0 \vdash C \parallel C_O \underset{\bar{t}_1}{\stackrel{t_1}{\Longrightarrow}} \quad \text{and} \quad \Xi_0 \vdash C \parallel C_O \underset{\bar{t}_2}{\stackrel{t_2}{\Longrightarrow}} \tag{3.52}$$

The super- and subscript of the reduction relation $\Longrightarrow$ are meant to indicate that the parallel composition of C and C_O are evolving by internal reduction steps, where C is doing t_1 (resp., t_2) at the interface and C_O is doing the complementary trace $\bar{t}_1$, resp., $\bar{t}_2$. That a $\Longrightarrow$-reduction can be decomposed into two complementary interaction traces and conversely, that two complementary traces cancel out into a common internal reduction is the topic of Section A.4.3 and A.4.2. Given the two behaviors of equation (3.52), the fact that $C \parallel C_O$ is a deterministic, closed program obviously does not imply $t_1 = t_2$. The remaining slack allows *prefixing* and *renaming*, since the congruence rules of the semantics do not determine the actual names (or to say it differently, the components and the traces are anyway considered up to permutation of names, only). We introduce a separate notation for prefixing up to choice of ν-bound names in a trace.

[17] Indeed, $\Xi_0 \vdash t : det_{\Delta}$ as restriction suffices, since the component C can produce only traces which are deterministic from the commitment perspective.

Definition 3.3.27 (Prefixing and renaming)**.** *Let s and t be traces. Then $s \preccurlyeq_\alpha t$, if there is a renaming of t, i.e., some t' with $t' =_\alpha t$ such that $s \preccurlyeq t$.*

Lemma 3.3.28 (Individual determinism)**.** *Assume a legal trace $\Xi_0 \vdash t_1 : trace$. Assume further a set of traces $T = \{u' \mid u' \preccurlyeq_\alpha u \quad or \quad u' \succcurlyeq_\alpha u\}$ for some trace $\Xi_0 \vdash u : trace$. If for all prefixes $u_1 \preccurlyeq t_1$, there exists a trace $u_2 \in T$ such that the two conditions of Definition 3.3.25 relate u_1 and u_2, then $\Xi_0 \vdash t_2 \asymp_\Delta t_1$ for some trace t_2.*

We can now combine Lemma 3.3.26 with Lemma 3.3.28.

Theorem 3.3.29 (Completeness)**.** *If $\Xi_0 \vdash C_1 \sqsubseteq_{may} C_2$, then $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$.*

Part II

Concurrency

CHAPTER 4

Multithreading

In this chapter, we extend the results from Part I to include concurrency in the form of multithreading. The presentation follows the one for the sequential language. The additions are modest: Basically, the language is extended by the possibility to *create* threads. Hence, components now contain, besides classes and objects, a dynamic "set" of named threads.

Concerning the *connectivity*, complications arise in that now calls are possible, where the caller clique is not known. The general setting remains similar. Especially the open interface behavior rests again on an appropriate formulation and treatment of commitment and assumption contexts including heap abstractions.

4.1 Introduction

We extend now the development of Chapter 2 and 3 by concurrency in the form of *multithreading*. Syntactically and concerning the type system, the changes are modest, and also the main point of the development, the consideration of object connectivity, remains basically unchanged. Hence, we reuse much of the material from the sequential setting; in particular we do not explain or sometimes not even show rules and definitions that carry over. The resulting calculus is more or less a syntactic extension (by classes) of the *concurrent* object calculus from [62, 82].

4.2 Syntax

Concerning available types, a new one is introduced, the type $thread$ of threads; the grammar of types is shown in Table 4.1 (cf. also Table 2.1).

$$\begin{array}{lcl} T & ::= & B \mid none \mid thread \mid [l{:}U, \ldots, l{:}U] \mid [\![l{:}U, \ldots, l{:}U]\!] \mid n \\ U & ::= & T \times \ldots \times T \to T \end{array}$$

Table 4.1: Types

Besides named objects and classes, the dynamic configuration of a program can now contain a number of *named threads* $n\langle t\rangle$ as active entities, which, like objects, can be dynamically created. I.e., instead of one single thread named $\natural$, each thread $n\langle t\rangle$ carried now a unique name n. Unlike objects, threads are not instantiated by some statically named entity (a "thread class", as in *Java*), but directly created by providing the code (cf. also Section 6.1.5 in the conclusion for a discussion, and [9] [12] [11] for an investigation including thread-classes). Otherwise, the syntax is largely unchanged. In addition to the syntax of Table 2.2, we introduce

$$currentthread \quad \text{and} \quad new\langle t\rangle$$

as new expressions, referring to the name of the current thread and an expression which spawns a new thread with the code given by t.

For the names, we generally use n and its syntactic variants for threads (or just in general for names), o for objects, and c for classes. Otherwise, we use the syntactic abbreviations and conventions agreed upon in Section 2.2.

4.3 Type system

The type system requires some modest adaptation and extension (cf. Section 2.3, especially Tables 2.3 and 2.4). At the level of components, we need to account for the fact that threads are now named and appear in the interfaces Θ and Δ. The contexts now additionally store thread names, i.e., besides bindings of the form $o{:}c$ and $c{:}[\![l{:}U, \ldots, l{:}U]\!]$ for objects and classes, they contain bindings for thread names of the form $n{:}thread$.

C	$::=$	$\mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid n[\![O]\!] \mid n[n, F] \mid n\langle t \rangle$	program
O	$::=$	F, M	object
M	$::=$	$l = m, \ldots, l = m$	method suite
F	$::=$	$l = f, \ldots, l = f$	fields
m	$::=$	$\varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
f	$::=$	$\varsigma(n{:}T).\lambda().\bot_c \mid fv$	field
fv	$::=$	$\varsigma(n{:}T).\lambda().v$	defined field
t	$::=$	$v \mid \mathit{stop} \mid \mathit{let}\ x{:}T = e\ \mathit{in}\ t$	thread
e	$::=$	$t \mid \mathsf{if}\ v = v\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{if}\ \mathit{undef}(v.l)\ \mathsf{then}\ e\ \mathsf{else}\ e$	expression
	$\mid$	$v.l(v, \ldots, v) \mid v.l := fv \mid \mathit{currentthread}$	
	$\mid$	$\mathit{new}\, n \mid \mathit{new}\langle t \rangle$	
v	$::=$	$x \mid n$	values

Table 4.2: Abstract syntax

$$\frac{}{\Delta \vdash \mathbf{0} : ()}\ \text{T-EMPTY} \qquad \frac{\Delta \vdash C_1 : \Theta_1 \qquad \Delta, \Theta_1 \vdash C_2 : \Theta_2}{\Delta \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2}\ \text{T-PAR}$$

$$\frac{\Delta \vdash C : \Theta, n{:}\mathit{thread}}{\Delta \vdash \nu(n{:}\mathit{thread}).C : \Theta}\ \text{T-NU}_{it}$$

$$\frac{\Delta \vdash C : \Theta, o{:}c \quad \Theta \vdash c : [\![\ldots]\!]}{\Delta \vdash \nu(o{:}c).C : \Theta}\ \text{T-NU}_{io} \qquad \frac{\Delta, o{:}c \vdash C : \Theta \quad \Delta \vdash c : [\![\ldots]\!]}{\Delta \vdash \nu(o{:}c).C : \Theta}\ \text{T-NU}_{e}$$

$$\frac{;\, \Delta, c{:}T \vdash [\![O]\!] : T}{\Delta \vdash c[\![O]\!] : (c{:}T)}\ \text{T-NCLASS} \qquad \frac{;\, \Delta \vdash c : [\![T_F, T_M]\!] \quad ;\, \Delta, o{:}c \vdash [F] : [T_F]}{\Delta \vdash o[c, F] : (o{:}c)}\ \text{T-NOBJ}$$

$$\frac{;\, \Delta, n{:}\,\mathit{thread} \vdash t : \mathit{none}}{\Delta \vdash n\langle t \rangle : (n{:}\,\mathit{thread})}\ \text{T-NTHREAD} \qquad \frac{\Delta' \leq \Delta \quad \Theta \leq \Theta' \quad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \text{T-SUB}$$

Table 4.3: Static semantics (components)

In Table 4.4, we list only the rules additional to the ones from Table 2.4. The statement $new\langle t\rangle$ for thread creation possesses, not surprisingly, the type $thread$, and the same holds for the keyword $currentthread$. Note that rule T-NEWT requires t to be well-typed with type $none$, in accordance with rule T-NTHREAD for named threads on component level in Table 4.3.

$$\frac{\Gamma;\Delta \vdash t : none}{\Gamma;\Delta \vdash new\langle t\rangle : thread}\ \text{T-NEWT} \qquad \frac{}{\Gamma;\Delta \vdash currentthread : thread}\ \text{T-CURRT}$$

Table 4.4: Static semantics (2), extending Table 2.4

4.4 Operational semantics

The *operational* semantics is again split into a part dealing with internal steps and those with externals steps.

Table 4.5 for the internal steps extends Table 2.5 by rules for the new constructs. The expression $currentthread$ evaluates to the name of the thread which executes the expression (cf. rule CURRENTTHREAD). A new thread with a new local name is created by NEWT, which starts executing asynchronously, i.e., after the creation by NEWT, the spawning thread and the new one are running in parallel. As in the rule for object creation NEWO$_i$ in Table 2.5, the ν-binder hides the new name n_2 outside the two involved threads. Note that the type system, especially rule T-NEWT, assures that the type T mentioned in the reduction rules CURRENTTHREAD and NEWT equals $thread$.

All other rules remain as they are, with $\natural$ replaced by n. As before, the reduction relations are used modulo *structural congruence* $\equiv$, i.e., the rules from Table 2.6 and 2.7 still apply.

$$n\langle let\ x{:}T = currentthread\ in\ t\rangle \rightsquigarrow n\langle let\ x{:}T = n\ in\ t\rangle \qquad \text{CURRENTTHREAD}$$

$$n_1\langle let\ x{:}T = new\langle t\rangle\ in\ t_1\rangle \rightsquigarrow \nu(n_2{:}T).(n_1\langle let\ x{:}T = n_2\ in\ t_1\rangle \parallel n_2\langle t\rangle) \qquad \text{NEWT}$$

Table 4.5: Internal steps, extending Table 2.5

4.5 External behavior of a component

The external behavior of a component is given in terms of transitions, labeled by calls and returns, as before (cf. Table 4.6; cf. also the corresponding Table 2.8 in the sequential setting). In the binding part of the label, the scope of new objects and thread names may be extruded, or a name of an object may be transmitted to be lazily instantiated.

γ	::=	$n\langle call\ o.l(\vec{v})\rangle \mid n\langle return(v)\rangle \mid \nu(n{:}T).\gamma$	basic labels
a	::=	$\gamma? \mid \gamma!$	receive and send labels

Table 4.6: Labels

The only change in comparison with the labels of Table 2.8 is that the label carries the *name* of the concerned thread. As a consequence, the name context Φ in the binding part of the label may now contain also the name of the thread, in case the name *escapes* via the communication to the other side. Unchanged from the sequential case is the *augmentation*, i.e., we use $o_1\ blocks\ for\ o_2$ and $o_2\ returns\ v\ to\ o_1$ as additional expressions, as described in Section 2.6.4 with the typing as before (cf. Table 2.9). Also the external calls are augmented by the identity of the caller, i.e., the self of the method (cf. equation (2.5)). Note we use the *self*-parameter also to augment code fragments $new\langle t\rangle$ inside the method, if t itself contains calls to external objects.

If the initial thread starts in the component, as asserted by $\Theta_0 \vdash \odot$, the activity does not start "inside" a particular object. The initial thread uses $\odot$ as "self"-augmentation for external method calls, i.e., the calls are augmented to external calls $\odot\ x.l(\vec{x})$.

As introduced in the sequential setting, we use $\odot$ to represent the initial clique of the system (cf. page 37). In the multithreaded setting here, we need a corresponding symbol for each thread. We use $\odot_n$ for the initial clique of thread n. Note that $\odot$ is still used additionally as representative for the very first clique.

4.5.1 Connectivity contexts

Again, in the presence of cross-border instantiation, the semantics must contain a representation of the connectivity as an abstraction of the program's heap. In the single-threaded setting, the assumption and the commitment context Δ and Θ contained object and class names. Here, they additionally contain thread names, i.e., bindings of the form $n{:}thread$. We refine our convention from now[1] on as follows: By convention, we refer with Σ to the bindings for threads, whereas Δ, resp., Θ contain the bindings for object and class names, as before.

Leaving aside the thread names, the assumption and commitment contexts adhered to the following invariant during reduction: A class either resides in the component or in the environment, and correspondingly for the objects. A named thread, in contrast, in general occurs both in the environment and the component. This, of course, was already true in the single-threaded case with the thread named $\natural$, only that with the name fixed by convention, there was no need to incorporate it into the type system.[2]

[1] In the type system of Section 4.3, we did without designating the thread bindings in a special way.

[2] Note, however, that the rule for parallel composition T-PAR does *not* allow that a thread occurs on both sides of the $\|$-construct, since the domains of the commitment contexts on both sides of the $\|$-operator must be disjoint. Indeed, the parallel composition $n\langle t_2\rangle \parallel n\langle t_1\rangle$ does not make sense, and thus the type system forbids this.

The external semantics is formalized as labeled transitions between judgments

$$\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \ , \tag{4.1}$$

where $\Delta, \Sigma; E_\Delta$ are the *assumptions* about the environment of the component C and $\Theta, \Sigma; E_\Theta$ the *commitments*. The assumptions consist of a part Δ, Σ concerning the existence (plus static typing information) of *named entities* in the environment. The semantics maintains as invariant that the assumption and commitment contexts are disjoint concerning object and class names, whereas a thread name occurs as assumption iff it is mentioned in the commitments. By convention, the contexts Σ (and their alphabetic variants) contain exactly all bindings for thread names.

This means, as invariant we maintain for all judgments $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ that Δ, Σ, and Θ are pairwise disjoint. A further invariant is that a thread name n occurs in Σ, iff $\odot_n$ occurs in either Δ or Θ. This means, besides being relevant for connectivity information, $\odot_n$ contains also the information whether the thread started its life in the environment or in the component.

As mentioned, the $\odot_n$-symbol is needed in particular because new thread names may be communicated between environment and component (even if not in argument position).

The connectivity contexts are still of the form (cf. equation (2.6))

$$E_\Delta \subseteq \Delta \times (\Delta + \Theta) \tag{4.2}$$

for the assumption contexts, and dually $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$. Note that E_Θ and E_Δ are relations only between objects references; connectivity concerning class names or thread names does not play a role. We write $o_1 \hookrightarrow o_2$ for pairs from the relations E_Δ, resp. E_Θ. Note that E_Δ (resp. E_Θ) does not include pairs from $\Delta \times \Sigma$ (resp. $\Theta \times \Sigma$) for connectivity. The reason that we can do without considering acquaintance of objects with threads names is that objects cannot pass around thread names in as arguments of method calls.[3] If they could, as we allowed in [9] [12], $E_\Delta \subseteq \Delta \times (\Theta + \Sigma + \Delta)$.However, since Θ and Δ can contain the symbols $\odot$ and $\odot_n$, pairs of the form $o \hookrightarrow \odot$ or $\odot_n \hookrightarrow o$ are possible.

Given E_Δ (plus Δ, Σ, and Θ), we write $\leftrightharpoons$ for the reflexive, symmetric, and transitive closure of $\hookrightarrow$ on objects from Δ (cf. also equation (2.7), i.e.,

$$\leftrightharpoons \triangleq (\hookrightarrow\downarrow_{\Delta\times\Delta} \cup \hookleftarrow\downarrow_{\Delta\times\Delta})^* \subseteq \Delta \times \Delta \ . \tag{4.3}$$

As before, we write $\leftrightharpoons\!\!\hookrightarrow$ for the union $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Theta)$, where the semicolon denotes relational composition. As judgment, we use $\Delta, \Sigma; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta, \Sigma$, resp. $\Delta, \Sigma; E_\Delta \vdash o_1 \leftrightharpoons\!\!\hookrightarrow o_2 : \Theta, \Sigma$. For Θ, Σ, E_Θ, and Δ, Σ, the definitions are applied dually.

4.5.2 Check and update of contexts

The semantics is formulated as transitions between typed judgments (cf. also equation (2.9))

$$\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \xrightarrow{a} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta \ .$$

[3] Indirectly, objects can pass around the name of a thread to another object, of course, namely simply in that it calls a method of that object. The callee can find out the name of the thread via the expression $currentthread$.

We can reuse much of the corresponding definitions from the sequential case, with appropriate adaptations, and we start by changing the abbreviation for combined contexts.

Notation 4.5.1 (Contexts)**.** *We adapt the conventions from Notation 2.6.5 as follows. We abbreviate the triple of name contexts Δ, Σ, Θ as Φ, and the context $\Delta, \Sigma, \Theta, E_\Delta, E_\Theta$ combining assumptions and commitments Ξ. The notations Ξ_Δ and Ξ_Θ refer to the assumption and the commitment context. Furthermore we understand $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta}$ as $\acute{\Phi}$, and $\acute{\Xi}$ as consisting of $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta}, \acute{E}_\Delta, \acute{E}_\Theta$, etc.*

The compliance check of an incoming communication step wrt. the assumptions now reads as follows: (cf. Definition 2.6.7).

Definition 4.5.2 (Connectivity check)**.** *An incoming core label a with sender o_s is* well-connected *wrt. context $\acute{\Xi}$ (written $\acute{\Xi} \vdash o_s \xrightarrow{a} \_ : wc$) if:*

$$\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash o_s \leftrightharpoons\!\hookrightarrow fn(a) : \acute{\Theta}, \acute{\Sigma} \,. \tag{4.4}$$

Note that we assume that a is the *core* of a label; in the rules of the semantics and the rules for checking legality, the definition is invoked on the label with the binding part already "stripped off", and in the post-context which already contains the new information. Consequently in case of an incoming *all* label, $fn(a)$ includes the receiver o_r and the thread name. However, pairs of the form $o \hookrightarrow n$ where n is a thread name, are not part of the connectivity contexts.

Definition 4.5.3 (Name context update)**.** *The* update $\acute{\Phi}$ *of an assumption-commitment context Φ wrt. an incoming label $a = \nu(\Phi')\lfloor a \rfloor ?$ is defined as follows.*

1. $\acute{\Theta} = \Theta + \Theta'$.
2. $\acute{\Delta} = \Delta + \odot_{\Sigma'}, \Delta'$.
3. $\acute{\Sigma} = \Sigma + \Sigma'$.

We write $\Phi + a$ for the update. The update for outgoing communication is defined dually. Especially, $\odot_{\Sigma'}$ is added to Θ, instead of Δ. The notation $\odot_{\Sigma'}$ abbreviates $\odot_n$ if $\Sigma' \vdash n$, otherwise $\odot_n$ is not present.

Now to the update of *connectivity*, which we basically reuse from the single-threaded setting (Definition 2.6.9). Incoming communication —for outgoing communication, the situation is dual— may bring entities in connection which had been separate before. For the commitment context, this can be directly formulated by adding the fact that the receiver of the communication now is acquainted with all transmitted arguments (part (1) of Definition 4.5.4). For the update of *assumption* connectivity context E_Δ, we add that the sender knows all of the names which are transmitted boundedly (cf. part (2)). No update occurs wrt. names already known.

The semantics maintains as invariant that for each thread name n mentioned in the Σ-context, either $\Delta \vdash \odot_n$ or $\Theta \vdash \odot_n$: A thread n known both at the environment and the component started on exactly one side, marked by $\odot_n$.

Definition 4.5.4 (Connectivity context update)**.** *The update $(\acute{E}_\Delta, \acute{E}_\Theta)$ of a connectivity context (E_Δ, E_Θ) wrt. an incoming label $a = \nu(\Phi')\lfloor a \rfloor?$ with sender o_s and receiver o_r is defined as:*

1. $\acute{E}_\Theta = E_\Theta + o_r \hookrightarrow fn(\lfloor a \rfloor)$.
2. $\acute{E}_\Delta = E_\Delta + o_s \hookrightarrow dom(\Phi')$.

We write $(E_\Delta, E_\Theta) + o_s \xrightarrow{a} o_r$ for the update.

Combining Definitions 4.5.3 and 4.5.4, we write

$$\Xi + o_s \xrightarrow{a} o_r \ , \tag{4.5}$$

when updating names and connectivity at the same time (cf. equation (2.13)).

In addition to checking connectivity we must type-check the label.

Definition 4.5.5 (Well-formedness and well-typedness of a label)**.** *We use the definitions of* well-formedness *of a label ($\vdash a$), the* expected argument types *of a method call, asserted by*

$$\vdash a \quad \textit{and} \quad \acute{\Delta}, \acute{\Theta} \vdash o.l? : \vec{T} \to T \quad \textit{and} \quad \acute{\Delta}, \acute{\Theta} \vdash o.l! : \vec{T} \to T \tag{4.6}$$

as given Definition 2.6.11. Furthermore, well-typedness *of a core label is given by Table 4.7. The rules* LT-CALLO *and* LT-RETO *are analogous and not shown.*

$$\frac{\acute{\Sigma} \vdash n : thread \qquad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \qquad a = n\langle call\ o_r.l(\vec{v})\rangle?}{\acute{\Delta}, \acute{\Sigma}, \acute{\Theta} \vdash a : \ \vec{T} \to \_} \text{LT-CALLI}$$

$$\frac{\acute{\Sigma} \vdash n : thread \qquad ; \acute{\Delta}, \acute{\Theta} \vdash v : T \qquad a = n\langle return(v)\rangle?}{\acute{\Delta}, \acute{\Sigma}, \acute{\Theta} \vdash a : \ \_ \to T} \text{LT-RETI}$$

Table 4.7: Checking static assumptions

The definition is taken basically unchanged from the sequential setting (cf. Definition 2.6.11). The adaptations are caused by the fact that the communication labels now additionally carry the name of the thread, i.e., they are of the form $\nu(\Phi).n\langle call\ o_r.l(\vec{v})\rangle?$ and $\nu(\Phi).n\langle return(v)\rangle?$ instead of $\nu(\Phi).\langle call\ o_2.l(\vec{v})\rangle?$ and $\nu(\Phi).\langle return(v)\rangle?$. In particular, compared to Table 2.10 in the sequential setting, the rules of Table 4.7 contain an additional check that the name n is indeed a thread name. Again, the assertions in equation (4.6) and of Table 4.7 are formulated mentioning contexts $\acute{\Delta}$, $\acute{\Theta}$, and $\acute{\Sigma}$ instead of Δ, Θ, and Σ (which would work as well, of course). This is done as reminder that the check is *used* in the rules always for the post-context.

The order of the checks from Definition 4.5.5 (in the external steps of the semantics and the characterization of the legal traces) will be as follows. Given, e.g., an incoming call $\nu(\Phi').n\langle call\ o_r.l(\vec{v})?$, checking for well-formedness is first, i.e., that Φ' is a well-formed name context, and furthermore that only names actually occurring in the core $n\langle call\ o_r.l(\vec{v})\rangle?$ of the label are bound by

Φ'. Afterwards, using $\acute{\Delta}, \acute{\Theta} \vdash o_r.l? : \vec{T} \to T$ from equation (4.6), resp., equation (2.15), checks that o_r is the name of a component object, that it supports (via its class) a method labeled l. This check furthermore determines the types $\vec{T}$ *expected* for the arguments of the call and T for the value handed back when returning for the call. In the third step, rule LT-CALLI checks the actual parameters $\vec{v}$ against their expected type $\vec{T}$, and in addition, that n is the name of a thread. Note that rule LT-CALLI makes no use of the return type T. The return type is needed when checking return labels with rule LT-RETI or LT-RETO, of course.

4.5.3 External steps

The operational rules are quite similar to the ones from Section 2.6.4 for the sequential setting. The three rules CALLI_0–CALLI_2 cover three different situations wrt. incoming calls: A call of a thread new to the component, a reentrant call, and a call of a thread whose name is already known in the component. To deal with component entities that are being created during the call, $C(\Theta')$ stands for lazily instantiated objects mentioned in Θ'.

Rules CALLI_1 and CALLI_2 work analogously to the single-threaded case. In CALL_2, the sender of the call is now $\odot_n$, the initial clique of thread n, whereas in the single-threaded setting, the sender was $\odot$, the initial clique at the very start of the program, as the rule could be applied to the only thread $\natural$ there. In the simpler setting, we did not introduce a $\odot_\natural$, but used $\odot$ to represent the initial clique of $\natural$. In the multithreaded setting now, the first thread, say n, that crosses the border is represented both by $\odot$ and by $\odot_n$ Both are put into the same clique, however, by the initial step, either by rule CALLI_0 or by CALLO_0, depending on whether $\Delta_0 \vdash \odot$ or $\Theta_0 \vdash \odot$. This can be seen as follows. Assuming for one case $\Delta_0 \vdash \odot$, then $\odot$ is the *only* choice for the source of the call in the premise $\Delta \vdash o$ of L-CALLI_0.[4] Therefore, after the call, the equation $\odot \hookrightarrow \odot_n$ is part of the (assumption) connectivity context, when n is the name of the thread in question.

Rule CALLI_0 deals with the situation, that the thread n enters the component for the first time, assured by the premise $\Phi' \vdash n$.[5] With the thread being new, we have no indication from which clique the call originates. However, the new thread must have been created at some point before by *some* environment clique. Indeed, *any* existing environment clique is a candidate for having created n. So the update to $\acute{\Xi}$ *non-deterministically guesses* to which environment clique the thread's origin $\odot_n$ belongs to, namely in the premise $\Delta \vdash o$. The guess is remembered by adding $o \hookrightarrow \odot_n$ to the connectivity context. Note that $\Sigma' \vdash n$ implies $\acute{\Delta} \vdash \odot_n$ after the call (cf. Definition 4.5.3(3)).

The return steps are simpler than the calls, as the element of guessing is not present: When a thread returns, the callee as well as the thread are already known. Returns are simpler than calls also in that only one value is communicated, not a tuple (and we do not have compound types). To avoid case distinctions and to stress the analogy with the treatment of the calls, we denote

[4] In the sequential setting of Table 2.11, the corresponding premise of CALLI_0 explicitly required $\Delta_0 \vdash \odot$, not simply $\Delta_0 \vdash o$. Here, we use $\Delta \vdash o$, since CALLI_0 not only applies for the initial step, but also later when a new thread crosses the interface.

[5] That n is indeed a name of a thread, i.e., that $\Phi' \vdash n : \mathit{thread}$, is assured by the type checking premises, in particular by rule LT-CALLI of Table 4.7.

$$\frac{a = \nu(\Phi').\, n\langle call\ o_r.l(\vec{v})\rangle? \qquad \Delta \vdash o \qquad \Phi' \vdash n \qquad \acute{\Xi} \vdash o_r.l? : \vec{T} \to T \quad \acute{\Xi} = \Xi + (o \hookrightarrow \odot_n) + \odot_n \xrightarrow{a} o_r \quad \acute{\Xi} \vdash \odot_n \overset{\lfloor a \rfloor}{\rightarrow} o_r : \vec{T} \to \_}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Theta') \parallel n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ x\ to\ \odot_n; stop\rangle}\ \textsc{CalliI}_0$$

$$\frac{a = \nu(\Phi').\, n\langle call\ o_r.l(\vec{v})\rangle? \quad t_{blocked} = let\ x'{:}T' = o\ blocks\ for\ o_s\ in\ t \quad \acute{\Xi} \vdash o_r.l? : \vec{T} \to T \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrow} o_r : \vec{T} \to \_}{\Xi \vdash \nu(\Phi).(C \parallel n\langle t_{blocked}\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Phi).(C \parallel C(\Theta') \parallel n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ x\ to\ o_s; t_{blocked}\rangle)}\ \textsc{CalliI}_1$$

$$\frac{a = \nu(\Phi').\, n\langle call\ o_r.l(\vec{v})\rangle? \quad \Delta \vdash \odot_n \quad \acute{\Xi} \vdash o_r.l? : \vec{T} \to T \quad \acute{\Xi} = \Xi + \odot_n \xrightarrow{a} o_r \quad \acute{\Xi} \vdash \odot_n \overset{\lfloor a \rfloor}{\rightarrow} o_r : \vec{T} \to \_}{\Xi \vdash C \parallel n\langle stop\rangle \xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Theta') \parallel n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ x\ to\ \odot_n; stop\rangle}\ \textsc{CalliI}_2$$

$$\frac{a = \nu(\Phi').\, n\langle call\ o_r.l(\vec{v})\rangle! \quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi \quad \acute{\Phi} = \Phi \setminus \Phi' \quad \acute{\Delta} \vdash o_r \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r}{\Xi \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_s\ o_r.l(\vec{v})\ in\ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Phi}).(C \parallel n\langle let\ x{:}T = o_s\ blocks\ for\ o_r\ in\ t\rangle)}\ \textsc{CalloO}$$

$$\frac{a = \nu(\Phi').\, n\langle return(v)\rangle? \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrow} o_r : \_ \to T}{\Xi \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_r\ blocks\ for\ o_s\ in\ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Phi).(C \parallel n\langle t[v/x]\rangle)}\ \textsc{RetI}$$

$$\frac{a = \nu(\Phi').\, n\langle return(v)\rangle! \quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi \quad \acute{\Phi} = \Phi \setminus \Phi' \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r}{\Xi \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_s\ returns\ v\ to\ o_r\ in\ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Phi}).(C \parallel n\langle t\rangle)}\ \textsc{RetO}$$

$$\frac{\Delta \vdash c}{\Xi \vdash \nu(\Phi').(C' \parallel n\langle let\ x{:}c = new\ c\ in\ t\rangle) \rightsquigarrow \Xi \vdash \nu(\Phi', o{:}c).(C' \parallel n\langle let\ x{:}c = o\ in\ t\rangle)}\ \textsc{NewO}_{lazy}$$

Table 4.8: External steps

the binding part of the label by $\nu(\Phi')$, resp., $\nu(\Delta', \Sigma', \Theta')$, as before, even if Σ' and at least one of the name contexts Δ' and Θ' are guaranteed to be empty. Rule $\textsc{NewO}_{lazy}$, as before, deals with lazy instantiation and describes the local instantiation of an external class.

As initial step, only calls are possible (by rule $\textsc{CalliI}_0$ or $\textsc{CalloO}_0$). As in the single-threaded setting, there is exactly one initial thread, either in the component or in the environment. Where the initial activity starts is marked by $\odot$, which makes it the only possible guess for o in $\textsc{CalliI}_0$. For the initial static contexts, we are given either $\Delta_0 \vdash \odot$ or $\Theta_0 \vdash \odot$.

CHAPTER 5

Full abstraction

Next we address full abstraction in presence of multithreading. Section 5.1 defines the notion of traces and the notion of observation as contextual equivalence, resp., contextual preorder; Section 5.2 deals with soundness and completeness. For completeness, we characterize the set of possible ("legal") traces in Section 5.2.1, state closure conditions on the set of traces in Section 5.2.2, and show how to realize a legal trace up to the unavoidable uncertainty (captured in the closure conditions) in Section 5.2.3, yielding completeness.

5.1 Trace semantics and ordering on traces

The trace semantics resembles the one from Section 3.1. Again, a trace of a well-typed component is a sequence of external steps where the corresponding rules from Table 3.1 can be reused. The only change is that the labels in the traces now carry additionally the name of the corresponding thread. Using the conventions from Notation 4.5.1, we write $\Xi_1 \vdash C_1 \stackrel{t}{\Longrightarrow} \Xi_2 \vdash C_2$ for C_1 exhibiting the external trace t in the assumption and commitment context Ξ_1. Further material which we reuse is the definition of future projection (Definition 3.1.3) and Definition 3.1.1 for connectivity after executing a trace:

$$\Xi \vdash_\Theta t \rhd o_1 \leftrightharpoons o_2, \tag{5.1}$$

resp. $\Xi \vdash_\Theta t \rhd o_1 \leftrightharpoons\!\!\hookrightarrow o_2$ (and dually for $\vdash_\Delta$).

Since the caller of a method is *anonymous,* the equivalences on the traces need a refinement. In the single-threaded setting, anonymity of the caller did not cause concern: The sender of a, say, incoming call can be determined by the history of the thread, determined at least up to the originating clique (cf. Definition 3.3.4). Also in case that the sole thread enters initially, the sender is determined as the representative $\odot$ of the initial clique.

Now, a *new* thread may enter the component via a method call with the caller unknown. The semantics deals with this circumstance in that the corresponding CALLI-rule non-deterministically *guesses* the originating clique, consistent with the current connectivity contexts, and where the step updates the contexts according to the guess.

For the closure conditions, especially replay, the problem lies in the fact that a given component trace does not contain enough information to determine the senders of those labels. What is worse, with the trace at hand, the external semantics from Section 4.5 is based on *one* choice of the identities of unknown senders, while in fact there might be more than one possible interpretation consistent with the trace. To see why this is problematic, consider the following example.

Example 5.1.1 (Replay)**.** *Consider Figure 5.1: Does a component exhibiting the behavior of Figure 5.1(a) unavoidably show also the one of Figure 5.1(b). I.e., is the last c' is justified by being a replay of earlier behavior? Let s_1 be the interaction of the trace with the clique of o_1 (without c), and analogously s_2 the interaction with the clique of o_2 (again without c). After s_1 and s_2, the component consists of two cliques, represented by the objects o_1 and o_2. After $s_1 s_2$, the component answers with the outgoing call c, where we assume that the call is issued by a thread new to Δ. This means that the origin of the call, either the clique of o_1 or of o_2, might be undetermined.*[1]

The interaction continues with s_1' which we assume to create a duplicate of the clique of o_1, and let us assume that the next outgoing call c' can originate only from o_1'. Now the question is, whether at the end of scenario 5.1(a), c' is unavoidable*, i.e., whether*

$$s_1\ c\ s_2\ s_1'\ \asymp_\Theta\ s_1\ c\ s_2\ s_1'\ c'\ \ ? \tag{5.2}$$

The answer is no. The sender of c' is (as we assume) the clique of o_1'. Since the previous outgoing call c might have originated not in the clique of o_1 but in o_2, the c' is not

[1] Sometimes, of course, the situation is such that further information carried with the call-label solves the uncertainty wrt. the origin of the call.

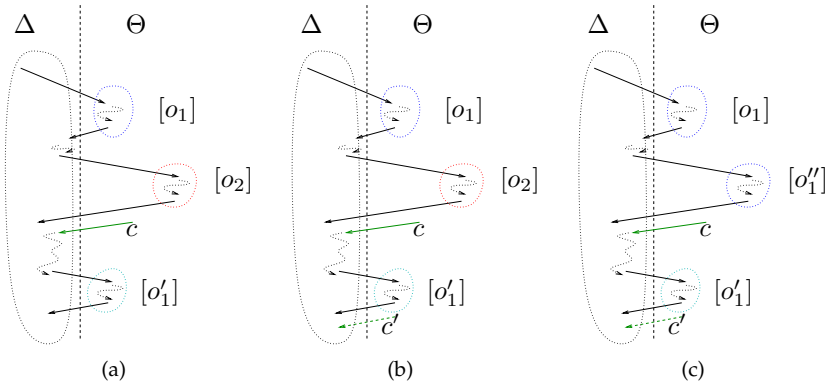


Figure 5.1: Replay

unavoidable, as in that situation, it is not a replay. It would be unavoidable only if for all *possible senders of c, $s'_1\ c'$ has been seen before.*

If the second clique o_2 of Figure 5.1(a) would itself be a replay of o_1, as indicated by o''_1 of Figure 5.1(c), then the second c' would be justified by replay.

A more formal justification for the above "no" goes as follows. Let us add information to the trace to disambiguate origin of a label. This is necessary for calls of new threads, only. We do this by adding the sender of the call to the call label and call such a label augmented. *Analogously, we call a trace with this additional information* augmented *(cf. Definition 5.1.4 below) and indicate an augmented trace as t^+, where t is the underlying unaugmented trace.*

In the above situation of equation (5.2), there are two possible augmentations for the trace t on the left, let us call them t_1^+ and t_2^+:

$$s_1\ c_1^+\ s_2\ s'_1 \quad \textit{and} \quad s_1\ c_2^+\ s_2\ s'_1\ . \tag{5.3}$$

The two possible situations are shown in 5.2(a) and 5.2(b), where the additional, fat arrow indicates the source clique of the call c, i.e., the caller. Remember that the call c is done by a new thread and that spawning *a new thread works* asynchronously. *For instance, in scenario 5.2(a), the new thread is created $[o_1]$ but remains invisible from the outside until after the interaction with the second clique $[o_2]$ has been executed.*

As assumed, the second call c' can come only *from o'_1, i.e., there are still only two augmentations for the longer trace $t_2 = t_1\ c'$, corresponding to the two augmentations of equation (5.3); with the origin of c' fixed by assumption, the longer trace does not introduce a further degree of freedom. For t_1^+, the longer $t_1^+\ c'$* is *a replay, for t_2^+, it is not. To sum up: $t\ c'$ a replay given t, if* for all *augmentations, $(t\ c')^+$ is a replay for t^+.* □

Example 5.1.2 (Replay (2))**.** *Let us extend the previous Example 5.1.1 such that the call c' in question has two possible source cliques (cf. Figure 5.3), and again we ask, whether it is unavoidable that, given the scenario s of Figure 5.3(a), the component shows also the behavior sc' of Figure 5.3(b) with the additional, trailing c'. Unlike the situation of Figure 5.1(a) and 5.1(b), now $s \asymp_\Theta sc'$. As before, the origin of c is undetermined —both the cliques of o_1 or of o_2 are candidates— but no matter which clique is the source of c, the second c' is unavoidable, since depending on the situation, it can extend s by extending o'_1 or o'_2.* □

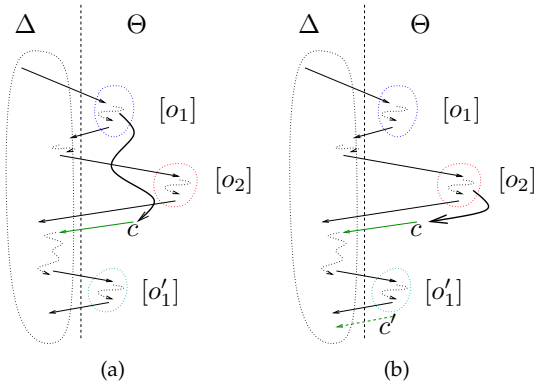


Figure 5.2: Replay

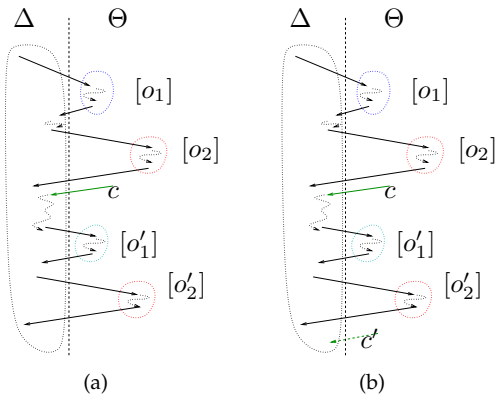


Figure 5.3: Replay

Example 5.1.3 (Anonymous caller)**.** *The next example illustrates the issue from the dual perspective of the observer. Consider the scenario 5.4(a). The observer on*

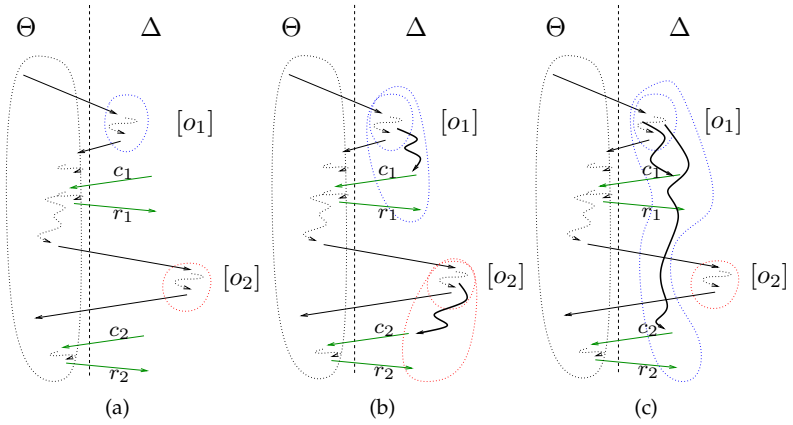


Figure 5.4: Anonymous caller

the right-hand side consists of two cliques, represented by o_1 and o_2, created by the program on the left. Additionally, the observer creates two new threads which interact with the component by a call and a return. Concerning the first call-return interaction c_1 and r_1, the originating clique is not in doubt: It's the only one present at that point, the one of o_1. This is guaranteed in the rule CALLI$_0$ *for incoming calls via a new thread by requiring that there exists an environment object $\Delta \vdash o$ acquainted with the arguments of the label.*[2]

The situation is different for the second call-return pair c_2 and r_2. Assuming that the connectivity for the arguments of the call does not disambiguate the origin of the incoming call, both cliques of o_1 and o_2 may have spawned the second new thread (cf. the scenarios 5.4(b) and 5.4(c)). In particular, 5.4(c) is possible, since a new thread may not immediately be visible at the interface and may have been created internally before the very first thread has left the clique o_1.

Now consider the component C_1 from the left-hand side of scenario 5.4(a) and let's denote its interface behavior as

$$t_1\ c_1\ r_1\ t_2\ c_2\ r_2\ . \tag{5.4}$$

Furthermore, assume a second component C_2 with the behavior

$$t_1\ c_1\ r_1\ +\ t_2\ c_2\ r_2\ , \tag{5.5}$$

i.e., it non-deterministically chooses the left-hand branch or the right-hand branch. The "+" can be understood as non-deterministic choice between the two traces. Alternatively one can think of the behavior described by (5.5) consisting of the two traces $t_1\ c_1\ r_1$ and $t_2\ c_2\ r_2$ (plus their prefixes).

Now, can an observer distinguish C_1 from C_2? The answer is yes; in particular, the observer on the right-hand side of scenario 5.4(c) can insist *on observing $t_1\ c_1\ r_1\ c_2\ r_2$*

[2] In more detail, the premise of the rule, requires that $\odot_n$, the "virtual" initial object/clique of the new thread is acquainted with the objects from the label after adding $o \hookrightarrow \odot_n$ to E_Δ (cf. part 2 of Definition 4.5.4).

before reporting success. Also another component C_2' with the behavior

$$t_1\ c_1\ r_1\ c_2\ r_2\ t_2\ , \tag{5.6}$$

i.e., where in comparison with scenario 5.4(a) and equation (5.4), t_2 and $c_2\ r_2$ are swapped, can be distinguished from C_1, namely by the observer from 5.4(b), where the clique of o_1 can block *progress after $t_1\ c_1\ r_1$. However, if C_1 may be successful, then also C_2'' doing*

$$t_1\ c_1\ r_1\ c_2\ r_2\ +\ t_2\ c_2\ r_2 \tag{5.7}$$

may be successful. No matter whether the observer is programmed to spawn the second thread in the clique o_1 or of o_2, it cannot hinder success.

Given a trace t, let t^+ represent the trace augmented with additional information about the callers' identities. Then the reason why success of C_1 implies success of C_2'' is that no matter whether the original trace of C_1 of scenario 5.4(a) is interpreted as t_b^+ or as t_c^+ as in the second and third scenario, there exists one branch of behavior from equation (5.7), which leads to success. □

As mentioned and discussed in the above examples, the origin of a communication in case of new threads is guessed but not remembered in the trace. To repair this lack of information, we *augment* the labels such that for each call by a new thread, the sender clique is kept in the trace. This augmentation is needed only when a *new* thread enters the component (via a method call). For other method calls and for returns the sender can be determined by consulting the history, as done in the single-threaded case. In case of a first interface interaction of a thread, as formalized in L-CALLI$_0$, the sender of the call is calculated by the premise $\Xi \vdash r \rhd \odot_n \xrightarrow{a} o_r : \vec{T} \to \_$ as $\odot_n$, the representative of the initial clique of thread n.

Definition 5.1.4 (Augmentation)**.** *An* augmented *trace of component C in context Ξ_0 is given by the rules of Table 4.8 where incoming call labels justified by rule* CALLI$_0$ *are kept in the trace as $\nu(\Phi', n{:}thread).n\langle[o]call\ o_r.l(\vec{v})\rangle?$, where o is the sender guessed in the premise $\Delta \vdash o$ of that rule.*

For outgoing *calls via* CALLO*, where the scope of the thread n escapes the component, i.e., for steps labeled $\nu(\Phi', n{:}thread).n\langle call\ o_r.l(\vec{v})\rangle!$, the augmentation works as follows:* Some *object o with $\Theta \vdash o$ and $\Xi \vdash o_s \leftrightharpoons o$ is added, where o_s is the object mentioned as sender in the augmented code in* CALLO*. This yields as augmented label $\nu(\Phi', n{:}thread).n\langle[o]call\ o_r.l(\vec{v})\rangle!$.*

We write $\Xi_0 \vdash C \stackrel{t^+}{\Longrightarrow}$ for C performing an augmented trace, where we understand t as the underlying unaugmented, original trace. Given $\Xi_0 \vdash C \stackrel{t}{\Longrightarrow}$, and in abuse of notation, we mean by t^+ also the set *of augmented traces of t, i.e., the set of all augmentations t^+ of t with $\Xi_0 \vdash C \stackrel{t^+}{\Longrightarrow}$.*

Remark 5.1.5 (Augmentation and justification pointers)**.** *The augmentation here is reminiscent to the use of* justification pointers *in arena games also known as HO-games (Hyland and Ong) [77]. What is called traces here, is often dubbed* paths *in game theory, i.e., sequences of moves (= labels). The moves of a game come equipped with an* enabling *relation, expressing potential causality: $m \vdash n$ reads "move m enables move n", where in standard situations, $m \vdash n$ implies that m is a player move and n one of the opponent, or vice versa; non-standard are initial situations for moves without having an different move to enable them, where $m \vdash m$, and which are called*

self-enabling. The considered plays (= traces) are not just arbitrary sequences of move, but the must adhere to a few restrictions. Apart from alternation*, one general condition is that the enabling relation $\vdash$ is respected in the following sense: Each* occurrence *of a move in the play is* justified *by a uniquely determined move occurring earlier in the play which enables it (with the exception of self-enabling moves, which can occur "spontaneously", without justification). This additional information pointers —paths with this additional pointer structure are called* justified*— resemble the augmentation with the caller identity we use in our traces. Cf. e.g., [76] for some introduction to game semantics.* □

With the augmentation, we can define the pre-order as follows (cf. also Definition 3.1.11 for the corresponding definition in the deterministic setting).

Definition 5.1.6 ($\sqsubseteq_{trace}$)**.** *$\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$, if the following holds. If $\Xi_0 \vdash C_1 \stackrel{t_1^+}{\Longrightarrow}$, then for all environment cliques $[o_1]$ after t_1, $\Xi_0 \vdash C_2 \stackrel{t_2^+}{\Longrightarrow}$ for some t_2^+, s.t.,*

1. *$\Xi_0 \vdash {}_o\!\downarrow t_2^+ = {}_o\!\downarrow t_1^+$, for all environment objects $o \in [o_1]$, and*
2. *$\Xi_0 \vdash t_2^+ \preccurlyeq_\Delta t_1^+ : trace$.*

The relationship between the definition of $\sqsubseteq_{trace}$ here and in the sequential setting (cf. Definition 3.1.11) is as follows. With only one thread in the sequential case, the augmented t^+ coincides with t, so Definition 5.1.6 degenerates to the old definition wrt. augmentation when applied to the single-threaded case. Secondly, expanding the condition $s \asymp_\Delta t$ of sequential setting (and after choosing an appropriate renaming of, e.g., t, gives condition 1 for the success-reporting clique of the above definition. In other words, part 1 corresponds to $\asymp_\Delta$ for *one clique* and with the clique chosen, the names in s and t can be renamed such that actual "tree equality" holds, as expressed in part 1, using projections. Part 1 indeed holds (apart from augmentation) in the deterministic setting as a consequence of the equality $\asymp_\Delta$. The difference is that here this form of equality hold only for *one* clique, whereas for all others, only the weaker "prefix" $\preccurlyeq_\Delta$ is required. Instructive is also the comparison with Definition 3.3.25 of $\sqsubseteq_{trace}^{nondet}$, which we introduced as auxiliary, weaker definition of $\sqsubseteq_{trace}$ in the sequential setting. Indeed, $\sqsubseteq_{trace}^{nondet}$ is closer to the definition of $\sqsubseteq_{trace}$ from Definition 5.1.6 of above, since, unlike the variant of Definition 3.1.11, it does not exploit the fact that in the single-threaded setting, programs behave deterministically.

As notion of observation, we use may testing preorder, i.e., basically the same definition as in the sequential setting (cf. Section 2.5 and especially equation (2.3) for the notion of barbing).

Definition 5.1.7 (May testing)**.** *Assume $\Xi_0 \vdash C_1$ and $\Xi_0 \vdash C_2$. Then $\Xi_0 \vdash C_1 \sqsubseteq_{may} C_2$, if*

$$(C_1 \parallel C) \Downarrow_{c_b} \quad \textit{implies} \quad (C_2 \parallel C) \Downarrow_{c_b} \tag{5.8}$$

for all $\bar{\Xi}_0, c_b{:}\mathsf{barb} \vdash C$, where $\bar{\Xi}_0$ corresponds to Ξ_0 with the roles of assumption and commitment contexts exchanged.

5.2 Soundness and completeness

The situation for soundness is not much more complicated than in the sequential setting. As before, $\bar{t}$ denotes the trace complementary to t.

Proposition 5.2.1 (Soundness)**.** $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$ *implies* $\Xi_0 \vdash C_1 \sqsubseteq_{may} C_2$.

5.2.1 Legal traces

As in the sequential case, we characterize the interface behavior in the form of possible traces. Half of the work has been done already by the careful design of the open semantics of Section 4.5, where the absent environment is represented abstractly by the assumption contexts. For characterizing the legal traces, we analogously abstract away from the program code, which makes the system completely symmetric.

The formalization works quite similar to the one from Section 3.3.2. The restrictions on the set of traces are again grouped into well-typedness, well-connectedness, and enabledness. One restriction missing now is, obviously, the requirement of determinism.

Enabledness, i.e., whether after a given history, an input or and output is possible and whether the next interaction can be a call and/or a return is given by Definition 3.3.3. The corresponding judgment is written as ("label a is enabled after history r"):

$$\Xi \vdash r \rhd a \,.$$

Furthermore important is the determination of sender and receiver from a given history. Based on the characterization of balance and with the help of the pop-function (cf. Definition 3.3.1 and Table 3.3), the functions $sender$ and $receiver$ are given in Definition 3.3.4 in the sequential setting for a trace of a single thread. Obviously, the notion of balance and, based on that, the definitions of sender, receiver, and enabledness, make only sense *per thread*. Therefore, the mentioned definitions are used here on the *projection* of the multi-threaded trace onto the thread of interest. I.e., for checking enabledness of $\Xi \vdash r \rhd a$, we use the single-threaded definition to check $\Xi \vdash r \downarrow_n \rhd a$, where n is the thread executing label a and where the projection $r \downarrow_n$ of r to thread n consists of the sequence of labels from r, with all labels *not* executed by n omitted. If $\Xi \vdash r \rhd \gamma?$ and n is the thread of label $\gamma?$, we say, thread n is input-enabled after r. Analogously for input-call enabledness, input-return enabledness, etc.

The legal traces are specified by a system for judgments

$$\Xi \vdash r \rhd s : trace \,, \tag{5.9}$$

where Ξ consists of an assumption context $\Delta, \Sigma; E_\Delta$ and a commitment context $\Theta, \Sigma; E_\Theta$. The judgment asserts that under the assumptions and commitments Ξ and after r, the trace s is legal. In the judgment, r represents the history of the trace, consulted to assure (amongst other things) that calls and returns appear in a balanced manner per thread. The rules for legal traces are shown in Table 5.1.

The legal trace system, as the external operational semantics, works nondeterministically in guessing the sender, when unknown, i.e., in the case of

$$\frac{}{\Xi \vdash r \rhd \epsilon : trace}\ \text{L-EMPTY}$$

$$\frac{\Xi \vdash r \rhd o_s \xrightarrow{a} o_r : \vec{T} \to \_ \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : \vec{T} \to \_ \quad \Phi \vdash n \quad a = \nu(\Phi').\, n\langle call\ o_r.l(\vec{v})\rangle? \quad \acute{\Xi} \vdash r\ a \rhd s : trace}{\Xi \vdash r \rhd a\ s : trace}\ \text{L-CALLI}_{1,2}$$

$$\frac{\Xi \vdash r \rhd \odot_n \xrightarrow{a} o_r : \vec{T} \to \_ \quad \Delta \vdash o \quad \acute{\Xi} = \Xi + (o \hookrightarrow o_s) + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : \vec{T} \to \_ \quad \Phi' \vdash n \quad a = \nu(\Phi').\, n\langle call\ o_r.l(\vec{v})\rangle? \quad \acute{\Xi} \vdash r\ a \rhd s : trace}{\Xi \vdash r \rhd a\ s : trace}\ \text{L-CALLI}_0$$

$$\frac{\Xi \vdash r \rhd o_s \xrightarrow{a} o_r : \_ \to T \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : \_ \to T \quad a = \nu(\Phi').\, n\langle return(v)\rangle? \quad \acute{\Xi} \vdash r\ a \rhd s : trace}{\Xi \vdash r \rhd a\ s : trace}\ \text{L-RETI}$$

Table 5.1: Legal traces (dual rules omitted)

L-CALLI$_0$. When using the rules on *augmented* traces, the system becomes deterministic.

The rules resemble the ones in the sequential case, checking whether after r, action a is possible, i.e., whether it is well-formed, well-typed, and adheres to the restrictions imposed by the connectivity contexts. Furthermore, the contexts are updated appropriately, and the rules recur checking the tail of the trace. The rules are symmetric wrt. incoming and outgoing communication.

Apart from the fact that now the thread name is part of the label, the main difference concerns the identity of the communication partners. In the sequential case, the sender of each communication is determined. Now that new threads can be created, it is possible, that the sender of a call is undetermined. Hence the check for legality makes a non-deterministic *guess* among the possible senders. In the semantics, this concerned CALLI$_0$ of Table 4.8, which deals with exactly this situation: A call enters the component by a *new* thread. The treatment in L-CALLI$_0$ here is analogous: the guessed sender o, which must be contained in the environment ($\Delta \vdash o$), is remembered by added $o \hookrightarrow o_s$ to the assumptions, where o_s is the sender calculated from the history r. The rule L-CALLI$_{1,2}$ for legal traces here combines CALL$_1$ and CALL$_2$ of the semantics where the sender is determined as the thread is already known and where consequently the sender can be determined consulting the history.

A further difference between the single-threaded setting and the rules now is that rule L-CALLI$_0$ (or dually L-CALLO$_0$) does not only cover the initial state, but deals with all situations where a new thread crosses the interface. Hence, unlike L-CALLI$_0$ from Table 3.5, the rule here must allow a non-empty history r left of the $\rhd$-symbol.

Remark 5.2.2. *Note that for rule* L-CALLI$_0$*, the sender is determined as* $\odot_n$*, which is not a "real" object but a place holder and hence has no type. However, being a call label, the sender argument is not needed for the type check of the core of the label in the premise* $\acute{\Xi} \vdash \odot_n \xrightarrow{\lfloor a \rfloor} o_r : \vec{T} \to \_$*, (cf. Table 4.7).* □

$$\Xi \vdash s\ \nu(\Phi').\gamma_2?\ \gamma_1?\ r \sqsubseteq_\Theta s\ \nu(\Phi').\gamma_1?\ \gamma_2?\ r : \mathit{trace} \qquad \text{O-II}$$

$$\Xi \vdash s\ \nu(\Phi').\gamma_2!\ \gamma_1!\ r \sqsubseteq_\Theta s\ \nu(\Phi').\gamma_1!\ \gamma_2!\ r : \mathit{trace} \qquad \text{O-OO}$$

$$\Xi \vdash s\gamma_2?\gamma_1!r \sqsubseteq_\Theta s\gamma_1!\gamma_2?r : \mathit{trace} \qquad \text{O-OI}$$

$$\frac{\Xi \vdash s \preccurlyeq_\Theta t : \mathit{trace}}{\Xi \vdash s \sqsubseteq_\Theta t : \mathit{trace}}\ \text{O-SwapReplay}_\Theta$$

$$\Xi \vdash s\gamma? \sqsubseteq_\Theta s : \mathit{trace} \qquad \text{O-Input}$$

Table 5.2: Closure preorder (on augmented traces)

5.2.2 Closure

Next we spell out the closure conditions for sets of traces, i.e., characterize the uncertainty of observation or, dually, the uncertainty up-to which a component can be programmed.

A few ingredients have already been mentioned at the beginning of Section 5.1, namely the tree-like structure of the traces, replay, and prefixing. Those are already present in the sequential setting. *Concurrency* adds one more aspect of observational uncertainty, namely the inability to *atomically observe* interaction, in particular the order of certain communication steps. Furthermore, since objects are *input enabled,* each trace can be extended by a further incoming call; the latter is of course not a consequence of concurrency. Note that for augmented traces, the post-assertions $\acute{\Xi}$ after a trace t are determined by Ξ and t (up to renaming, of course).

Definition 5.2.3 (Closure preorder)**.** *The* closure *on traces is defined as:*

$$\Xi \vdash s \sqsubseteq_\Theta t : \mathit{trace}$$

iff for all t^+, there exists an s^+ such that (in abuse of notation) $\Xi \vdash s^+ \sqsubseteq_\Theta t^+ : \mathit{trace}$, where $\sqsubseteq_\Theta$ is the reflexive and transitive closure generated by the rules from Table 5.2. The traces left and right of $\sqsubseteq_\Theta$ are tacitly assumed to be legal.

O-SwapReplay$_\Theta$ imports the tree-like clique structure and replay into the closure conditions (cf. Definition 3.1.8). Rule O-Input expresses *input enabledness.* Note that a component is enabled not only wrt. incoming calls but also wrt. incoming returns.

The remaining three rules allow to exchange the order of two neighboring steps in certain situations. We call the slack introduced in Table 5.2 in addition to $\preccurlyeq_\Theta$ and input enabledness and caused by the non-atomicity of interaction as *switching* to distinguish it from *swapping* by which we mean reordering due to separate observer cliques. We use $\sqsubseteq_\Theta^{switch}$ to denote the order relation generated by the 3 additional rules.

Switching has nothing to do with connectivity and concerns the behavior of even a single object. Indeed, the switching rules (as well as the one for input-enabledness) are already present in an object-based setting, for instance

in [82]. Note further that the (implicit) proviso that $\sqsubseteq$ is considered for legal traces, only, implies that two labels which can be switched concern two *different* threads.

Two incoming communications may occur in any order (cf. rule O-II). The reason is that an incoming call step, resp., an incoming return has no immediate, atomic side effects. Hence the order in which the incoming communication steps are traced at the interface says nothing about the order in which they lead to observable effects in the state of the concerned objects.

The reason why two outgoing steps (cf. rule O-OO) can be exchanged, is a bit different: After an outgoing step has occurred, the responsible thread is *blocked* and can therefore not influence the second step. By the same reason, an outgoing communication before an incoming communication step can be postponed (cf. rule O-OI). Note that an inverse rule to O-OI is not correct: If an incoming communication occurs *before* an outgoing one, the second one may be causally dependent on the first; hence it is not guaranteed that the trace can occur also in the switched order. See also the switching Lemma C.2.5.

Remark 5.2.4 (Thread classes)**.** *In a language featuring thread classes (cf. e.g., [9]), a possible interface interaction is thread creation. Thread creation is an asynchronous interaction, i.e., the spawner of a thread is* not *blocked after issuing the spawn. As a consequence, when allowing thread creation interaction (or other forms of asynchronous communication), the rule* O-OO *would not be correct in general. See also Section 6.1.5 in the conclusion.* □

Remark 5.2.5 (Monitors)**.** *Java allows that methods are executed under mutual exclusion, specified by the* `synchronized`*-modifier. In a setting where the objects act as (re-entrant) monitors, inequations in addition to those of Table 5.2 are needed. In particular, a rule reversing* O-OI *can be added, if the two actions concern the same monitor. Cf. [10] and also Section 6.1.4 in the conclusion.* □

Let us illustrate the interplay between swapping and switching.

Example 5.2.6. *Consider the scenarios from Figure 5.5(a) – 5.5(c). From the perspective of the observer on the right, the three behaviors t_a, t_b, and t_c are indistinguishable: The observer cannot distinguish t_a from t_b, as the order of the two neighboring outgoing actions, here marked* 1 *and* 2*, cannot be determined (cf. rule* O-OO*).*[3] *Furthermore, the observer cannot distinguish t_b from t_c, because of its clique structure. From the perspective of the component (and assuming that the left-hand side consists of just one clique), t_b and t_c are clearly not equivalent, i.e., $t_b \asymp_\Theta t_c$ does not hold. However, t_a and t_b are equivalent due to switching, also from the perspective of the component.*

A further point can be seen from the scenarios. Comparing 5.5(a) with 5.5(c), we cannot separate the effects of $\asymp_\Delta$ and of switching in such a way that it is possible to transform the scenario of 5.5(a) into 5.5(c) by doing first only $\asymp_\Delta$ and afterwards apply the reordering via switching (or in the opposite order). I.e., $t_a \sqsubseteq_\Delta t_c$ and $t_a \sqsupseteq_\Delta t_c$ does not imply that $t_a \asymp_\Delta t'_a \sqsubseteq_\Delta^{switch} t_c$ for some t'_a. □

[3] The rules of Table 5.2 on the preceding page are formulated from the perspective of the component, not the observer, as indicated by the notation $\sqsubseteq_\Theta$. Hence, strictly speaking, the situation corresponds (from the perspective of the observing component) to rule O-II.

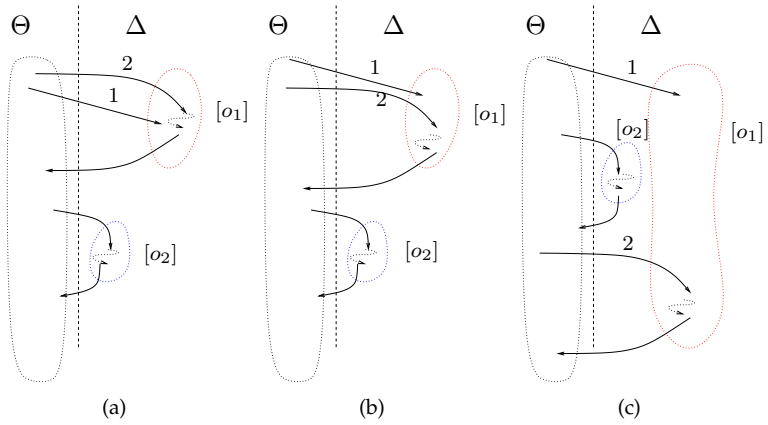


Figure 5.5: Swapping and switching

5.2.3 *Definability*

The core of completeness is a constructive argument: Given a trace, program a component which (1) realizes this trace, and moreover, realizes it exactly, (2) at least up-to the unavoidable imprecision of the semantics. Of course we can realize only traces which are actually possible, i.e., legal. Point (2) corresponds to the closure conditions above. This section provides the construction of the component from a given legal trace.

Outline of argument

Interestingly enough, the construction in the presence of thread creation almost completely corresponds to the construction in the single-threaded case. Remember Section 3.3.1 and 3.3.3 for an outline of the completeness argument and of the definability construction for the sequential language. Especially the data structures mentioned abstractly in that section can be used unchanged. The only two points in which the construction deviates or extends the old one are the following:

thread creation: The sending of *new* thread names across the interface must be realized in the code by appropriate thread creation. Incoming new thread names are unproblematic.

mutual exclusion: The core of the algorithm as explained in the sequential part can be used unchanged, i.e., the implementation of connectivity and the update of the corresponding information still works as before. In the multithreaded setting, however, we must curb the *concurrent access* to the common data structures to preclude destructive interference.

Basically, we must implement "synchronized" versions of the methods or algorithms of the sequential setting, synchronized, however, not on the level of objects, but on the level of cliques.

Of course, the traces are slightly more complex now in that they the labels now contain additionally the thread name and the sender objects for calls, as

we are dealing with augmented traces. The extension of the corresponding data structures for the implementation is straightforward.

Data structures and algorithms

As mentioned, the code for the observer in the concurrent setting here is similar to the one for the sequential setting. The key data structure is, as before, the static representation of the still open futures together with the role-bindings for the identities already encountered (cf. Definition 3.3.16). The only adaptation we need to do (at this level of abstraction) is to include *thread identities* into the data representation. In the same way as for object identities, each thread name n is statically represented by a corresponding instance variable of type $thread$, referred to by x_n or also $\check{n}$. Furthermore we need to change the data structure for labels ("type" label in the representation) such that it now contains the name of the thread as additional entity. Otherwise, the corresponding Definitions 3.3.16 and 3.3.17 can be reused.

The definition of the observer of a given trace is basically identical to the one in the sequential setting (cf. Definition 3.3.20), except that the initial thread is hidden now. The synchronization code t^i_{sync} and t^o_{sync} in the method body mentioned in equation (5.11) below (resp. (3.47) in the sequential setting) needs some adaptation here to deal with *race conditions* or contention, more precisely to assure that interaction with the component cliques is executed under *mutual exclusion.*

Definition 5.2.7 (Observer for trace t)**.** *Assume $\Xi_0 \vdash t : trace$. The* observer *for t, denoted by C_t, is defined as as follows. Each class mentioned in the commitment assertion Θ is equipped with the data structures as given in Definition 3.3.16, with $scripts = \bot$ and*

$$init = \{(\sigma_\bot, \check{t}_o) \mid t_o = {}_o{\downarrow} t,\ o \in names(t)\}\ . \tag{5.10}$$

Each public method $l : \vec{T} \to T$ of each component class c is implemented as

$$l \triangleq \varsigma(s{:}c).\lambda(\vec{x} : \vec{T}).t^i_{sync}(l, \vec{x}); t^o_{sync}\ . \tag{5.11}$$

If $\Delta_0 \vdash \odot$, then C_t contains no thread. If otherwise $\Theta_0 \vdash \odot$, then C_t is of the form

$$\Xi_0 \vdash C_t \ \triangleq\ \Xi_0 \vdash \nu(n{:}thread).(C'_t \parallel n\langle let\ x{:}c_i = new\ c_i\ in\ x.\rangle; x.start()\rangle) \tag{5.12}$$

for some class c_i with $\Theta \vdash c_i$.

The definition of C_t refers to code, which is shown in detail only later, in Section B in the appendix. We sketch here their functionality on an abstract level, only. The $t^i_{sync}(l, \vec{x})$ and t^o_{sync} (see Definition B.2.2 and B.2.11) is the code for "input" and "output synchronization", by which we mean, that $t^i_{sync}(l, \vec{x})$ and t^o_{sync} have play the scripts as illustrated in the overview of Section 3.3.3 in the deterministic setting. Input synchronization is performed *after* and incoming communication and output synchronization *before* an outgoing communication. The arguments, handed over in an input step from the environment, are remembered in the instance state. We use the notation $t^i_{sync}(l, \vec{x})$ as a reminder that the code for input synchronization contains the formal parameters $\vec{x}$ of the method freely, and l refers to the label of the method body the code

is contained in. The block syntax dealing with incoming returns (not visible at the level of Definition 5.2.7) will be of the form $t^i_{sync}(return, x)$, where x is the let-bound variable used to receive the return value (see Definition B.2.12).

The synchronization code for a method from equation (5.11) in particular contains a locking mechanism to assure mutually exclusive access to the data structures. Conceptually, the code $t^i_{sync}(l, \vec{x}); t^o_{sync}$ of equation (5.11) is of the form

$$(\!|\tilde{t}^i_{sync}(l, \vec{x})|\!)\ ;\ (\!|\tilde{t}^o_{sync}|\!)\ ,$$

where $(\!|$ and $|\!)$ mark the begin and the end of the critical section, executed under mutual exclusion (at the level of component cliques). The $(\!|$ and $|\!)$ are given in Definition B.2.21, using some locking scheme. See also the discussion below, how the implementation of mutual exclusion allows to reduce the arguments for the concurrent setting here to the arguments in the sequential setting. In the initial situation , described by equation (5.12) in case $\Theta_0 \vdash \odot$, the $|\!)$ at the very first object is used to *initialize* the lock of that object approproately, setting the lock to be "free", before the invocation of $x.start()$ kicks off the further execution of the thread, which starts with the first output synchronization (see Definition B.2.17).

We continue by showing total correctness of the construction, i.e., that C_t can indeed perform the trace t. In the inductive proof, we can reuse the judgment $\Xi \vdash [o] :: s$, asserting that the component C is able to perform the (global) trace s (cf. Definition 3.3.22).

Lemma 5.2.8 (Total correctness)**.** *Let t be a legal trace and $\Xi_0 \vdash C_t$ given as in Definition 5.2.7. Then $\Xi_0 \vdash C_t \stackrel{t}{\Longrightarrow}$.*

Taming concurrency or reduction to the sequential case The counterpart of total correctness is *partial correctness* or *exactness* of C_t: The component C_t can basically do nothing else than t (cf. Lemma 5.2.10, resp., definability from Corollary 5.2.11 below). The main complication, in comparison with the sequential setting, is the loss of exactness due to concurrency, reflected in the switching rules from the closure conditions of Table 5.2. In particular, the order of labels in a trace, coming from two different threads, can (in many cases) not be fixed absolutely by programming (cf. rules O-II, O-OO, and O-OI).

These additional closure conditions complicate the reasoning. In the following, we get rid of those switchings, such that we can argue as in the sequential setting. To do so means to *disentangle* the steps of a reduction sequence wrt. their threads. We illustrate the idea on a simple example: Assume a trace of two labels, $\gamma_1?\ \gamma_2!$, where the thread of γ_1 is n_1, and for γ_2, it is n_2:

$$\Xi \vdash C \stackrel{\gamma_1?}{\Longrightarrow}\stackrel{\gamma_2!}{\Longrightarrow}\ . \tag{5.13}$$

With a closer look at the single reduction steps, the sequence looks as follows:

$$\Xi \vdash C \underset{n_2}{\Longrightarrow} \bullet \xrightarrow[n_1]{\gamma_1?} \bullet \underset{n_1,n_2}{\Longrightarrow} \bullet \xrightarrow[n_2]{\gamma_2!} \bullet \underset{n_1}{\Longrightarrow} \bullet\ . \tag{5.14}$$

In the sequence, we assume for simplicity, that no threads other than n_1 and n_2 play a role. The threads carrying the internal steps are indicated below the respective arrow (and algebraic congruence "steps" are not shown; they are

not executed by any thread, anyway). In particular, the reduction sequence between n_1's incoming communication $\gamma_1?$ and n_2's outgoing communication $\gamma_2!$ can be a *mixture* of steps of n_1 and of n_2. To *disentangle* the steps of n_1 and n_2 amounts to reorder the steps such that those of n_1 trailing $\gamma_1?$ and those of n_2 preceding $\gamma_2!$ do not occur in this mixed manner. We call such a reduction, the result of the disentangling, *clean* (cf. also Definition C.4.1).

In general, of course, we *cannot* disentangle the sequence of equation (5.14)! In particular the sequence $\xrightarrow[n_1,n_2]{}^*$ can contain non-confluent $\xrightarrow{\tau}$-steps (accessing the instance state) which cannot be arbitrarily reordered (cf. the switching Lemma C.2.5). In particular, the order of steps corresponding to a read-write or a write-write conflict —one thread reads from an instance state and the second thread writes to the state, or both write— cannot be changed without endangering the outcome.

The key to make this disentangling possible is *mutual exclusion!* Reconsider the execution (5.13) and assume we are dealing with the interaction of a *single* component clique.[4] Writing $(\!|$ and $|\!)$ for the beginning and the end of the critical section, i.e., the code executed under mutual exclusion, the picture changes as follows. Conceptually, only one of the following 5 executions is possible:

$$\Xi_0 \vdash C \xrightarrow{\gamma_1?} \bullet \overset{(\!|1|\!)}{\underset{n_1}{\Longrightarrow}} \bullet \overset{(\!|2|\!)}{\underset{n_2}{\Longrightarrow}} \bullet \xrightarrow{\gamma_2!} \bullet \tag{5.15}$$

or

$$\Xi_0 \vdash C \overset{(\!|2|\!)}{\underset{n_2}{\Longrightarrow}} \bullet \xrightarrow{\gamma_1?} \bullet \xrightarrow{\gamma_2!} \bullet \overset{(\!|1|\!)}{\underset{n_1}{\Longrightarrow}} \bullet \tag{5.16}$$

$$\Xi_0 \vdash C \xrightarrow{\gamma_1?} \bullet \overset{(\!|2|\!)}{\underset{n_2}{\Longrightarrow}} \bullet \xrightarrow{\gamma_2!} \bullet \overset{(\!|1|\!)}{\underset{n_1}{\Longrightarrow}} \bullet$$

$$\Xi_0 \vdash C \overset{(\!|2|\!)}{\underset{n_2}{\Longrightarrow}} \bullet \xrightarrow{\gamma_1?} \bullet \overset{(\!|1|\!)}{\underset{n_1}{\Longrightarrow}} \bullet \xrightarrow{\gamma_2!} \bullet$$

$$\Xi_0 \vdash C \xrightarrow{\gamma_1?} \bullet \overset{(\!|2|\!)}{\underset{n_2}{\Longrightarrow}} \bullet \overset{(\!|1|\!)}{\underset{n_1}{\Longrightarrow}} \bullet \xrightarrow{\gamma_2!} \bullet .$$

Since $(\!|$ and $|\!)$ assure mutual exclusion, the reduction steps from the atomic section $(\!|1|\!)$ of n_1 either precedes $(\!|2|\!)$ of n_2, or vice versa.

A crucial difference concerns the reduction (5.15) and the four reductions of (5.16); in the first case, the $(\!|1|\!)$ occurs *before* $(\!|2|\!)$, in the latter, the order is opposite. In particular in (5.16), the order in which the two critical sections are executed is *reversed* compared to the order in which the corresponding *labels* $\gamma_1?$ and $\gamma_2!$ appear in the interface.

Another way to characterize the difference between the two groups of scenarios is that for those of (5.16), the steps of n_1 and n_2 are *not* cleanly grouped. For instance, $\gamma_1?$ is separated from the $\Longrightarrow$-reduction implementing the trailing critical section $(\!|1|\!)$ in the first reduction of (5.16); similarly for the other 3 reductions of that group. Given the situation (5.16), however, we can disentangle the execution, if we *switch* the externally visible steps $\gamma_1?$ and $\gamma_2!$, yielding

$$\Xi_0 \vdash C \overset{(\!|2|\!)}{\underset{n_2}{\Longrightarrow}} \bullet \xrightarrow{\gamma_2!} \bullet \xrightarrow{\gamma_1?} \bullet \overset{(\!|1|\!)}{\underset{n_1}{\Longrightarrow}} \bullet , \tag{5.17}$$

[4] This is to avoid that we also draw the tree structure of the semantics into the current discussion. In case that the steps of n_1 and of n_2 interact with two different component cliques, then they do not interfere with each other anyhow.

where the steps are switched in the first reduction from (5.16). Note that this switch is the *reverse* order as stipulated by O-OI from Table 5.2. However, the switching inequations in that table speak about switching *weak* steps, i.e., $\stackrel{a}{\Longrightarrow}$-steps, whereas the reduction sequence of (5.17) is obtained by switching two single reduction steps (which by themselves do not have any side effect on the instance state), when taking the first reduction of (5.16).

The common denominator of the scenarios from (5.15) and (5.16), both realizing the same observable sequence $\gamma_1?\ \gamma_2!$, is that by switching a number of execution steps, the reduction can be brought into a form where the steps of n_1 and n_2 are disentangled. Reduction (5.15) is already of this form, those from the second block can be transformed in a finite number of transposition steps.

As said, these transpositions are used in the *opposite* direction of the switching steps of Table 5.2 (cf. Lemma C.4.2). The order is relevant in particular for the combination of an input label $\gamma_1?$ followed by an output label $\gamma_2!$, illustrated in the scenarios above, which corresponds to the reversal of the switching rule O-OI. Note that Table 5.2 does not contain a rule O-IO. This is consistent with the observation, that given the sequence $\gamma_1!\ \gamma_2?$, there is no uncertainty in which order the respective atomic regions are positioned, namely reflecting the order of the external steps:

$$\Xi \vdash C \stackrel{(\!|1|\!)}{\Longrightarrow} \bullet \xrightarrow{\gamma_1!} \bullet \xrightarrow{\gamma_2?} \bullet \stackrel{(\!|2|\!)}{\Longrightarrow} \cdot \tag{5.18}$$

Remark 5.2.9 (Lock grabbing)**.** *Concerning the reductions (5.15) and (5.16), we remark the following. The use of* $(\!|$ *and* $|\!)$ *is a slight idealization (but no distortion) of the actual situation at the lowest level in that the notation seem to indicate an* atomic*, single step lock-grabbing. The lock handling operations* $(\!|$ *and* $|\!)$ *are encoded by terms of the calculus (cf. Definition B.2.21 and B.2.26). The lock-grabbing* $(\!|$*, e.g., consists of quite a number of elementary internal steps (which of course must assure that the lock is taken* as if *the action were atomic; that's the whole purpose of* $(\!|$*, after all). As a consequence, compared to the microscopic level of single reduction steps, the sequence of (5.15), for instance, is idealized in that it pretends that no elementary step of thread* n_2 *precedes the* $(\!|$ *of thread* n_2 *(and similar for* n_1*, etc.). More precisely, the reduction of (5.15) could look as follows:*

$$\Xi_0 \vdash C \underset{n_2}{\Longrightarrow} \bullet \xrightarrow{\gamma_1?} \bullet \underset{n_1}{\stackrel{(\!|1|\!)}{\Longrightarrow}} \bullet \underset{n_2}{\stackrel{(\!|2|\!)}{\Longrightarrow}} \bullet \xrightarrow{\gamma_2!} \bullet\ . \tag{5.19}$$

I.e., there may be actions of n_2 preceding *the atomic region* $(\!|1|\!)$ *of* n_1*, even if* $(\!|2|\!)$ *comes after* $(\!|1|\!)$*. Those steps of* n_2 *belong to the "trying section" of the mutex protocol:* n_2 *starts the protocol for acquiring the lock. However, thread* n_1 *intervenes, wins the race for entering the critical section, and only* after *it has left it again by executing* $|\!)$*, thread* n_2 *can enter. The reduction (5.15) is, however, no distortion of the general idea, in that, if (5.19) is possible, then, after reordering, (5.15), as well and with the same effect. In other words, we can consider* $(\!|$ *and* $|\!)$ *as atomic steps.* □

To sum up: the additional uncertainty of observation due to concurrency —switching— can be undone if the observer realizes mutual exclusion. We call *clean* a reduction, where the steps of the different threads do not occur in a mixed manner (see Definition C.4.1). Since each reduction can be turned into a clean one (by "disentangling"), the proof of partial correctness can be basically carried over from the sequential setting. See Section C.4.1 for further details.

Lemma 5.2.10 (Exactness/partial correctness)**.** *Let t be a legal trace and the observer $\Xi_0 \vdash C_t$ given by Definition 5.2.7.*

$$\textit{If} \quad \Xi_0 \vdash C_t \stackrel{s}{\Longrightarrow} \quad \textit{then} \quad b\Xi_0 \vdash s \sqsubseteq_\Theta t : \mathit{trace} \ . \tag{5.20}$$

Now we combine the result for clean reductions from Lemma C.4.3 with properties of the switching relation into completeness.

Corollary 5.2.11 (Definability)**.** *Assume $\Xi_0 \vdash t : \mathit{trace}$. Then there exists a component C with the following property: $\Xi_0 \vdash C \stackrel{s}{\Longrightarrow}$ if and only if $\Xi_0 \vdash s \sqsubseteq_\Theta t$.*

Theorem 5.2.12 (Completeness)**.** *If $\Xi_0 \vdash C_1 \sqsubseteq_{may} C_2$, then $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$.*

Part III

Conclusions

CHAPTER 6

Conclusion

This final section contains a short discussion of possible variations and extensions of the results, and also refers to related work.

6.1 Variations and extensions

We phrased our results in a specific calculus intended to capture the core features of current class-based, object-oriented languages like *Java* or *C#*, notably classes, objects, and threading. The core message when compared to the object-based case is that object-connectivity becomes important as part of the observable semantics.

The choice of features was motivated to capture this central aspect without complicating the main story (at least not beyond necessity ...). Here we discuss the effect on the development if some of the choices had been taken differently.

6.1.1 Constructors and constructor methods

The presented calculus does not contain constructor methods, or one could say only a trivial, default one, which means the constructor (method) is not programmable. This means it cannot be used for observations.

One can identify two levels of complexity when adding programmable constructors. In the simpler case, the constructors are simply *functional* and allow to pass values to the newly created instance. In this case, the syntax of classes is extended to $c\lambda(\vec{x}{:}\vec{T})[\![F, M]\!]$ and the fields of the class can refer to the formal parameters of the constructor. So the only way the constructor can be used is to store the handed-over values in the fields, but without further side effects or execution of code (in the development in the main body of the work, without constructors, the fields remain undefined after instantiation until set via methods).

The expression for instantiation then reads $new\, c(\vec{v})$ and the rule NEWO$_i$ for instantiation of component-internal objects (cf. Table 2.5) is replaced by:

$$c\lambda(\vec{x} : \vec{T})[\![F, M]\!] \parallel n\langle let\ x{:}c = new\, c(\vec{v})\ in\ t\rangle \rightsquigarrow c\lambda(\vec{x} : \vec{T})[\![F, M]\!] \parallel \nu(o{:}c).(o[c, F[\vec{v}/\vec{x}]] \parallel n\langle let\ x{:}c = o\ in\ t\rangle)$$

Of course, in *Java*, for instance, constructors are more flexible; they can be freely programmed (beyond just parameter passing) and thus act like *methods*. Syntactically classes this case can be represented by $c[\![\varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).t, F, M]\!]$ and the reduction rule for instantiation resembles a combination of the old rules for instantiation and method calls:

$$c[\![\varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).t_c, F, M]\!] \parallel n\langle let\ x{:}c = new\, c(\vec{v})\ in\ t\rangle \rightsquigarrow c[\![\varsigma(s{:}c)\lambda(\vec{x}{:}\vec{T})t_c, F, M]\!] \parallel \nu(o{:}c).(o[c, F] \parallel n\langle let\ x{:}c = t_c[o/s][\vec{v}/\vec{x}]; o\ in\ t\rangle)$$

To keep the issue of monomorphism of the type-system separate, there should be exactly one constructor method with fixed arguments.[1] One possible syntactic representation of classes would be to explicitly require that the

[1] Of course one could by convention arrange that without explicit constructor method, still the default one is present. But that's syntactic sugar. But constructor overloading, which we wish to avoid here, is a different issue again.

name of exactly one of the class methods coincides with the name of the class, similarly as the issue is handled in *Java*. Without (constructor or method) overloading, a better representation, i.e., a representation less reliant on "conventions", is to add as additional "unnamed" method in the class.

The absence of constructor methods made instantiation as such unobservable, i.e., we used a "lazy instantiation" scheme where cross-border instantiation appears at the interface trace only at the point when the first method of the instantiated object is called. The introduction of (functional) constructors or constructor methods requires some refinement.

With constructor methods, cross-border instantiation becomes observable, which means, one needs to introduce a corresponding new-label and include instances of the label in the traces. More concretely, for outgoing instantiation, the label could be written as $\nu(o{:}c).\nu(\Theta).\, new(\vec{o})!$, where o is the *environment* object being instantiated and Θ contains the bindings for the *component* objects from $\vec{o}$ whose scope extrudes by this instantiation (we omit from the discussion the case where also the thread name is new). Note that without lazy instantiation, there are no fresh environment objects in the label except the one being instantiated now. Similarly, the labels for outgoing calls, for example, simplify from $\nu(\Delta,\Theta).n\langle call\ o_r.l(\vec{v})\rangle!$ to $\nu(\Theta).n\langle call\ o_r.l(\vec{v})\rangle!$.

Functional constructors take some middle ground between constructor methods and the setting without (any but the default) constructors that we considered. Functional constructors cannot be used for immediate observation; in particular they cannot be programmed to report success. Therefore the exact point in time in the trace *where* the constructor is called remains unobservable. However, which arguments are handed over during instantiation must be recorded to take care of connectivity.

6.1.2 Class variables

The presented calculus is imperative: Objects contains as state the fields or instance variables, which can be updated. Class-based languages often feature another kind of updateable state, the so-called *static* variables or *class variables.*

If we introduced public (and non-final, i.e., with read and write access) static variables, the consequence to the semantics were rather drastic. Basically, they correspond to global "communication channels". Being globally known and accessible, the notion of separate cliques of objects, unable to communicate with each other, would break down. The crux of this work, however, was how the introduction of classes and of cross-border instantiation leads to a situation of groups of objects without any common communication channel.[2]

With *private* class variables, as understood as in *Java*, the situation would be more refined, in that only instances of the class have access to a static field restricted by the `private`-modifier. Thus, instances of a given class would be able to communicate and therefore would never be placed in separate cliques. This means the clique structure would get coarser. Indeed, [45] used a language featuring class variables.

[2] To be overly precise: The discussion around the question, what can be seen by more than one observer in Section 1.4.2 revealed that in a certain sense there is some common "hidden communication channel" between separate cliques of observers: If a later one succeeds, one can conclude that earlier ones at least did not get stuck.

A further consequence of static variables —public or otherwise— is that one crucial property in the relationship between classes and object disappears, which is: "Instances of a class are identical, except their name, until some interaction is performed that makes a difference". In its class variables and in the presence of constructor methods (cf. Section 6.1.1), the class can keep track, how many times it has been instantiated. This means, different instances of a class are no longer α-equivalent; the object itself is aware (via its class) whether it is the nth or the mth instance, and using this information it can behave differently from the start. In other words, instantiation looses the spirit of *name generation!* In the semantics, therefore, the ν-binder is no longer needed, and in particular, the labels in the trace need not distinguish between bound and free transmission of names. Clearly, also the issue of *replay* in the trace semantics disappears. So with connectivity collapsing and without ν-binders, the semantics would resemble quite close some standard trace semantics: The observational semantics of a program component is without much complications simply the set of traces.

6.1.3 Libraries

The presented general setup is completely symmetric: Component and environment interact symmetrically, together yielding a closed program. Which part of the program is the environment and which the component under observation is in the eye of the beholder: The observer of the component plays the role of the environment for the program under observation, but dually the program can be understood as the observer environment in the same manner.[3]

Considering now the observable behavior of a library, the situation is intuitively *asymmetric.* The user program uses classes from the library in that it instantiates objects from it, *but not vice versa.* This corresponds to the intuitive understanding that a library cannot instantiate user classes because it does does not know by name any classes of the user program. The initial state of a library is therefore characterized that it does not need any *assumptions* about its environment to be well-typed:

$$\vdash C : \Theta \,,$$

where Θ is static, i.e., it contains neither an instance nor an activity. Consequently, also the relational part E_Θ is empty. As a consequence, instantiation as one possible interaction between library component and observer only works in one direction. It is therefore a straightforward invariant that the observer forms one *single clique* of objects, and so for the observer. On the other hand, for the library this obviously does not hold and instantiation from the client program still can fragment the component objects.

Technically, the full abstraction result presented directly subsumes the one for the library case. In the restricted case, the construction for the completeness proof could is slightly simpler, as one need not take care of the creation and the merging of cliques. Furthermore one could do without the projection of traces onto single cliques. Nonetheless, the information propagation of identities within the single observer clique would still be required.

[3] The symmetry, however, is slightly broken in the definition of may-success, as the type system enforces that only the observer can report a success.

6.1.4 Concurrency control

The calculus we used (just as the object-calculus) has no native mechanism for *concurrency control.* Threads execute concurrently upon the shared state which is organized in objects forming the heap. Having shared state without means of protection against concurrent access and against interference by other concurrent activity is of course intolerable for practical programming.[4] *Java* (and $C^{\#}$) offer the following mechanism on the language level: Each object acts as *monitor,* i.e., it comes equipped with a *lock* which can be used for concurrency control, in particular, to control the access of threads to the instance.[5] More concretely, *Java* realizes the concept of *re-entrant* monitors [73][31][93]: A thread already owning the lock of an object can "re-enter" the monitor via (direct or indirect) recursion. See e.g. [7][5] for a Hoare-style proof-theoretic account of multithreading and reentrant monitors in *Java*.

Adapting the framework presented here to incorporate *monitors,* in particular `synchronized` methods, requires a number of extensions.

To start with, it is easy to see that the presence of synchronized methods changes what is *observationally equivalent*. Indeed, sticking to may-testing as notion of observation, the implied notion of observational equivalence in a language with monitors is *incomparable* to the one in a language *without* monitors. This is illustrated by the following two examples.

Example 6.1.1 (Synchronized methods decrease distinguishing power)**.** *This example shows, how the presence of locks in the observer renders certain observations impossible, i.e., using synchronized methods one looses discriminating power. Consider the following two traces:*

$$\begin{aligned} t_1 &= \gamma_1!\,\gamma_2? \\ &= \nu(n_1{:}\mathit{thread}, o_0{:}c_0).n_1\langle \mathit{call}\ o_1.l_1(o_0)\rangle!\ n_1\langle \mathit{call}\ o_0.l_0()\rangle? \end{aligned} \tag{6.1}$$

and

$$\begin{aligned} t_2 &= t_1\ \gamma_3! \\ &= t_1\ \ \nu(n_2{:}\mathit{thread}).n_2\langle \mathit{call}\ o_1.l_2()\rangle!\ . \end{aligned} \tag{6.2}$$

More precisely, consider the components C_{t_1} and C_{t_2} performing t_1, respectively t_2. Note that clearly such components C_{t_1} and C_{t_2} exists; in particular, in the setting with synchronized methods, the component C_{t_2} is possible (which implies that C_{t_1} is possible as well, and also the setting with non-synchronized methods makes the realization only easier).

Now, in the non-synchronized setting, the following observer distinguishes between C_{t_1} and C_{t_2}: The initial thread starts in the component, and the observer reports success as soon it has seen $\gamma_1!\ \gamma_2?\ \gamma_3!$, i.e., the longer t_2. Obviously, confronted with C_{t_1}, the observer will not report success, since $\gamma_3!$ is missing in the observation, but it reports success with C_{t_2} (for which the observer was tailor-made).

It is almost as easy to see that C_{t_1} and C_{t_2} cannot *be distinguished in the synchronized setting.*[6] *Looking at the traces, the* only *difference is the additional outgoing*

[4] *Software* solutions at user level for the mutual exclusion problem or the interference problem, for instance, do not qualify as practical approach.

[5] We assume in the discussion the discipline we also used in the technical development, namely that the fields of an object are accessed and changed only via methods but not directly. In particular, in a concurrent setting, direct field access is a non-advisable programming practice ...

[6] Since we have to argue about *all* possible observers which are unable to see a difference as

call γ_3! of thread n_2. This call cannot be observed, because in order to be observed it must enter the monitor o_1 but that is guaranteed to be impossible: No matter how the observer is programmed, the lock of o_1 is taken for sure by thread n_1 after t, and thus n_2 cannot enter that monitor. □

Example 6.1.2 (Synchronized methods increase distinguishing power)**.** *In contrast to the previous example, this one indicates that the presence of locks can increase the accuracy of discrimination. Consider the following trace:*

$$\begin{aligned} t &= \gamma_1?\,\gamma_2!\,\gamma_3?\,\gamma_4! \\ &= \nu(n_1{:}thread, o'{:}c').n_1\langle call\ o_1.l(o')\rangle?\ n_1\langle call\ o'.l()\rangle! \\ &\qquad \nu(n_2{:}thread, o''{:}c').n_2\langle call\ o_2.l(o'')\rangle?\ n_2\langle call\ o''.l()\rangle!\ . \end{aligned} \tag{6.3}$$

First, the observer invokes a method of a component object o_1, which is answered by the component with an outgoing call. Next, the observer calls another component object o_2 via a new thread, which is followed by a further outgoing call of that second thread.

In a setting without locks, the last outgoing call can be implemented by the component in two different ways: (1) o_2's method l directly *calls back l of o'', or (2) o_2 does an* internal *call to o_1 which then realizes the outgoing call.*

With locks, the latter implementation would not lead to the last outgoing call of trace t, since object o_1 is locked by thread n_1, and therefore cannot realize the call in this situation. Thus, an observer whose success report depends on the last outgoing call could distinguish components implemented in the first or in the second manner, whereas no observer in a setting without locking could tell them apart. □

It is relatively straightforward to extend the syntax and the semantics to deal with re-entrant monitors. The basic syntactical extension is to equip objects with a flag indicating whether the lock is free or whether it has been taken by a thread n, as expressed by the following two syntactical phrases

$$o[c, F, n] \quad \text{and} \quad o[c, F, \bot_{thread}]\ , \tag{6.4}$$

representing objects, where n is the name of a thread and $\bot_{thread}$ a specific name (but not a value) denoting that the lock is free. A bit more thought requires the design of the operational semantics. To maintain the clean decoupling of environment and component, i.e., to maintain a clean assumption/-commitment framework, it is best to have the *internal* steps deal with lock-grabbing and lock-release. See Table 6.1 for a formalization of the corresponding internal steps. Having the internal semantics responsible for lock-handling implies that the external steps are basically unchanged, at least wrt. the component part. As far as Θ-locks, i.e., the locks of the component, are concerned, an incoming communication, in particular an incoming call, is *always* possible, since it is *not* the interface action which takes the component lock or blocks, but a subsequent internal step (cf. the rules CALLI$_i^s$).

This *non-atomic* lock handling allows a clean semantical decoupling of component and environment. However, separating the visible interface interaction from the action lock-handling introduces a uncertainty of observation, which makes it harder to characterize when a lock is free, resp., taken. In other words, the characterization of the interfaces behavior, the definition of the legal traces, becomes more complex.

opposed to find a single one that sees the difference, the argument now is conceptually more complex. However, the two components C_{t_1} and C_{t_2} are quite simple.

In particular, without atomic lock handling at the interface, the trace of interface interaction contains not enough information to observe *exactly* in all situations when the lock is taken or not. With atomic lock handling, one definitely knows that after observing trace $t\ n\langle call\ o_r.l()\rangle?$, the lock of the component object o_r is taken (assuming that l is a synchronized method).

If the lock management, as sketched, is handled by the internal steps from Table 6.1, the lock of the callee o_r *may* or *may not* be taken yet. Whether it is taken or not depends on the history t —if the lock has been taken definitely after t, then this still is true after $t\ n\langle call\ o_r.l()\rangle?$— and on the (non-observable) internal scheduling: If the incoming call is such that it *applies* for the lock of o_r, then after the call, the thread *may* own the lock, i.e., there are states where it does not yet hold the lock and there can be states where it owns it. Only after a further subsequent outgoing call, one has the definite, observable knowledge that the thread now *must* hold the lock. The description of the legal traces, as it seems, works with may- an must- approximation of lock-ownership. Concentrating on a single thread and writing $\Xi \vdash_\Theta t : \Diamond o$ for "the thread may own the lock of component object o after trace t (and analogously for "must" represented by $\Box$ instead of $\Diamond$), the formalization uses rules as shown in Table 6.2.

Based on these abstractions for lock-ownership, one can define when a trace r can be extended by an additional action a without violating *mutual exclusion.* For instance, when a lock is known to be taken for sure, all other interaction with the concerned monitor must either happen *before* the lock is taken, or *after* it has been released again. This concerns also possible *data dependencies* in that it is not sufficient that a (new) value is handed over at the interface to be used by trailing reaction, it must be delivered to the monitor. See [10] for more details, where we formalize the ideas sketched above, based on $\Diamond$ and $\Box$ approximations for lock ownership and three kinds of causal dependencies calculated from a trace: Data-dependence, control-dependence, and dependence due to mutual exclusion. The formalization is carried out for a multithreaded setting *without* classes, i.e., concentrating on the problem of mutual exclusion, but without connectivity and cliques.

As for future work, one can consider the combination of merging clique structure and the lock handling. The combination is not completely straightforward, since the update of the connectivity structure would have to respect the monitor discipline: If some references are handed over at the interface, which lead to a merge of cliques, this merge is not effective until the corre-

$$c[\![F, M]\!] \parallel o[c, F', \bot_{thread}] \parallel n\langle let\ x{:}T = o.l^s(\vec{v})\ in\ t\rangle \rightsquigarrow$$
$$c[\![F, M]\!] \parallel o[c, F', n] \parallel n\langle let\ x{:}T = M.l^s(o)(\vec{v})\ in\ release(o); t\rangle \qquad \textsc{Call}^s_{i_1}$$

$$c[\![F, M]\!] \parallel o[c, F', n] \parallel n\langle let\ x{:}T = o.l^s(\vec{v})\ in\ t\rangle \rightsquigarrow$$
$$c[\![F, M]\!] \parallel o[c, F', n] \parallel n\langle let\ x{:}T = M.l^s(o)(\vec{v})\ in\ t\rangle \qquad \textsc{Call}^s_{i_2}$$

$$o[c, F, n] \parallel n\langle let\ x{:}T = release(o)\ in\ t\rangle \xrightarrow{\tau} o[c, F, \bot_{thread}] \parallel n\langle t\rangle \qquad \textsc{Release}$$

Table 6.1: Internal steps: Lock handling

$$\frac{receiver(t\ \gamma_c?) = o}{\Xi \vdash_\Theta t : \Diamond o}\ \text{M-I}\Diamond \qquad \frac{\Xi \vdash t : \Diamond o}{\Xi \vdash_\Theta t\ \gamma_c? : \Box o}\ \text{M-O}\Box$$

Table 6.2: May and must lock-ownership (example rules)

sponding action enters the monitor. This does not seem a conceptually hard problem; however, the notation/formalization would become quite "heavier", when considering the graphs of dependencies for mutual exclusion and the connectivity graphs at the same time. Besides that, one should make the monitor setting more realistic by adding the thread coordination mechanism of wait and signal (*Java*'s `wait` and `notify` methods).

Remark 6.1.3 (Concurrency control)**.** *In the definability construction underlying the completeness proof in the concurrent setting, we implemented a form of concurrency control, introducing the constructs* $(\!|$ *and* $|\!)$*, where* $(\!|\ t\ |\!)$ *was used for the execution of* t *without interference by other threads. Indeed, the constructs were* implemented *using some form of locks as part of the objects. Two remarks in comparison with the native monitor concept of* Java *are in order.*

First of all, the may testing framework used for observational equivalence puts us, as implementors of the observer component, in a rather comfortable position: For the proof, it is not required to solve the full blown mutual exclusion problem (cf. e.g. Dijkstra [49]). It is just required to prevent *interfering runs in the critical sections bracketed by* $(\!|$ *and* $|\!)$*, but not to* resolve *a situation with concurrent attempt to access a critical section, for instance by blocking all competitors except one until the lock is free again. The may-setting, i.e., the restriction on the safety aspect of mutual exclusion, thus allows that one can simply stop executing (by termination or divergence) once the critical section is attempted of being entered by a second thread. Obviously, this simplifies the implementation of* $(\!|\_|\!)$ *considerably, in particular considering that we needed non-interference not for just one object, but on the level of cliques of objects. It allows basically that the conceptual lock for the clique of objects can be implemented in a distributed manner.*

Secondly, the mechanism implemented in the completeness proof was not required to support reentrant monitor locks. To be sure: In the implementation in Section 3.3.3, resp., 5.2.3, reentrant method calls occur, especially in the identity propagation algorithm. However, the synchronization code does not make nested use *of the* $(\!|\_|\!)$*-brackets: One particular thread executing its synchronizing code, acquires the lock (if available) once, performs its task and then gives the lock back. Thus the locks themselves may be oblivious of the identity of the threads.* □

6.1.5 Thread classes

The basic syntactic change from the object-based to the *class-based* setting presented in Section 2.2 respectively Section 4.2 was the addition of classes as *"generators of state"*. In contrast to the object-based setting, this allowed to generate or instantiate new objects across the component boundaries. Objects (as classes) are passive entities; the active part of the program is represented by threads. Indeed, in the multithreaded setting, there was also a mechanism for

"generating new activity", i.e., for creating new threads. The thread instantiation mechanism, however, corresponds more to the situation of object instantiation in the object-based setting, since the code t in the thread instantiation expression $new\langle t\rangle$ is directly given. In other words, for threads, there is no *cross-border* generation. In *Java* (or similarly in $C^{\#}$), however, the designers choose to entangle the concepts of thread and classes respectively objects in a particular way[7] in that certain objects are instances of so-called *thread classes* and they contain one particular method, the `start`-method, which can be called at most once and which spawns a new thread.

Remark 6.1.4 (Thread classes)**.** *The details in Java are a bit more convoluted and rely on sub-classing and overriding. The actual code which constitutes the initial sequential part of the new thread is contained in a method called* `run`*, whose body is provided by the user. The* `start`*-method calls (typically) the run-method and spawns the new activity. The situation there is insofar a bit muddy, in that the concepts are not cleanly separated and their functioning relies to some part on conventional correct use. For instance, the start method may be overridden, such that it does no longer call the run method. Furthermore, nothing prevents the user to directly invoke the run-method as many times as wished, even if that's probably not what the run-method was designed for, namely to provide the code for a new thread. Finally, it is perfectly ok, that the "thread object" serves the program also in a role as container for data and other methods; after all it is not forbidden that a thread object contains fields and further methods beside the (inherited) start- and the overridden run-method.*

In that set-up, creating a new thread amounts to instantiating a new object and invoke its start-method. Since the method, as mentioned, can be called at most once, the thread identity can be identified with the identity of the object where it started its life. We adopted this model in the proof-theoretical account of multithreaded Java (Java$_{MT}$), for instance in [8]. For a comprehensive account see especially Ábrahám's thesis [5]. Relevant for our current discussion is only that cross-border generation of threads is possible, if the code of the new thread is contained in a "class". □

In our setting, we can introduce following construct for thread classes to the syntactic category of component (cf. Table 4.2):

$$c\langle\!\langle t_a \rangle\!\rangle \quad \text{with} \quad t_a \triangleq \lambda(\vec{x} : \vec{T})t \; .$$

The phrase t_a is the body of the thread class, abstracted over its arguments. This, the interface type of a thread class is $T_1 \times \ldots \times T_n \to \mathit{thread}$. Unlike ordinary classes, where we omitted constructor methods, thread classes *must* have a mechanism that allows to hand over arguments during instantiation (cf. Remark 6.1.5). The reason is that in the formalization, instantiation at the same time means the thread starts executing. An alternative would be, to have an designated *method* (like `start` in *Java*) which gets the new thread running, and which could be used to hand-over values to the thread.

Concerning typing, the system from Table 4.3 and 4.4 is to be extended by the following rules of Table 6.3. Also the operational rules for thread creation are rather straightforward. As for ordinary classes, one distinguishes between internal and external thread creation (cf. Table 6.4 resp. 6.5; we show only the additional rules).

[7] Probably to avoid introducing another core concept into the language at the syntactic level.

$$\frac{;\Delta, c_t{:}T \vdash \langle\!\langle t_a \rangle\!\rangle : T}{\Delta \vdash c_t\langle\!\langle t_a \rangle\!\rangle : (c_t{:}T)}\ \text{T-NTCLASS}$$

$$\frac{\Gamma, x_1{:}T_1, \ldots, x_k{:}T_k; \Delta \vdash t : none}{\Gamma; \Delta \vdash \langle\!\langle \lambda(\vec{x}{:}\vec{T}).t \rangle\!\rangle : \vec{T} \to thread}\ \text{T-TCLASS}$$

$$\frac{\Gamma; \Delta \vdash c_t : \vec{T} \to thread \quad \Gamma; \Delta \vdash \vec{v} : \vec{T}}{\Gamma; \Delta \vdash spawn\, c_t(\vec{v}) : thread}\ \text{T-SPAWN}$$

Table 6.3: Typing for thread classes

$$c_t\langle\!\langle \lambda(\vec{x}{:}\vec{T}).t_2 \rangle\!\rangle \parallel n_1\langle let\ x{:}T = spawn\, c_t(\vec{v})\ in\ t_1 \rangle \rightsquigarrow$$
$$c_t\langle\!\langle \lambda(\vec{x}{:}\vec{T}).t_2 \rangle\!\rangle \parallel \nu(n_2{:}T).(n_1\langle let\ x{:}T = n_2\ in\ t_1 \rangle \parallel n_2\langle t_2[\vec{v}/\vec{x}] \rangle) \qquad \text{SPAWN}_i$$

Table 6.4: Internal steps (thread classes)

An internal spawning of a thread works similar to internal object creation. In particular, a new scope is introduced which hides the name of the new thread outside the spawning thread.

$$\frac{\begin{array}{c}\acute{\Xi} = \Xi + o_s \xrightarrow{a} \odot_n \quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\hookrightarrow} \odot_n : thread \\ a = \nu(\Phi').\langle spawn\ n\ of\ c_t(\vec{v}) \rangle ? \quad \acute{\Theta} \vdash \odot_n \quad \Delta \vdash o_s \quad \Theta \vdash c_t \quad \Sigma' \vdash n\end{array}}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle c_t(\vec{v}) \rangle}\ \text{SPAWNI}$$

$$\frac{\begin{array}{c}a = \nu(n'{:}thread, \Phi').\langle spawn\ n'\ of\ c_t(\vec{v}) \rangle ! \quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi_1 \quad \acute{\Phi}_1 = \Phi_1 \setminus \Phi' \\ \Delta \vdash c_t \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} \odot_{n'}\end{array}}{\Xi \vdash \nu(\Phi_1).(C \parallel n\langle let\ x{:}T = o_s\ spawn\, c_t(\vec{v})\ in\ t \rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Phi}_1).(C \parallel n\langle let\ x{:}T = n'\ in\ t \rangle)}\ \text{SPAWNO}$$

Table 6.5: External steps (thread classes)

Remark 6.1.5 (Thread constructors)**.** *The spawner can hand over values to the new thread via the thread constructor. One consequence of thread constructors is that there is no "lazy instantiation" for new threads; without thread constructor, there would be no need for a separate spawn-label, since instantiation would not be immediately observable, as for class instantiation, where we have* not *considered class constructors.*

In case of threads, however, the absence of constructors would be more drastic: Without acquaintance with objects handed over at instantiation time, the new thread would not be able to contact any of the existing objects in the component as well as in the environment. The spawner is "acquainted" with the new thread in that it knows its identity but it cannot "communicate" with its child thread. In some sense, the only

point where the spawner can communicate with the threads is during instantiation and without this possibility, the multithreading would degenerate to a program consisting of groups each with one single threads which are globally *completely separate, i.e., not simply separate when considered the connectivity as seen from the component or the environment. Note that it does not mean that for instance an environment thread created by the component via* L-SPAWNO *cannot "call back" to the component, only that for calling back it need to create its own cliques of objects unrelated to the rest.* □

In [12] we formulated an operational semantics for thread classes. In contrast to the language presented here, a further difference is that thread names can be communicated via message passing. Besides that the paper explores a different representation of the legal trace system, namely one where the branching nature of the merging clique structure is directly reflected in a branching of the legal trace system. As for full abstraction (not covered in that paper), the most significant change, as it would seem, is that the closure conditions would have to be adapted. In particular, thread creation is an *asynchronous* communication, in that the thread performing the spawn action is not *blocked* by that step. In our lazy instantiation setting, where the instantiation gets visible only by the first cross-border call, the call action leads to a blocked thread (waiting for return). As a consequence, the switching inequations from Table 5.2 cannot be used identically for spawn-labels, but need adaptation. The question of the exact formulation (and definability, etc.) is left for future research.

6.1.6 Cloning

An extension with interesting semantical consequences is *cloning*. Syntax, typing, and operational behavior are straightforwardly defined. We assume a parameterless clone-method. For simplicity, let each object be cloneable. To be cloneable, a value must be an object, and the clone has the same class type as the original object:

$$\frac{\Gamma;\Delta \vdash v : c \qquad \Gamma;\Delta \vdash c : [\![T]\!]}{\Gamma;\Delta \vdash v.\, clone() : c}\ \text{T-CLONE}$$

Cloning is similar to instantiation: The clone-method cannot be used to program any observations and we assume an unbounded heap. In case of instantiation, we introduced the mechanism of lazy instantiation: The new instance is created only "on demand", i.e., the first time it is used in a method call, either as target object or as argument passed across the border.

The absence of programmable constructor methods allows to postpone the actual creation of the new instance in instantiation because we can be sure that the first time someone calls the object, it still is in its pristine, initial state (cf. rule NEWO_{lazy} in Table 2.11). The same holds for cloning: Even if the clone o', when created as a shallow clone from an extant object o, "inherits" the connectivity of o, since all its references are copied, the rest of the objects, in particular o, does *not in turn* know o'!

With cloning not being immediately observable, there is not need to introduce a clone label. Similarly, we did not include an explicit instantiation label.

Cloning an existing internal object (cf. rule CLONE_i) just copies the state of the object and creates a new local scope for the freshly created reference. This

corresponds to the notion of *shallow* clone. The corresponding rule is shown in Table 6.6.

$$n\langle let\ x{:}T = o.\ clone()\ in\ t\rangle \parallel o[c, F]$$
$$\rightsquigarrow \qquad \nu(o'{:}c).(n\langle let\ x{:}T = o'\ in\ t\rangle \parallel o'[c, F]) \parallel o[c, F] \qquad \textsc{Clone}_i$$

Table 6.6: Cloning (internal step)

But how to treat the cloning of an external object, i.e., what is the analog to $\textsc{NewO}_{lazy}$? A first attempt could look as follows, quite similar to the rule for instantiation:

$$\frac{\Delta \vdash o : c}{\Xi \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}c = o.\ clone()\ in\ t\rangle) \rightsquigarrow \acute{\Xi} \vdash \nu(\Phi, o'{:}c).(C \parallel n\langle let\ x{:}c = o'\ in\ t\rangle)}$$

However, the rule ignores the difference between instantiation and cloning, namely that the cloned object "inherits" the connectivity of the original object. When later the object o actually is exported to the environment, e.g., in that the component calls one of its methods, o' will have no connectivity except the connections handed over by the call itself, which correctly describes the situation for instantiation, but not for cloning.

Note that an "eager" approach, exporting the clone o' immediately together with the connectivity of the original o would be wrong, as well:

$$\frac{\Delta \vdash o : c \qquad \acute{\Xi} = \Xi + o' \hookrightarrow \Xi(o)}{\Xi \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}c = o.\ clone()\ in\ t\rangle) \rightsquigarrow \Xi \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}c = o'\ in\ t\rangle)}$$

In the premise, $\Xi + o' \hookrightarrow \Xi(o)$ is meant as adding all acquaintances of o according to Ξ to the new o'. The rule is wrong, as in the extended context $\acute{\Xi}$, not just o' knows all references that o knows, but also inversely, since connectivity is interpreted as the reflexive, symmetric, and transitive closure of $\hookrightarrow$ (as far as environment objects are concerned in the above situation of output).

To formalize *lazy cloning*, we can generalize the framework as follows: The ν-constructor for components must be extended such that it hides not only references, but also their connectivity. I.e., we need to consider bindings of the form $\nu(\Delta', E'_\Delta).C$, where Δ' contains the lazily instantiation or lazily cloned environment objects, and E'_Δ their "inherited" connectivity:

$$\frac{\Delta, \Delta' \vdash o : c \qquad \acute{\Xi} = \Xi, \Xi'}{\begin{array}{l}\Xi \vdash \nu(\Xi').(C \parallel n\langle let\ x{:}c = o.\ clone()\ in\ t\rangle) \rightsquigarrow \\ \qquad \Xi \vdash \nu(\Xi', o{:}c; o' \hookrightarrow (\acute{\Xi}(o)).(C \parallel n\langle let\ x{:}c = o'\ in\ t\rangle)\end{array}}$$

Consequently, the external labels do not just exchange information about fresh names, but also *connectivity* information. I.e., labels are of the form $\nu(\Xi').\gamma$,

and the context update in the external steps and the legal traces system needs to be adapted accordingly.

A further consequence is that if the observer can procure itself copies of an object makes choice points in the behavior of the objects observable, i.e., it exhibits the branching structure of the behavior.

Example 6.1.6 (Cloning)**.** *The example represents the prototypical example distinguishing between a linear (trace based) semantics and a branching semantics, such as bisimulation (cf. e.g. [102]). The example is schematically shown in Figure 6.1. With*

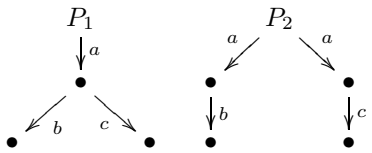


Figure 6.1: Branching

the observer having the power to create a duplicate after the a-action, it can distinguish the left and the right process: In the first case, the original process and its copy can do b and *c, while in the second case, both can do either only b or only c. This idea is easily representable in the object calculus as follows; here the essence of the example is programmed in* Java*:*

Listing 6.1: Branching and cloning

```
class P1 implements Cloneable {
  private int x = 0;
  private java.util.Random gen = new java.util.Random();

  public Object clone ()   {
    try {return super.clone();}              // use the native clone-method
    catch(CloneNotSupportedException e) {    // just catch it.
    }
    return new P2();                         // unreachable
  }

  public void choose () { x=gen.nextInt(2)+1;return;}     // x in {1,2}

  public void  a() {return;}
  public void  b() {
    this.choose();
    if (x==1) {return;} else {System.exit(0);};
  }
  public void  c() {
    this.choose();
    if (x==2) {return;} else {System.exit(0);}
  }
}

public class O {  // component
  public static void main(String[] arg) {
    P1 x = new P1();
    x.a();
    P1 y = (P1)x.clone();
    x.b();y.c();
```

```
      System.out.println("success");
    }
}
```

The actual behavior is slightly more complex, as the interaction between the environment and the instance is not atomic as sketched in Figure 6.1 but consists of a pair of call and return, so the label a of the abstract example corresponds now to the call and the return of the a-method etc. Furthermore, the calls are always enabled.

The code shows only the first alternative P_1, where the choice is taken after *the cloning. The schematic figure is imprecise insofar, as it looked as if after a, both b and c were enable, while the code internally chooses between b and c. Clearly, with an instance of P_1, the observer may report success. On the other hand, in case of P_2 the observer can never report success.*[8] *Hence $P_1 \not\sqsubseteq_{may} P_2$.* □

Not only the observer can duplicate objects of the component, also the program can apply cloning to the observer. The detailed consequences on the semantics are left for future research. A final word on cloning: The complications entailed by cloning are *not* caused by cloning alone, it is the possibility of cloning across the environment/component border, that exhibits the branching structure.

6.1.7 Subtyping and inheritance

One major feature of class-based object-oriented languages not tackled here is *inheritance*. Also subtyping is employed only in a simple and restricted manner. It is commonly accepted that inheritance and subtyping are conceptually different [41][23][134]. Subtyping is one specific form of polymorphism, popularized by object-oriented languages. The types of a language form a partial order, and characteristic of subtyping is that an element of a smaller, more special, type can safely used at places where an inhabitant of a larger, i.e., more general, type is expected ("subsumption"). Inheritance, on the other hand, is a mechanism for *code* reuse, mostly in the form of class inheritance, where a sub-class inherits code from its super class(es).

Subtype polymorphism and hiding

The type system presented in Section 2.3, resp., in Section 4.3 is almost monomorphic. A type system is monomorphic, if each program or term has at most one type; if not, it is *polymorphic.* Typically, object-oriented languages (or, perhaps better, modern programming languages ...) are polymorphic in various different ways. See e.g., [39] for a well-known classification of various flavors of polymorphism. Anyway, our type system from Section 2.3 contains exactly two points where the strict monomorphic type discipline is broken. One concerns the type of the stopped process, which has any type. This reflects the fact that the types are *partial correctness* assertions, and since the control-flow never reaches the point after a stopped thread and especially a thread does not return any value, *stop* has any type. For the same reason, the auxiliary block- and return-expression have any type.

[8] The code of P_2 is not shown. It differs from P_1 simply by moving the call to `this.choose()` from the body of method a into both methods b and c.

Apart from that, the type system injects a small quantum of *structural subtyping* into the derivation system, allowing a rudimentary subtype polymorphism, sometimes known as *width subtyping*. Indeed, when considering closed programs using the internal semantics we could as well formulate the type system without subtyping without changing the semantics. Subtyping just allows to use type declaration more precise than actually provided by the implemented methods (cf. Definition 2.3.1 for the subtype relation $\leq$ on interface types). So subtyping is a question of *hiding* (but of nothing else) and due to subject reduction, this additional flexibility of the type system has no run-time significance.

The aspect of hiding is also the reason why subtyping is needed in the *external* semantics, where the type information is part of the component *interface*, both as assumptions and as commitments. Since we allow observations only for well-typed programs, the amount of publicly available information *does* have a semantical import. For instance, already the external semantics is formulated by steps between typed judgments $\Delta \vdash C : \Theta$ and objects not exported by the commitment Θ cannot be called from outside, and likewise, methods not mentioned in the type interface because they are hidden due to subtyping cannot be invoked from outside. After all, that's the meaning of hiding: It makes things unobservable ....

Note in this context, that the ν-binders allows to hide names to the outside, as well. However, the more fine-grained export of class interfaces possible by width-subtyping —some methods are public to the environment and some are not— plays a crucial role in the completeness proof. The programmed observer makes use of a number of private methods, which must not be available for its environment. Whether the full abstraction result in the calculus is possible without this form of hiding is unclear.

It seems possible to achieve full abstraction without subtyping but with the possibility to hide classes via the ν-binder. The programming of the observer, the core of the completeness, would be quite different from the one presented in Chapter B. Without private methods but with hidden classes, the realization could not rely on the "distributed" implementation of the connectivity of E_Θ as given in Chapter B, where each object keeps track of its share of connectivity, i.e., all the object names it has ever learnt, which where the information is broadcast to all clique object during execution to keep the information in sync.

Instead, one would have to *centralize* the data-storage concerning connectivity and the "playing of the scripts" and the task of achieving mutual exclusion into a central server or broker, whose class can be kept hidden and outside the reach of the environment. Clearly, the implementation cannot rely on a *single* broker, but there would have to be one for each clique. Just using a centralized solution and exploiting class hiding does not allow to connect objects in different cliques. Hence, in that form of solution, one would have to program a dynamic number of broker objects, which are created when new cliques are created and which must be combined into a single broker when cliques are merged. An observer using such a centralized implementation has been presented in [66]. It is interesting that the form of hiding —hiding of classes vs. hiding of methods via subtyping— dictates the realization of the observer — centralized broker solution containing all the bookkeeping code in one spot vs. a decentralized solution, distributing the code over the objects.

Without any form of hiding, the result seems impossible or at least complex beyond hope. The coding of the observer must, in general, rely on entities of the language not available for the environment. Without hidden classes and without hidden methods in otherwise public classes, the only programmable entity *not* exported to the outside are threads. As threads themselves only contain a local state and cannot access the instance states other than by methods (which are all public) one possible realization would have to rely on some "encoding" of the semantics in thread configurations (which seems impossible). Alternatively, if the semantics is to be encoded into instance variables (as we did in Chapter B), their manipulation would require to use the public methods, since instance variables cannot be directly accessed across object boundaries. To make that scheme work would then require that the body of the method can discriminate "real" calls from the environment, and the use of the method in an internal calls just to update the book-keeping of the instance state and to realize the required behavior. However, the strict typing discipline forbids any straightforward solution for that. In particular, in absence of method overloading, we cannot simply pass an additional argument as indication that the method is to be evaluated as internal call. To put it differently and more precisely: If the type system allowed this, then also the environment could pass the additional argument and we would have gained nothing. Also this route does not look promising.

Inheritance

The semantical impact of inheritance is more considerable than that of subtyping. However, even if conceptually different, subtyping and inheritance are often related. A typical choice is that "inheritance implies subtyping", i.e., the type of instances of a class is a subtype of the instances of super-classes.

In our setting, where objects are typed by the name of their class, the natural way to introduce subtyping in connection with inheritance is *nominal subtyping*. In this scheme, a class c_1 is a (direct) subtype of another c_1 exactly if c_1 is *defined* to inherit from c_2.[9] *Java*, e.g., uses the `extends`-keyword to express inheritance, which implies subtyping.

Apart from subtyping, the main complication when introducing inheritance is that the component becomes open in one more aspect. Classes of the component cannot just be used for cross-border instantiation, but also for "cross-border inheritance". This works in two directions: The environment can make observations about the component by inheriting from component classes, but also by inheriting code *to* the component. This makes more details of the component observable. Sometimes this is known under the slogan "inheritance breaks encapsulation".

Part of this phenomenon is also called the *fragile base class problem* [100] [131] [127]. Listing 6.2 presents a simple example.

Listing 6.2: Fragile base class

```
class A {
    void add () {...}
    void add2 () {...}
```

[9] In Definition 2.3.1, we used structural subtyping for interface types (which do not carry names).

```
    ...
    }


class B extends A {
    void add () {
        size = size + 1;
        super.add();
    }
    void add2 () {
        size = size + 2;
        super.add2();
}
```

It shows two classes A and B, implementing some container data structure, where the methods add and $add2$ add one, resp., two elements. This completely (if informally) describes the intended behavior of A's two shown methods. Class B in addition keeps information about the size of the container. The respective instance variable is accordingly updated in the overridden methods add and $remove$, which are assumed to behave identical to the methods of A and are therefore implemented using the $super$-keyword. The same is done for the $add2$-method, which increases the size by 2.

Now, the implementation of B shown in the figure is wrong, if the $add2$-method of the superclass A is implemented via $self$ using (for instance) twice the add-method. The real problem, however, is that *nothing* in the interface or the functional specification of A helps to avoid the problem!

The upshot of the simple example is that in the presence of sub-classing, overriding, and late binding, the *dependence* of the methods amongst each other is observable. Ultimately this seems to mean that also the self-communication is observable, basically that the "implementation" of a method in terms of *sequences* of self-calls is visible. This can be interpreted as "inheritance breaks encapsulation", since it exposes details of A to the environment which would normally be considered implementation details.

A similar phenomenon has been investigated in [136], albeit in an object-based setting with method update instead of a class-based setting with inheritance and method update.

6.2 Related work

Observable equivalence of programs is a natural and fundamental notion. It has been addressed from many angles, for various notions of observability, for different mathematical or semantical frameworks, and in particular for all sorts of language constructors. Consequently, the literature on observable semantics and full abstraction results is vast. Our choice of language features, motivated by languages like *Java* and $C^{\#}$, concentrated on object-oriented features such as classes as generators of fresh objects and concurrency in the form of multi-threading. So in the discussion of related work, besides mentioning a few "classical" results mainly for various λ-calculi and process calculi, we concentrate on object-oriented languages and calculi. Also we cover some results on languages with name-passing/name-generation facilities, notably the π-calculus, since name generation for allocating addresses for new objects plays a role in our semantics. See also the tutorial paper [114] for further discussion of obser-

vational equivalences for (sequential) calculi, especially for functional calculi involving state.

6.2.1 Observational semantics and full abstraction

The *contextual* definition of program equivalence —two programs are equivalent if, when put into all possible contexts, they behave the same— appeared first in the thesis [107] of Morris, for a call-by-name λ-calculus. Especially for parallel programs and process algebras, the notion of *testing* has been widely studied: The observer runs in parallel with the program under observation, typically interacting via message exchange, and reaching a defined point (witnessed by a predefined communication) is rated as success. In a non-deterministic setting and when comparing two processes wrt. their successful-ness confronted with all possible observers, one distinguishes necessary and potential success, leading to must, resp., may testing equivalence. The important notion of testing equivalence has been introduced by de Nicola and Hennessy [108] (see also [68]).

Sequential languages

The issue of full abstraction for programming language semantics started with sequential (and functional) languages [101][116]. Plotkin [116] investigated the semantics of a functional language, i.e., PCF ("programming language for computable functions"), an idealized typed functional language, basically a simply-typed λ-calculus with recursion and call-by-name evaluation. The influential paper indeed contains a negative result, namely that a standard denotational model, where the ground types are interpreted as flat cpos and arrow types are modeled as continuous functions, is sound, but fails to be fully abstract wrt. a standard notion of program equivalence (observational equivalence). Enriching the syntax by a "parallel" construct to the language mends the discrepancy between the denotational and the observational equivalence. See also [29] for a (slightly dated) survey of full abstraction results for sequential languages. At the same time, Milner [101] gave a fully abstract model of (a combinatory representation of) PCF without extending the language, but relying on equivalence classes of terms as basis of the model. The thesis of Stoughton [132] presents a language-independent theory of fully abstract denotational semantics of programming languages.

Parallelism and trace semantics

Testing equivalences, trace equivalences, or finer notions of equivalences and related full abstraction results have been extensively studied for numerous process calculi. For instance, full abstraction results for CSP, Hoare's "communicating sequential processes" [74], are presented in Roscoe's book [123], especially using various variations of the trace model, for instance for deterministic processes, or considering failures and divergence. Early results concerning trace semantics for CSP include [135] [110] see also [138] [33]. Concerning *shared variable* concurrency, Park [111] invented the "*transition trace*" semantics, a denotational semantics using traces, where a transition trace is a

finite sequence of *pairs* of states recording the potential interaction of the program with its environment (cf. also [122]). A *resumption* semantics for shared-variable concurrency is presented in [71]. The semantics, a denotational semantics using power domains, is compositional, but not fully abstract, at least not without extending the original language. The approach is extended to a fully abstract denotational semantics by [32], based on a transition trace semantics. Quite similar is the fully abstract trace semantics presented in [46] [75]. Various forms of traces have also been used as fully abstract semantics for dataflow networks, a computation model based on asynchronously communicating, parallel "agents", e.g., in [85] [87] [124] [90].

Object-oriented languages and calculi

Viswanathan [136] investigates the full abstraction problem in an object calculus with *subtyping*. The setting is a bit different from the one used here as he does not compare an contextual semantics with an denotational one, but a semantics by translation by a direct one. The paper considers neither concurrency nor aliasing. As source languages, functional object calculi with first-order types (including recursive types) are considered, more precisely $\mathbf{Ob}_{1\mu}$ and $\mathbf{Ob}_{1\leq:\mu}$ from [2], and encoded into a functional record calculus. The starting point is the observation that a straightforward encoding, Kamin's [86] so called self-application semantics fails to be fully abstract, it is too concrete; basically, the straightforward encoding exposes the dependency of a method body on the self-parameter to the observer.

Gordon et.al [63] investigate observational equivalence(s) in the setting of Abadi and Cardelli's (untyped and sequential) imperative object calculus $\text{imp}\varsigma$, featuring method update and cloning (but no classes). For the formalization of the operational semantics, i.e., not on the user level, locations in a global store are used. Three forms of operational semantics are used at the source level: A small-step semantics (based on a contextual definition and using an object store) and big-step substitution based semantics, and finally a big-step *closure* based semantics. The three equivalences at the source level operational semantic are compared with a lower-level semantics, given by an abstract machine.

Gordon and Rees [64] consider a Morris-style contextual equivalence for a (stateless) sequential object calculus with recursive types and subtyping (more precisely, $\mathbf{Ob}_{1\leq:\mu}$ from [2]). Starting from the standard definition of contextual equivalence (or "observational congruence") considering two programs as equivalent if not context can tell them apart observing their termination behavior on ground types, they characterize the contextual equivalence by a bisimulation relation.

[84] present a fully abstract trace semantics of a (single-threaded) core of a *Java*-like languages (*JavaJr*). The work in loc. cit. is similar to our framework; especially and unlike the formalism in [82], *JavaJr* features *classes* and instantiation of objects. Beside objects and classes, the language features a "third level" of program structure ("packages") which as in *Java* group together classes. For the may-testing based semantical framework, packages are taken as the unit of composition. The notion of package, however, differs package concept of *Java*. One difference is that packages in [84] are equipped with a package-interface, something not present in *Java*. Being basically a unordered lump of classes (and interfaces for instances of classes), the notion of package in *Java* is

indeed a rather crude abstraction and composition mechanism, offering little more than convenient way of addressing classes.[10] In the light of the work presented here, a crucial restriction is that instantiation *across package boundaries* is not possible. As a consequence, the resulting trace semantics does not have to deal with object connectivity, cliques, and their consequences, for instance swapping and replay. The resulting trace semantics, fully abstract wrt. may-testing, is very similar to the semantics in the object-based setting (cf. [82]) due to the absence of cross-border instantiation.

Smith [126] presents a fully abstract model for *Object-Z*, an object-oriented extension of the *Z* [128][118] specification language. The *complete-readiness* model, as it is called in [126], is closely related to the readiness model of Olderog and Hoare [110]. The observational starting point is less restrictive than ours: The environment is allowed to observe (possibly infinite) traces of events of the component within a context, not simply the fact whether a certain point is reached, as in the barbing set-up used here. So the starting point itself is already "almost" the fully abstract semantics. It turns out, however, that the plain trace semantics is not compositional and thus not fully abstract in the presence of non-determinism.[11] The intuitive reason that the plain trace model is unsound is, very abstractly, that the observer in the model has the power to "observe" whether a given operation (or method) is currently enabled.

A more general and abstract approach is investigated in [60], based on the categorical notion of sheaves, for concurrent, interacting objects. Sheaves as semantic basis of concurrency have been advocated in [61] and can be understood as a generalization of *traces.* Other work using sheaves as a model for concurrency and object-oriented systems is [106] and [51].

Similar in spirit to the work presented here, but with a different form of language in mind, [45] presents a fully abstract trace semantics for object-oriented programs, featuring instantiation and concurrency. Abstracting away from syntactical details, the pertinent differences to the work presented here are the following. (1) The concurrency model in not based on multithreading as here, but on "active objects" (cf. [22] for an early discussion of ways to combine concurrency and object-orientation). More concretely, the the behavior of (the instances of) a class is given in the form of state machines communicating via *synchronous operations*, i.e., the objects are bearer of activity and of state and the concurrency model resembles a message-passing process framework. Especially, there is no call-return discipline obey in the external interaction. Like here, the objects are instantiated across the component boundaries, which makes it necessary to consider the *connectivity* of the instances. (2) The semantical framework is simpler, however, wrt. instantiation: The presence of static class variables makes the number of instances of a class to be observable. Hence the problem of replay is absent (cf. also the discussion in Section 6.1.2).

Full abstraction in nominal calculi

Languages and calculi with the ability to dynamically create and communicate names have attracted much attention. The pioneering calculus in this context

[10] The "little more" refers to the fact that packages can be organized in *sub-packages,* where the membership of classes to (sub-)packages influences whether for instance class inheritance across package boundaries is allowed.

[11] The thesis uses the notion of contextual full abstraction, which implies compositionality.

is the π-calculus [103] [125] (the "calculus of mobile processes"), the standard process algebra for name passing and dynamically changing communication structures. Fiore, Moggi, and Sangiorgi [52] present a full abstraction result for the π-calculus. The extensional semantics draws heavily on techniques from domain and category theory (e.g., using functor categories), and using (strong, late) *bisimulation* equivalence as starting point, not may testing resp. traces as here. Apart from the categorical machinery, a key feature of the semantics is that the denotation of a process is given *relative* to the free names available to the outside. Indeed, as interface information, the *number* of maximally used free names is used. So this interface information corresponds to the commitment contexts Θ, except that here (as in [82]) actual names are exported modulo α-conversion, that the names carry a type here, and that the *exact* set of names is exported, not an upper bound. The reason seems to be, that here the names stand for objects (and threads and classes), and once created, objects never disappear, they are not destroyed and there is no garbage collection. [139] gives equational full abstraction for the standard translation of the polyadic π-calculus into the monadic one. Without additional information, the translation is not fully abstract, and [139] introduces graph-types as an extension to the π-calculus sorting to achieve full abstraction. The graph types abstracts the *dynamic* behavior of processes. In capturing the dynamic behavior of interaction, Yoshida's graph types are rather different from the graph abstracting the connectivity of objects presented here. Another fully abstract (filter) model for the π-calculus wrt. may testing is presented in [44]. A fully abstract (filter) model for mobile ambients [38] is investigated in [42], the higher-order case, where ambients themselves can be sent and received, is covered in [43].

[25] presents a fully abstract encoding of a π-calculus with terms (πT) into the more basic (polyadic) π-calculus (without native data), employing may-testing as the notion of observation. In contrast to [24], the translation is shown to be fully abstract, in particular the more concrete level cannot be used for more discriminating tests. The key to achieve full abstraction in the data encoding is to deviate from the standard π-calculus encoding trick, which represents data types as processes and the operations on the data by "interaction protocols". Such encodings are similar to the traditional Church encodings of data types in the λ-calculus, where data values are represented as functions. The problem with the "data-as-process" approach is that the process behaves as intended, i.e., as the value it represents, so long the client using the data adheres to the protocol, interacting with the channels as foreseen. An environment, however, which is not "playing according to the rules" can in general observe more, breaking full abstraction. In some sense, there is to much (and too liberal) interference with the data. The key trick of [25] is then not have the data behave like an interacting process, but centralize the access to the data via some "integrity manager", offering interference control. One "service" it offers is mutual exclusion (using mutex locks) for the data access. So the solution there is reminiscent to one crucial ingredient of our observer construction, namely the use of lock synchronization for concurrency control in the multi-threaded setting. In particular the centralized "broker" for maintaining the required data structures investigated in [66] and mentioned in Section 6.1.7 resembles the integrity manager of [25]. A difference is that in our object-oriented setting, we can use the data-storage facilities, the updateable fields, for basic data-representation. This seemed more straightforward than a "objects-as-

data"-representation, which would also seem possible.

Hennessy [69] gives a fully abstract semantics for higher-order CCS in the form of a path semantics, i.e., some form of trace semantics. Similarly, [70] contains a fully abstract set-theoretic denotational model for the π-calculus, for may- and must-testing. The model is based on functor categories. Those constructions are in particular used to give categorical meaning to the fact that the semantics of a process is given relative to an index-set of free names available in the process. The functor category uses a category of finitely many names and injections (renamings) as "source" category. The source category represents in particular the idea that the set of externally visible names is *dynamic* (due to the scoping mechanism of the ν-binders) and that the behavior is invariant under renaming.

Pitts and Stark [115, 130] combined name-generation and higher-order functions into the ν-calculus, an extension of a call-by-value simply-typed λ-calculus with a ground type for *names* and with the ability to create fresh names, which can be passed around and tested for equality. Unlike the treatment in object calculi (for instance here), name creation is not linked to object creation, but is a basic construct in its own right. The language itself is thus rather restricted, especially the calculus is not really imperative: Part of the semantics is a "heap" of created names, which grows larger during evaluation when new names are added, but the "references" cannot be destructively updated; on the other hand, as λ-calculus, of course, the ν-calculus features higher-order functions. [115] shows that observational equivalence coincides with applicative equivalence (defined using logical relations, expressing representation independence for the generated names) for terms of first-order types and that, under this restriction, ν-calculus is decidable for first-order terms.

Recently, contextual equivalences in the presence of *parametric polymorphism* have been investigated in the context of the π-calculus. Pierce and Sangiorgi [113] use barbed equivalence as notion of equivalence and discuss semantical issues considering a polymorphic discipline in connection with channel typings. In particular the "impure" nature of the calculus (aliasing and the possibility to compare names even when their type remains "abstract") is identified as a source of headaches. Jeffrey and Rathke [83] present a fully abstract model for the polymorphic π-calculus. A further study of polymorphism in the π-calculus is done in [27], including a fully abstract embedding of Girard and Reynold's polymorphic λ-calculus, also known as "System F", [58] [59] [120] into the process calculus.

Full abstraction and security

The name-generation facilities of the π-calculus have proved useful to provide a foundation of key-elements of cryptographic protocols. The prototypical calculus in this respect is Abadi and Gordon's spi-calculus [4], an extension of the π-calculus by primitives for encoding, decoding, and key generation.

The concept of (equational) full abstraction has also been proposed as useful criterion for assuring security properties and describing protection of (software) systems against attackers. The "observer", in that perspective, is the attacker or adversary which interacts with the program, e.g., the security protocol. in arbitrary ways. Abadi [1] investigates these issues, comparing *Java*-programs with their translation into byte code (translation correctness; the

same question it discussed in the paper for translating the π-calculus into the spi-calculus). Translational full abstraction means that the translation from a source to a target language preserves and reflects observational equivalence. When going from a higher-level, more abstract language to a lower-level, more detailed one may allow the observer to more more detailed observations, leading to security breaches ([1] [88] sketches the breach of equivalence by translation into byte code and resulting security threats for $C^{\#}$ and Microsoft's .NET-architecture.

Baldamus, Parrow, and Victor [24] deal with the same question in the context of translating the cryptographic spi-calculus into the more basic π-calculus, more precisely, the synchronous, untyped, polyadic π-calculus with late value passing semantics. Using may-tests as observations, the paper provides an encoding which preserves observational equivalence, *provided* the observer in the target language in a translation of a source language observer.

Malacaria and Hankin [94] apply a categorical, game-theoretic semantics as foundation for flow analysis, in particular for secure information flow. The set-up is an observational: The game consists of the opponent (the observer) and the player (the program). The analysis makes use of the fully abstract game semantics for *PCF*.

Boreale, de Nicola, and Pugliese in e.g., [30] use contextual equivalences, for the analysis of cryptographic protocols, formalized in the spi-calculus. They show full abstraction of some trace semantics wrt. may-testing (and additionally the paper relates weak bisimulation to barbed equivalence). In the cryptographic setting, compared to a framework without encryption like the π-calculus, the formalization of the external behavior is more complex as the knowledge of names is *protected* by the knowledge of keys, represented by names, as well. To keep track of that knowledge, the operational semantics is enriched by a "data base" of known keys and the transitions are restricted by the knowledge the environment has about names and in particular keys. This set-up is reminiscent of the *assumption contexts* used in the operational semantics here (and in [82]). Outgoing communication updates the environment knowledge, whereas incoming communication has to be checked against the assumptions (where newly created names also extend the environment knowledge). In particular, to do an input step, the environment assumptions must contain enough knowledge to produce the label, for instance by containing an expression proving that the names of the label can be obtained by using, encoding, and decoding the available information. No connectivity is involved, however.

Similar the results of Abadi, Fournet, and Gonthier: The paper [3], investigate under which circumstances a translation from a higher-level to a low-level language preserves *security properties*. Technically, the result is presented as full abstraction of the translation from the join-calculus (a member of the π-calculus inspired family of languages) [53] into the the sjoin-calculus, a lower-level join-calculus with cryptographic primitives.

Game semantics

Game theory recently has gained attraction as a fresh and unifying approach to compositional and fully abstract semantics. Some of the concepts of the game semantic approach (not very surprisingly) bear some general resemblance with

the set-up presented here. The *player* or more precisely the player's strategy, represents the program and the strategy of the *opponent* represents the environment or context. Often a strategy is represented as a set of sequences of moves, where a moves constitutes a basic interface interaction. Thus, the game semantics can be seen as some form of *trace* semantics, and a typical full abstraction result would state that two program fragments are observably equivalent if the corresponding sets of plays (i.e., "traces") are equivalent.

This general framework has been applied to a number of programming calculi, often variations and extensions of *PCF* or idealized Algol, considering features like a store, pointers, local-variables, procedures, and different evaluation mechanisms, like call-by-value vs. call-by-name evaluation. One pioneering contribution in this context is Hyland and Ong's fully abstract game semantics model for PCF [77]. [15] investigates a simply-typed call-by-value λ-calculus with higher-order store and ML-like references and present a fully abstract game semantics for observational equivalence. General references, i.e., references not just to data cells but also to functions, is a powerful language construct; in particular, one can straightforwardly encode the notions of objects, object references, and instantiation (cf. [15, Sect. 2.3]). The encoding, however, does not model classes, and so connectivity does not play a role here. Abramsky and McCusker provide a fully abstract game semantics for (full) idealized Algol [121] [109] in [17]. This result is simplified by Ghica and McCusker in [57], restricting the language to a finitary, recursion-free, second-order fragment. The fully abstract semantics is based on regular languages, where terms are interpreted as regular languages. The full abstraction results gives decidability of program equivalence for the considered fragment. A regular-language model for a similar call-by-value language (instead of call-by-name) is given in [56].

Also for nominal calculi, games semantics have been employed. Laird [92] presents a categorical game semantics for Pitt's and Stark's ν-calculus [115], a typed λ-calculus extended by names, respectively an imperative extension (also called $\lambda\nu$!-calculus), but the semantics is not fully abstract. Abramsky et.al. [14] present a fully abstract semantics based on *nominal* games for the ν-calculus. Neither work considers concurrency. More background on the ν-calculus can be found Stark's thesis [130] and [129].

An interesting recent contribution are *asynchronous* games [97] [99] [98]. The plays of a game in game semantics are typically characterized by a strict alternation of player and opponent moves and in this sense *sequential*. Melliès, in a series of papers, relaxes this constraint, introducing a notion of *concurrency* "into the arena". Strict alternation is abandoned, such that the player and the opponent can pursue more than one game each that the same time, to stay in the picture. Furthermore, drawing from rewriting theory, the models allow to *permute* independent moves, similar to the treatment of independence and concurrency in Mazurkiewicz traces [95][96]. To do that properly, i.e., to avoid confusion when permuting independent moves or labels, [97] (re-)introduces indexed threads to the plays (i.e., traces) of the games, similar as in the game semantics for PCF from [16]. See also the *concurrent* games in [19]. To represent the permutations on traces like swapping and switching in a game theoretic framework the techniques from the theory of asynchronous game look promising.

Concerning specifically object-oriented features, the PhD thesis of Burt [36]

presents a game-theoretical, denotational semantics for *FJS* ("Featherweight *Java* with store"), an extension of *FJ* (Featherweight *Java*) [81]. The language is sequential, but more true to *Java* than our setting in certain respect, especially wrt. typing issues, in that it features inheritance, subtyping (and even type casts). In that work, the semantics of *FJS* is given via some encoding into two lower level PCF-style languages featuring subtyping resp. subtyping and references ($\text{PCF}_{\leq}$ and $\text{REF}_{\leq}$). The translation is a variant of Kamin's [86] self-application semantics. For the lower level languages, Burt shows full abstraction for a game-theoretical denotational model. The lower level calculi $\text{PCF}_{\leq}$ and $\text{REF}_{\leq}$, however, for which the full abstraction results are shown, do not feature *classes* and *instantiation*, i.e., also there, the problem of connectivity does not show up.

Bibliography

[1] M. Abadi. Protection in programming-language translations. In J. Vitek and C. D. Jensen, editors, *Proceedings of MOS '98*, volume 1603 of *Lecture Notes in Computer Science*, pages 19–34. Springer-Verlag, 1999. An extended version appeared earlier as DEC research report 154, April 1998.

[2] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[3] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In LICS'98 [78], pages 105–116.

[4] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999. An extended abstract appeared in the *Proceedings of the Fourth ACM Conference on Computer and Communications Security (Zürich, April 1997)*. An extended version of this paper appears as Research Report 149, Digital Equipment Corporation Systems Research Center, January 1998, and, in preliminary form, as Technical Report 414, University of Cambridge Computer Laboratory, January 1997.

[5] E. Ábrahám. *An Assertional Proof System for Multithreaded Java — Theory and Tool Support*. PhD thesis, University of Leiden, 2004. Defended 20.1.2005.

[6] E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In Z. Li and K. Araki, editors, *ICTAC'04*, volume 3407 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, July 2004.

[7] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. Inductive proof-outlines for monitors in Java. In E. Najm, U. Nestmann, and P. Stevens, editors, *FMOODS '03*, volume 2884 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, Nov. 2003. A longer version appeared as technical report TR-ST-03-1, April 2003.

[8] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theoretical Computer Science*, 331, 2005.

[9] E. Ábrahám, A. Grüner, and M. Steffen. An open structural operational semantics for an object-oriented calculus with thread classes. Technical Report 0505, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, May 2005.

[10] E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. In R. Gorrieri and H. Wehrheim, editors, *FMOODS '06*, volume 4037 of *Lecture Notes in Computer Science*, pages 218–232. Springer-Verlag, 2006.

[11] E. Ábrahám, A. Grüner, and M. Steffen. Dynamic heap-abstraction for open, object-oriented systems with thread classes. Technical Report 0601, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Jan. 2006. A slightly shorter version has been submitted for inclusion into the Journal of Software and Systems Modeling (SoSym).

[12] E. Ábrahám, A. Grüner, and M. Steffen. Dynamic heap-abstraction for open, object-oriented systems with thread classes (extended abstract). In *Proceedings of Computability in Europe 2006: Logical Approaches to Computational Barriers, CiE'06*, Jan. 2006. Accepted for publication. A preliminary version has been included in the informal workshop proceedings of Cosmicah'05, as Queen Mary Technical Report RR-05-04, a longer version has been published as Technical Report 0601 of the Institute of Computer Science of the University Kiel, January 2006.

[13] S. Abramsky. Semantics of interaction: An introduction to game semantics. In *Semantics and Logics of Computation*, pages 1–32. Cambridge University Press, 1997.

[14] S. Abramsky, D. R. Ghica, A. Murawski, C. H. L. Ong, and I. D. B. Stark. Nominal games and full abstraction for the nu-calculus. In LICS'04 [80], pages 150–159.

[15] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In LICS'98 [78].

[16] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000. An earlier version appeared in the proceedings of TACS'94, LNCS 789.

[17] S. Abramsky and G. McCusker. Full abstraction for Idealized Algol with passive expressions. *Theoretical Computer Science*, 227(1–2):3–42, 1999.

[18] S. Abramsky and G. McCusker. Game semantics. In U. Berger and H. Schwichtenberg, editors, *Computational Logic*. Springer-Verlag, 1999.

[19] S. Abramsky and P.-A. Melliès. Concurrent games and full completeness. In LICS'99 [79].

[20] ACM. *23rd Annual Symposium on Principles of Programming Languages (POPL) (St. Petersburg Beach, Florida)*, Jan. 1996.

[21] A. Ahern and N. Yoshida. Formalizing Java RMI with explicit code mobility. In *Twentieth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '05*. ACM, 2005. In *SIGPLAN Notices*.

[22] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[23] P. America. Formal techniques for parallel object-oriented languages. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing 1991*, volume 612 of *Lecture Notes in Computer Science*, pages 119–140. Springer-Verlag, 1992.

[24] M. Baldamus, J. Parrow, and B. Victor. Translating spi calculus to π-calculus preserving may-tests. In LICS'04 [80], pages 22–31. An earlier and extended version appeared as Technical Report 2003-63, University Uppsala, Department of Information Technlogy.

[25] M. Baldamus, J. Parrow, and B. Victor. A fully-abstract encoding of the π-calculus with data terms (extended abstract). In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Proceedings of ICALP 2005*, volume 3580 of *Lecture Notes in Computer Science*, pages 1202–1213. Springer-Verlag, July 2005. An earlier and extended version appeared as Technical Report 2005-004, University Uppsala, Department of Information Technlogy.

[26] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[27] M. Berger, K. Honda, and N. Yoshida. Genericity and the π-calculus. In A. D. Gordon, editor, *Proceedings of FoSSaCS 2003*, volume 2620 of *Lecture Notes in Computer Science*. Springer-Verlag, Apr. 2002.

[28] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001.

[29] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 89–132. Cambridge University Press, 1985.

[30] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2002.

[31] P. Brinch Hansen. *Operating System Principles*. Prentice Hall, 1973.

[32] S. D. Brookes. Full abstraction for a shared variable parallel language. *Information and Computation*, 127(2):145–163, 1996. An earlier version appeared in the proceedings of LICS' 93, p.98–103. Also as reprint in [109].

[33] S. D. Brookes, A. W. Roscoe, and D. J. Walker. An operational semantics for CSP. Technical report, 1988.

[34] K. B. Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics*. MIT Press, Mar. 2002.

[35] T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 2 edition, 1997.

[36] M. T. Burt. *Games, Call-By-Value, and Featherweight Java*. PhD thesis, University London, Imperial College, 2004.

[37] L. Cardelli. Object-based vs. class-based language. PLDI'96 Tutorial, 1996.

[38] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98* (LNCS1378): 140–155.

[39] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[40] D. Caromel and L. Henrio. *A Theory of Distributed Objects. Asynchrony — Mobility — Groups — Components*. Springer-Verlag, 2005.

[41] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Seventeenth Annual Symposium on Principles of Programming Languages (POPL) (San Fancisco, CA)*, pages 125–135. ACM, January 1990. Also in the collection [67].

[42] M. Coppo and M. Dezani-Ciancaglini. A fully-abstract model for mobile ambients. *Electronic Notes in Theoretical Computer Science*, 62, 2001. Proceedings of TOSCA'01.

[43] M. Coppo and M. Dezani-Ciancaglini. A fully-abstract model for higher-order mobile ambients. In A. Cortesi, editor, *Proceedings of the Third International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI) 2002*, volume 2294 of *Lecture Notes in Computer Science*, pages 255–271. Springer-Verlag, 2002.

[44] F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. A filter model for mobile processes. *Mathematical Structures in Computer Science*, 9(1):63–101, 1999.

[45] F. S. de Boer, M. M. Bonsangue, M. Steffen, and E. Ábrahám. A fully abstract trace semantics for UML components. In M. Bosangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors, *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[46] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. The failure of failures: Towards a paradigm for asynchronous communication. In J. C. M. Baeten and J. F. Groote, editors, *Proceedings of CONCUR '91*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991. A longer version appeared as Technical Report RUU-CS-90-40, December 1990, Utrecht University.

[47] W.-P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods*. Cambridge University Press, 2001.

[48] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, 1965. Reprinted in [55].

[49] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[50] ECMA International Standardizing Information and Communication Systems. *C# Language Specification*, 2nd edition, Dec. 2002. Standard ECMA-334.

[51] H.-D. Ehrich, J. Goguen, and A. Sernadas. A categorial theory of objects as observed processes. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages (REX Workshop)*, volume 489 of *Lecture Notes in Computer Science*, pages 203–228. Springer-Verlag, 1991.

[52] M. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus (extended abstract). In *Proceedings of LICS '96*, pages 43–54. IEEE, Computer Society Press, July 1996.

[53] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In POPL'96 [20], pages 372–385.

[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[55] F. Genyus. *Programming Languages*. Academic Press, 1968.

[56] D. R. Ghica. Regular language semantics for a call-by-value programming language. In *17th Conference on the Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science, pages 85–98, 2000.

[57] D. R. Ghica and G. McCusker. The regular-language semantics of second-order idealized Algol. *Theoretical Computer Science*, 2003. Submitted.

[58] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Second Scandinavian Logic Symposium '71 (Oslo, Norway)*, number 63 in Studies in Logic and the Foundations of Mathematics, pages 63–92. North-Holland, 1971.

[59] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupure dans l'arithmetique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[60] J. Goguen. Sheaf semantics for concurrent interacting objects. Tutorial-Manuscript, June 1990. REX School: Foundations of Object-Oriented Programming.

[61] J. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1:49–67, 1991.

[62] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[63] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In S. Ramesh and G. Sivakumar, editors, *Proceedings of FSTTCS '97*, volume 1346 of *Lecture Notes in Computer Science*, pages 74–87. Springer-Verlag, Dec. 1997. Full version available as Technical Report 429, University of Cambridge Computer Laboratory, June 1997.

[64] A. D. Gordon and G. D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In POPL'96 [20], pages 386–395. Full version available as Technical Report 386, Computer Laboratory, University of Cambridge, January 1996.

[65] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Second edition, 2000.

[66] A. Grüner. Cliques and components: Implementing traces and object-connectivity for a concurrent language. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, July 2004.

[67] C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming, Types, Semantics, and Language Design*. Foundations of Computing Series. MIT Press, 1994.

[68] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[69] M. Hennessy. A fully abstract denotational model for higher-order processes. *Information and Computation*, 112(1):55–95, 1994. An extended abstract appeared in *Proceedings of LICS '93*, pages 397–408.

[70] M. Hennessy. A fully abstract denotational model for the π-calculus. *Theoretical Computer Science*, 282(2):53–89, 2001. An earlier version appeared as CogSci Report cs1996:04 of the University of Sussex, Brighton, June 1996.

[71] M. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In J. Bečvář, editor, *Eighth Mathematical Foundations of Computer Science (Olomouc, Czechoslovakia)*, volume 74 of *Lecture Notes in Computer Science*, pages 95–103. Springer-Verlag, 1979.

[72] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[73] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[74] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[75] E. Horita, J. W. de Bakker, and J. J. M. M. Rutten. Fully abstract denotational models for nonuniform concurrent languages. *Information and Computation*, 115(1):125–178, 15 Nov. 1994. Also as Technical Report CS-R9027, CWI Amsterdam.

[76] J. M. E. Hyland. Game semantics. In *Semantics and Logics of Computation*. Cambridge University Press, 1997.

[77] J. M. E. Hyland and C. H. L. Ong. On full-abstraction for PCF: I. Models, observables, and the full abstraction problem. II. Dialogue games and innocent strategies. III. A fully abstract and universal game model. *Information and Computation*, 163:285–408, 2000.

[78] IEEE. *Thirteenth Annual Symposium on Logic in Computer Science (LICS) (Indiana)*. Computer Society Press, July 1998.

[79] IEEE. *Forteenth Annual Symposium on Logic in Computer Science (LICS) (Trento, Italy)*. Computer Society Press, July 1999.

[80] IEEE. *Nineteenth Annual Symposium on Logic in Computer Science (LICS) (Turku, Finland)*. Computer Society Press, July 2004.

[81] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '99*, pages 132–146. ACM, 1999. In *SIGPLAN Notices*.

[82] A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*. IEEE, Computer Society Press, July 2002.

[83] A. Jeffrey and J. Rathke. Full abstraction for polymorphic Pi-calculus. In V. Sassone, editor, *Proceedings of FoSSaCS '05*, volume 3441 of *Lecture Notes in Computer Science*, pages 266–281. Springer-Verlag, 2005.

[84] A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In M. Sagiv, editor, *Proceedings of ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.

[85] B. Jonsson. A fully abstract trace model for dataflow networks. In *Proceedings of POPL '89*, pages 155–165. ACM, January 1989.

[86] S. Kamin. Inheritance in Smalltalk-80: A denotational definition. In *ACM Conference on Programming Language Design and Implementation (Atlanta, GA)*. ACM, June 1988. In *SIGPLAN Notices* 23(7).

[87] R. M. Keller and P. Panangaden. Semantics of digital networks containing indeterminate operators. *Distributed Computing*, 1(4):235–245, 1986.

[88] A. Kennedy. Securing the .NET programming model (industrial application). In *Proccedings of the APPSEM II Workshop (Industrial application session), Frauenchiemsee, Lake Chiemsee, Munich, Germany, September 12-15, 2005*, 2005.

[89] D. E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, May 1966.

[90] J. N. Kok. A fully abstract semantics for data flow nets. In *Proc. PARLE, II, LNCS 259*, pages 351–368. Springer-Verlag, 1987.

[91] J. D. Laird. *A Semantic Analysis of Control*. PhD thesis, University of Edinburgh, 1997.

[92] J. D. Laird. A game semantics of local names and good variables. In *Proceedings of FoSSaCS 2004*, volume 2987 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[93] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*, volume 2. Addison-Wesley, 1999.

[94] P. Malacaria and C. L. Hankin. Non-deterministic games and program analysis: An application to security (extended abstract). In LICS'99 [79], pages 443–542.

[95] A. Mazurkiewicz. Concurrent program schemas and their interpretation. In *Proc. Aarhus Workshop on Verification of Parallel Programs*, 1977.

[96] A. Mazurkiewicz. Introduction to trace theory. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*. World Scientific, Singapore, 1995.

[97] P.-A. Melliès. Asynchronous games 1: A group-theoretic formulation of uniformity. Manuscript, Available online, 2003.

[98] P.-A. Melliès. Asynchronous games 3: An innocent model of linear logic. In L. Birkedal, editor, *Proceedings of the 10th Conference on Category Theory and Computer Science, Copenhagen, 2004. CTCS 2004*, volume 122 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2004.

[99] P.-A. Melliès. Asynchronous games 2: The true concurrency of innocence. *Theoretical Computer Science*, 2006. Submitted to a special issue for selected papers of CONCUR 04. An extended abstract is published in the Proceedings of the 15th International Conference on Concurrency Theory, London, 2004. LNCS 3170, pages 448–465.

[100] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–354. Springer-Verlag, 1998. A longer version has been published as Turku Centre of Computer Science Technical Report TUCS Nr. 117, June 1997 under the title "The Fragile Base Class Problem and Its Solution".

[101] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[102] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[103] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.

[104] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[105] J. C. Mitchell. *Foundations of Programming Languages*. Foundation of Computing Series. MIT Press, 1996.

[106] L. Monteiro and F. Pereira. A sheaf-theoretic model of concurrency. Technical Report CSLI-86-62, Center for the Study of Languages and Information, Stanford University, Oct. 1986.

[107] J. H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[108] R. D. Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[109] P. W. O'Hearn and R. D. Tennent, editors. *Algol-like Languages, Progress in Theoretical Computer Science*. Birkhäuser, two volumes, 1997.

[110] E.-R. Olderog and C. A. R. Hoare. Specification-oriented semantics of communicating processes. *Acta Informatica*, 23(1):9–66, 1986. A preliminary version appeared under the same title in the proceedings of the 10th ICALP 1983, volume 154 of LNCS.

[111] D. M. R. Park. On the semantics of fair parallelism. In D. Bjørner, editor, *Abstract Software Specifications (1979 Copenhagen Winter School)*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526. Springer-Verlag, 1980.

[112] B. C. Pierce. *Types and Programming Languages*. MIT Press, Feb. 2002.

[113] B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–584, 2000. An extended abstract appeared in *Proceedings of POPL '97*: 241–255. Full version available as University of Pennsylvania Technical Report MS-CIS-99-10.

[114] A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *APPSEM 2000*, volume 2395, pages 378–412, 2002.

[115] A. M. Pitts and I. D. B. Stark. Observable properties of higher-order functions that dynamically create local names, or: What's new. In A. M. Borzyszkowski and S. Sokołowski, editors, *Proceedings of MFCS '93*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Sept. 1993.

[116] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[117] G. D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

[118] B. F. Potter, J. E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Series in Computer Science. Prentice Hall, 1990.

[119] M. Raynal. *Algorithms for Mutual Exclusion*. Scientific Computation Series. MIT Press, Cambridge, Massachusetts, 1986.

[120] J. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la programmation (Paris, France)*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[121] J. C. Reynolds. The essence of Agol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium in Algorithmic Languages*, pages 345–372. North Holland, 1981. As reprint in [109].

[122] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[123] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[124] J. R. Russell. Full abstraction for nondeterministic dataflow networks. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 170–177. IEEE Press, 1989. An extended version appeared as Techical Report TR 89-1022, Department of Computer Science, Cornell University.

[125] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[126] G. P. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, Oct. 1992.

[127] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '86 (Portland, Oregon)*, pages 38–45. ACM, 1986. In *SIGPLAN Notices* 21(11).

[128] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 1989.

[129] I. D. B. Stark. Categorical models for local names. *LISP and Symbolic Computation*, 9(1):77–107, 1994.

[130] I. D. B. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, Dec. 1994.

[131] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '95*, pages 200–214. ACM, 1995. In *SIGPLAN Notices*) 30(10).

[132] A. Stoughton. Fully abstract models of programming languages. *Research Notes in Theoretical Computer Science*, 1988. A revision with additions of the University of Edinburgh PhD thesis, Technical Report CST-40-86.

[133] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[134] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, Sept. 1996.

[135] A. Valmari. The weakest deadlock-preserving congruence. *Information Processing Letters*, 53:341–346, 1995.

[136] R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In LICS'98 [78].

[137] W. von Dyck. Gruppentheoretische Studien. *Mathematische Annalen*, 30:1–44, 1882.

[138] D. J. Walker. An operational semantics for CSP. M.sc. thesis, Oxford University, 1986.

[139] N. Yoshida. Graph types for monadic mobile processes. In V. Chandru and V. Vinay, editors, *Proceedings of FSTTCS '96*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer-Verlag, 1996. Full version as Technical Report ECS-LFCS-96-350, University of Edinburgh.

Part IV

Proofs

APPENDIX A

Sequential

This chapter collects the proofs as well as some additional lemmas left out from the main body of the work. It follows in structure the main part.

A.1 Operational semantics

First some straightforward invariants of the operational semantics. Many carry over to the multithreaded case, where we will not prove them again.

The following standard lemma states that well-typedness of a component is preserved under reduction. This property is also known as *subject reduction.* It will be used later tacitly at various places.

Lemma A.1.1 (Subject reduction)**.** *Assume* $\Xi \vdash C$.

1. (a) *If* $C \rightsquigarrow C'$, *then* $\Xi \vdash C'$.
   (b) *If* $C \xrightarrow{\tau} C'$, *then* $\Xi \vdash C'$.
   (c) *If* $C \equiv C'$, *then* $\Xi \vdash C'$.
2. *If* $\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}$, *then* $\acute{\Xi} \vdash \acute{C}$.

Proof. All parts by induction on the length of derivation for the corresponding reduction step. The judgment asserts well-typedeness and connectivity; we treat both parts separately and show a few cases for each.

Case: RED: $\natural\langle \mathit{let}\ x{:}T = v\ \mathit{in}\ t\rangle \rightsquigarrow \natural\langle t[v/x]\rangle$
Since $\Xi \vdash C$, we know $\Delta' \vdash \natural\langle \mathit{let}\ x{:}T = v\ \mathit{in}\ t\rangle : ()$ for some name context Δ' which furthermore implies $;\Delta' \vdash v : T$ and $x{:}T;\Delta' \vdash t : \mathit{none}$ (inverting T-THREAD, T-LET, and T-NAME, using the fact that v can have at most one type. Note that the value v can only be an object reference o).[1] Hence by a (standard) substitution lemma, $;\Delta' \vdash t[v/x] : \mathit{none}$, which entails $\acute{\Xi} \vdash \acute{C}$, as required.

Case: NEWO$_i$: $c[\![F, M]\!] \parallel \natural\langle \mathit{let}\ x{:}c = \mathit{new}\ c\ \mathit{in}\ t\rangle \rightsquigarrow c[\![F, M]\!] \parallel \nu(o{:}c)(o[c, F] \parallel \natural\langle \mathit{let}\ x{:}c = o\ \mathit{in}\ t\rangle)$
Well-typedness of the pre-configuration entails $\Delta_1 \vdash c[\![F, M]\!] : (c{:}T_c)$ for some context Δ_1 and type $T_c = [\![O]\!]$, and furthermore $\Delta_1, c{:}T_c \vdash \natural\langle \mathit{let}\ x{:}c = \mathit{new}\ c\ \mathit{in}\ t\rangle : ()$. The latter judgment (inverting rules T-THREAD, T-LET, and T-NEWC) gives $x{:}c; \Delta_1, c{:}T_c \vdash t : \mathit{none}$. By weakening, thus (1) $x{:}c; \Delta_1, o{:}c, c{:}T_c \vdash \natural\langle t\rangle : \mathit{none}$. For the named object: $\Delta_1 \vdash c[\![F, M]\!] : (c{:}T_c)$ entails (2) $\Delta_1, o{:}c \vdash o[\![c, F]\!] : (o{:}c)$. Using the typing derivation for the pre-configuration, the post-configuration can thus be justified with T-PAR, T-NU$_i$, and using (1) and (2).

The remaining rules of Table 2.5 work similarly.

The proof for part 1c, that the structural congruence from Table 2.6 preserves well-typedness, is straightforward, using a weakening, resp., a strengthening property for typing in the case $C_1 \parallel \nu(n{:}T).(C_2) \equiv \nu(n{:}T).(C_1 \parallel C_2)$, where n does not occur free in C_1.

Case: CALLI$_0$ with $a = \nu(\Phi')\langle \mathit{call}\ o_r.l(\vec{v})\rangle$?
We need basically to argue that $C(\Theta') \parallel \natural\langle \mathit{let}\ x{:}T = o_r.l(\vec{v})\ \mathit{in}\ o_r\ \mathit{returns}\ x\ \mathit{to}\ \odot\rangle$ is well-typed in the given evaluation context. The $C(\Theta')$ is defined as $o_1[c_1, F_1] \parallel \ldots \parallel o_k[c_k, F_k]$, where $\Theta' = \vec{o}{:}\vec{c}$ are the bindings for the lazily instantiated objects, i.e., $\Theta \vdash c_i : T_i$ for all c_i.

The typing part of $\acute{\Xi}$ in the premise is given by $\acute{\Theta} = \Theta + \Theta'$ and $\acute{\Delta} = \Delta + \Delta'$ (Definition 2.6.8). Thus, each $o_i[c_i, F_i]$ from $C(\Theta')$ is well-typed (with

[1] In the multithreaded setting, besides a reference to an object, the value can also be a reference to a thread.

type c_i). Expanding the typing part of the premise $\acute{\Xi} \vdash o_r \overset{\lfloor a \rfloor}{\leftarrow} \odot$, we are given $;\acute{\Theta} \vdash o_r{:}c_r$, furthermore $;\acute{\Delta},\acute{\Theta} \vdash c_r : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]$, and $;\acute{\Delta},\acute{\Theta} \vdash \vec{v} : \vec{T}$ (cf. LT-CALLI from Table 2.10). In particular, the declared type T of x coincides with the return type of method l. Well-typedness of $\natural\langle \mathit{let}\ x{:}T = o_r.l(\vec{v})\ \mathit{in}\ o_r\ \mathit{returns}\ x\ \mathit{to}\ \odot \rangle$ is justified by T-THREAD, T-CALL, and T-RETURN, and the mentioned premises.

Case: CALLO with $a = \nu(\Phi')\langle \mathit{call}\ o_r.l(\vec{v})\rangle$!
By well-typedness of the preconfiguration, (inverting in particular T-LET), $;\Delta_1 \vdash o_r.l(\vec{v}) : T$ and $x{:}T;\Delta_1 \vdash t : \mathit{none}$, and the result follows using T-BLOCK.

For connectivity, proceed similarly by induction on the steps. As $\rightsquigarrow$-steps do not access the instance state and affect only the top-most stack frame, the induction step is immediate. Note that internal reduction steps affect only the top-most stack frame and that no information is passed to deeper stack frames (or from the stack of one thread n to another thread n' in the multithreaded case), especially not by the substitution in rule RED. □

Lemma A.1.2 (Static nature of class names)**.** *If* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$*, then for all class names* c*,* $\Delta \vdash c$ *iff* $\acute{\Delta} \vdash c$ *and likewise* $\Theta \vdash c$ *iff.* $\Theta \vdash c$*.*

Proof. Obvious. Class names cannot be sent around; hence they never occur in a communication label. □

Lemma A.1.3 (Invariants)**.** *Assume* $\Xi_0 \vdash C \overset{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$*. Then:*

1. $\acute{E}_\Delta \subseteq \acute{\Delta} \times (\acute{\Delta} + \acute{\Theta})$ *and* $\acute{E}_\Theta \subseteq \acute{\Theta} \times (\acute{\Theta} + \acute{\Delta})$*.*
2. $\mathit{dom}(\acute{\Delta}) \cap \mathit{dom}(\acute{\Theta}) = \emptyset$*, for all object and class references.*

Proof. Straightforward by inspection on the rules for external steps from Table 2.11; internal steps obviously preserve the properties.

For part 2: By induction on the length of reduction. Internal steps and the rules for structural congruence leave the contexts untouched. The external steps from Table 2.11 add a fresh *object* name only to *either* Δ or to Θ, and the freshness assumption assures that the new name does not occur on both contexts. Class names are never exchanged boundedly (cf. Lemma A.1.2). □

In the multithreaded case later, there will be a third category of names besides class and object names, namely names for the threads (see Lemma C.1.1.

We call a well-typed component $\Delta \vdash C : \Theta$ is *instance closed,* if for all identifiers o with $;\Theta \vdash o : c$, also $;\Theta \vdash c : T$. In other words, each object identifier typeable in Θ and thus occurring free in the component C, is an instance of a class also typeable in Θ. Note that the type system assures that T is a class type, i.e., $T = [\![T']\!]$. For example, $\vdash o[c, F] \parallel c[\![O]\!] : o{:}c, c{:}[\![T']\!]$ is instance closed, but the component containing the object o in isolation is not. Instance closedness is preserved under reduction.

Lemma A.1.4 (Preservation of instance closedness)**.** *Assume* $\Xi \vdash C \overset{a}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$*. If* $\Xi \vdash C$ *is instance closed, then so is* $\acute{\Xi} \vdash \acute{C}$*.*

Proof. Straightforward. Internal steps do not change the contexts nor do they change the externally visible object names (or externally visible thread names in the multithreaded setting). The same holds for the structural rules (apart

from renaming). The external rules maintain instance closedness by distinguishing in the exchange of bound (object) references according to the class, as indicated by writing $\nu(\Delta', \Theta').\ \langle call\ o_r.l(\vec{v})\rangle?$, resp., $\nu(\Delta', \Theta').\ \langle return(v)\rangle?$ for labels of incoming communication, where by convention Δ' refers to references to external objects and threads whose scope is extruded in the step, and Θ' to references to component objects and threads. In the case of object references in Θ, this realizes lazy instantiation. For outgoing communication, the situation is dual. Remember that the assumption and commitment contexts are disjoint as far as class names are concerned (Lemma A.1.3(2)). □

In the whole development we will always assume that well-typed components are instance closed.

Proof of Lemma 2.6.15 on page 42 (no surprise). By definition of the incoming steps from Table 2.11, using the context update from Definition 2.6.8 and 2.6.9. □

A.2 Traces and equivalences

This section contains material about traces. Some of the definitions are used later in the characterization of the legal traces. Section A.2.1 collects a number of properties in connection with the parenthetic nature of the calls and returns in a trace (of one given thread). After formalizing predicates for *balance* (each call must be answered by a matching return) and *weak* balance, characterizing prefixes of balanced traces, we define the sender and receiver of a communication, given the past interaction, and prove properties about enableness of communication after a trace.

Sections A.2.2 and A.2.3 give equational characterizations of the swapping and replay relations relation $\asymp_\Theta$ and $\preccurlyeq_\Theta$, which were introduced in Section 3.1 using the notion of projection (cf. Definition 3.1.7 and 3.1.8).

A.2.1 Balance conditions

Lemma A.2.1 (Balance and alternation)**.** *If $\vdash t : wbalanced$, then t is alternating. For non-empty t: If $\vdash t : wbalanced^+$, then $t = t'\ \gamma?$. Dually for $wbalanced^-$. A fortiori, the same property holds for strictly balanced traces.*

Proof. By straightforward induction on the rules of Table 3.3 and 3.4. □

The next property shows that one can remove balanced subsequences from a weakly balanced trace without destroying weak balance. The reverse property —balanced subsequences can be added, as well— is covered in Lemma A.2.4.

Lemma A.2.2 (Removal of balanced parts)**.** *If $\vdash s_1\ t\ s_2 : {}^{p_1}wbalanced^{p_2}$ and $\vdash t : balanced$, then $\vdash s_1\ s_2 : {}^{p_1}wbalanced^{p_2}$.*

Proof. For the proof prove in addition to the property of the lemma (part 3 below), two simpler properties:

1. If $\vdash s_1\ \gamma_c?\ \gamma_r!\ s_2 : balanced^p$, then $\vdash s_1\ s_2 : balanced^p$.
2. If $\vdash s_1\ \gamma_c?\ \gamma_r!\ s_2 : {}^{p_1}wbalanced^{p_2}$, then $\vdash s_1\ s_2 : {}^{p_1}wbalanced^{p_2}$.

3. If $\vdash s_1\ t\ s_2 : {}^{p_1} wbalanced^{p_2}$ and $\vdash t : balanced$, then $\vdash s_1\ s_2 : {}^{p_1} wbalanced^{p_2}$.

Part 1 by induction on the rules from Table 3.3, using for the cases B-II and B-OO the observation (a) that non-empty balanced traces start with a call and end with a return. Part 2 by induction on the rules of Table 3.4, using part 1. The observation (a) again assures that we can proceed by induction in the cases for WB_1 and WB_2.

For part 3, proceed by induction on the balance derivation for t. For one case, assume $\vdash t : balanced^+$. The case of B-EMPTY$^+$ is immediate. For B-II, we are given $t = t_1\ t_2$ with $\vdash t_1 : balanced^+$ and $\vdash t_2 : balanced^+$, where t_1 and t_2 are strictly shorter than t. Hence the case follows by using twice the induction hypothesis. For case B-IO, $t = \gamma_c?\ t'\ \gamma_r!$ with $\vdash t' : balanced^+$ by a subderivation. By part 2, $\vdash s_1\ \gamma_c?\ \gamma_r!\ s_2 : {}^{p_1} wbalanced^{p_2}$, and hence by induction, $\vdash s_1\ \gamma_c?\ t'\ \gamma_r!\ s_2 : {}^{p_1} wbalanced^{p_2}$, as required. □

Concatenation preserves weak balance, provided that the two traces fit together in the sense that alternation is respected. Also a balanced sub-sequence can be inserted without destroying weak balance.

Lemma A.2.3 (Balance and insertion)**.** *Assume* $\vdash s_1\ s_2 : balanced^{p_1}$ *and furthermore* $\vdash s_1 : {}^{p_1} wbalanced^{p_2}$*, and* $\vdash t : balanced^{p_2}$*, then* $\vdash s_1\ t\ s_2 : balanced^{p_1}$.

Proof. Proceed by induction on the derivation of $\vdash s_1\ s_2 : balanced^{p_1}$. In case of B-EMPTY$^+$ or B-EMPTY$^-$, the result is trivial. For B-II, we have that $p_1 = +$. We distinguish according to the way, the rule B-II splits $s_1 s_2 = s$. If $s = s_1^1\ s_1^2\ s_2$ with $\vdash s_1^1 : balanced^+$ and $\vdash s_1^2\ s_2 : balanced^+$ as subgoals of B-II, we get by induction $\vdash s_1^2\ t\ s_2 : balanced^+$, and the result follows with B-II. The case where s_2 is split, i.e., where $s = s_1\ s_2^1\ s_2^2$ and with subgoals $\vdash s_1\ s_2^1 : balanced^+$ and $\vdash s_2^2 : balanced^+$, works analogously. Finally, if B-II has $\vdash s_1 : balanced^+$ and $\vdash s_2 : balanced^+$ as subgoals, the result follows directly using twice B-II. The case for B-OO works analogously.

For B-IO, $s = s_1\ s_2 = \gamma_c?\ s_1'\ s_2'\ \gamma_r!$, with $\vdash s_1'\ s_2' : balanced^-$ as subderivation, and the case follows by induction and B-IO. The case for B-OI works analogously. □

Lemma A.2.4 (Weak balance, concatenation, and insertion)**.**

1. *The following two rules for weak balance are admissible:*

$$\frac{\vdash s : {}^{p_1} wbalanced^{p_2} \qquad \vdash t : {}^{p_2} wbalanced^{p_3}}{\vdash s\ t : {}^{p_1} wbalanced^{p_3}}\ \text{WB-CONC}$$

$$\frac{\vdash s_1\ s_2 : {}^{p_1} wbalanced^{p_3} \quad \vdash s_1 : {}^{p_1} wbalanced^{p_2} \quad \vdash t : balanced^{p_2}}{\vdash s_1\ t\ s_2 : {}^{p_1} wbalanced^{p_3}}\ \text{WB-INSERT}$$

2. *Assume* $\vdash s : wbalanced$ *and* $\vdash t : wbalanced$*. If* $s\ t$ *is alternating, then* $\vdash s\ t : wbalanced$.

Proof. For part 1, we start with WB-CONC (as the admissibility of WB-INSERT uses WB-CONC): Assume that $s \neq \epsilon$ and $t \neq \epsilon$ (the result is trivial then) and

proceed by induction on the derivation of the first sub-sequence s. If s is balanced (rule WB-B), the case follows by WB_1. The case for WB_1 follows by induction, and using WB_1. In the case for WB_2, we have that $s = s_2\ s_3$, with $\vdash s_2 : {}^{p_1}\mathit{wbalanced}^{p_2}$ and $\vdash s_3 : \mathit{balanced}^{p_2}$. By induction we get $\vdash s_3\ t : {}^{p_1}\mathit{wbalanced}^{p_3}$, and again by induction, $\vdash s_2\ s_3\ t : {}^{p_1}\mathit{wbalanced}^{p_3}$, as required.

Concerning rule WB-INSERT. Assume that s_1, s_2, and t are not empty, otherwise the argument is immediate: If $t = \epsilon$, the result is trivial. If $s_1 = \epsilon$ (and neither t nor s_2 empty), then WB_1 yields the result, observing that $p_2 = p_1$. For $s_2 = \epsilon$, the argument is analogous, using WB_2 instead, and observing that $p_2 = p_3$.

Otherwise, proceed by induction on the length of derivation for the judgment $\vdash s_1 : {}^{p_1}\mathit{wbalanced}^{p_2}$ from Table 3.4. The case for WB-B is immediate (using B-II or B-OO, depending on the polarity). For WB_1, the result follows by straightforward induction and WB_1. For WB_2, we are given $s_1 = s_1'\ s_1''$ such that $\vdash s_1' : {}^{p_1}\mathit{wbalanced}^{p_2}$ and $\vdash s_1'' : \mathit{balanced}^{p_2}$. By rule B-II or B-OO of Table 3.3, $\vdash s_1''\ t : \mathit{balanced}^{p_2}$. By removal of balanced parts from Lemma A.2.2, the premise of WB-INSERT $\vdash s_1'\ s_1''\ s_2 : {}^{p_1}\mathit{wbalanced}^{p_3}$ implies $\vdash s_1'\ s_2 : {}^{p_1}\mathit{wbalanced}^{p_3}$ (since s_1'' is balanced). Hence we get by induction, $s_1'\ (s_1''\ t)\ s_2 : {}^{p_1}\mathit{wbalanced}^{p_3}$, as required.

For WB-CALL$^+$, we are given $s_1 = \gamma_c'?\ s_1'$ with $\vdash s_1' : {}^{+}\mathit{wbalanced}^{p_2}$ (and p_1 must equal $-$). By concatenation using rule WB-CONC, $\vdash s_1'\ s_2 : {}^{+}\mathit{wbalanced}^{p_3}$. Hence by induction, $\vdash s_1'\ t\ s_2\ {}^{+}\mathit{wbalanced}^{p_3}$, whence $\vdash \gamma_c'?\ s_1'\ t\ s_2\ {}^{-}\mathit{wbalanced}^{p_3}$ follows with WB-CALL$^-$, as required. The case for WB-CALL$^-$ works analogously.

Part 2 is a straightforward consequence. □

The next lemma establishes the mentioned intuition of the weak balance condition, namely that a weakly balanced trace is a prefix of a balanced one. Of course, a weakly balanced one can be completed not to just one single balanced trace, but to infinitely many, since balanced parts can be injected at will (so long alternation is preserved) in the prolongation. Given a weakly balanced trace r, there is, however, a minimal, canonical balanced trace t with $r \preccurlyeq t$, which is the one, where all unanswered calls are just completed by the responding return, i.e., where $r\ s = t$, where s contains only calls. We do not need this property, so Lemma A.2.5 simply states that there is *some* balanced trace which completes the weakly balanced one.

Lemma A.2.5 (Balance and weak balance)**.** *Given a trace r. Then $\vdash r : \mathit{wbalanced}$ iff $r \preccurlyeq t$ for some t such that $\vdash t : \mathit{balanced}$.*

Proof. There are two directions to show.

Case: "if"

Assume $r \preccurlyeq t$, i.e., $r\ s = t$ for some s, and $\vdash t : \mathit{balanced}$. First, for any t of the form $t_1\ t_2\ t_3$ we have by Lemma A.2.2: If $\vdash t : \mathit{balanced}^-$ and $\vdash t_2 : \mathit{balanced}$, then $\vdash t_1\ t_3 : \mathit{balanced}^-$. Let r' be defined as r with all balanced subsequences removed. In analogous way, s' is obtained from s. Clearly, r' contains only calls, and s' only returns. Assuming otherwise contradicts the fact r' or s' do not contain balanced subsequences.

For one case, assume $\vdash t : \mathit{balanced}^-$. By the above observation, $\vdash r'\ s' : \mathit{balanced}^-$, where r' contains only calls. Therefore, $\vdash r' : \mathit{wbalanced}^-$, resp.,

$\vdash r' : wbalanced^+$. This implies with the rules from Table 3.4, that also $\vdash r : wbalanced^-$, resp., $\vdash r : wbalanced^+$, as required.

Case: "only if"
By induction on the derivation from Table 3.4, so assume $\vdash r : {}^{p_1} wbalanced^{p_2}$. The case where r is already strongly balanced (rule WB-B) is immediate. The case of WB_1 follows by straightforward induction on the weakly balanced premise and using B-OO. In the case for WB_2, $r = r_1\, r_2$ with $\vdash r_1 : {}^{p_1} wbalanced^{p_2}$ and $\vdash r_2 : balanced^{p_2}$, both by subderivation. By induction, the weakly balanced r_1 is a prefix of a balanced trace t_1, i.e., $r_1\ s_1 = t_1$ for some t_1 with $\vdash t_1 : balanced^{p_1}$. By Lemma A.2.3, $\vdash r_1\ r_2\ s_1 : balanced^{p_1}$, as required.

For WB-CALL$^+$ we are given that $r = \gamma_c?\ r'$ and $\vdash t : balanced^+$. By rule B-IO. $\vdash \gamma_c?\ r'\ \gamma_r! : balanced^-$, as required. Rule WB-CALL$^-$ works symmetrically. □

Corollary A.2.6 (Closure under prefix)**.** *Weakly balanced traces are closed under prefix, i.e., $\vdash t : wbalanced$ and $s \preccurlyeq t$, then $\vdash s : wbalanced$*

Proof. An immediate consequence of Lemma A.2.5. □

Balance and weak balance are given by recursive, "context-free", definitions, capturing in a natural way the parenthetic nature of calls and returns. The operational semantics and the system for legal traces, however, generate, resp., check a trace not following the context-free structure of calls and return, but step by step. To prove invariants of the trace semantics or of legal traces (and the connection between the two), the following characterization is sometimes better suited (see also Lemma A.2.17, which constitutes basically the reverse direction of the next lemma).

Lemma A.2.7 (Number of calls and returns)**.** *Given t as trace of calls and returns. Let k_Δ be the number of outgoing calls minus the number of incoming returns, i.e., the number of calls unanswered by the environment. Dually, let k_Θ be the number of incoming calls minus the number of outgoing returns.*

1. *If t is alternating and for each prefix of t,*

$$k_\Delta \geq 0 \quad \textit{and} \quad k_\Theta \geq 0\,, \tag{A.1}$$

   then

   (a) *if the length of t is even, then $k_\Theta = k_\Delta$.*
   (b) *if the length of t is odd and the last label of t is outgoing, then $k_\Theta = k_\Delta - 1$. If alternatively, the last label is incoming, then $k_\Theta = k_\Delta + 1$.*

2. *If t is balanced, $k_\Delta = k_\Theta = 0$.*

3. *If t is weakly balanced, then equation (A.1) holds.*

4. *If t is weakly balanced, then the two implications of 1a and 1b hold for t.*

Proof. For part 1, proceed by straightforward induction on the length of the trace. Assume for one case that the first interaction of the trace is incoming, i.e., the thread starts in the environment. Let $r \preccurlyeq t$. The base case for $r = \epsilon$ trivially satisfies the conditions. Now consider $r\ a$ for the induction step. If r is even,

it means that a is an incoming communication, since the trace is alternating. By induction, using part 1a, $k_\Theta = k_\Delta$ before the extension. After the incoming call, the component has one more unanswered call, i.e., $\acute{k}_\Theta = k_\Theta + 1$, satisfying part 1b. An incoming return gives $\acute{k}_\Delta = k_\Delta - 1$, likewise covered by part 1b. If otherwise the length of r is odd, the last action of r is incoming, and therefore a is outgoing. By induction on part 1b, $k_\Theta = k_\Delta + 1$ for r. If a is an outgoing call, $\acute{k}_\Delta = k_\Delta + 1$, establishing the property of part 1a after the call. Similarly for outgoing returns, where $\acute{k}_\Theta = k_\Theta - 1$. The case where the first interaction in the trace is outgoing is analogous.

Part 2 is shown by straightforward induction on the rules of Table 3.3. Part 3 by straightforward induction on the rules from Table 3.4, using the result of part 2, and the easy observation that $\vdash a\ t : \mathit{wbalanced}$ implies that a is a call. Part 4 is the combination of part 1, 3, and the alternation Lemma A.2.1. □

Lemma A.2.8 (Weak balance: Characterization)**.** *Assume $\vdash t : \mathit{wbalanced}^-$ but not balanced. Then t is of the form $t_1\ \gamma_c!\ t_2$ with $\vdash t_2 : \mathit{balanced}^-$. The property holds dually for $\mathit{wbalanced}^+$.*

Proof. By Lemma A.2.7. □

The next lemma states that the derivation of balance of a trace is deterministic as far as the *choice* of rules of Table 3.3 is concerned. Note that in rules B-II and B-OO we require that s_1 and s_2 are non-empty. Note, however, that the tree of derivation is not determined by the trace whose balanced is checked.

Lemma A.2.9 (Determinism)**.** *Given a balanced trace s. Then for all derivations of $\vdash s : \mathit{balanced}^+$ (resp. $\vdash s : \mathit{balanced}^-$) exactly one of the following three conditions applies: All end with an instance* B-EMPTY$^+$*, or all end with* B-II*, or all end with* B-OI *(dually for $\vdash s : \mathit{balanced}^+$).*

Proof. In case of a non-empty trace, only B-II or B-OI applies. Assume for a contradiction then that $\vdash s : \mathit{balanced}^+$ can be derived by both B-II and B-OI in the last step. This means that

$$s = s_1\ s_2 = \gamma_c!\ s_1'\ s_2'\ \gamma_r!\ ,$$

where $\gamma_c!\ s_1'$ and $s_2'\ \gamma_r?$ are balanced and also $s_1' s_2'$ is balanced. We furthermore know that, since balanced, s_1 ends with a return, and similarly s_2 starts with a call. From Lemma A.2.7: The assumption that s_1 is balanced means the number of calls equals the number of returns in s_1. The assumption that $s_1'\ s_2'$ is balanced implies that in its prefix s_1' the number of returns is less or equal than the number of calls, which yields a contradiction. □

Lemma A.2.10 (Cut of a balanced trace)**.** *Assume $\vdash s_1 s_2 : \mathit{balanced}$. Then:*

1. *$\vdash s_1 : \mathit{balanced}$ iff $\vdash s_2 : \mathit{balanced}$.*
2. *$\vdash s_2 : \mathit{balanced}$ iff $\vdash s_2 : \mathit{wbalanced}$.*

And as direct consequence:

3. *$\vdash s_1 : \mathit{balanced}$ iff $\vdash s_2 : \mathit{wbalanced}$.*

Proof. Part 1 by Lemma A.2.9, Part 2 by Lemma A.2.7. Part 3 is a combination of 1 and 2. □

Lemma A.2.11 (Unique last unanswered call)**.** *Assume $s = s_1\ a_1\ t_1 = s_2\ a_2\ t_2$ where a_1 and a_2 are call labels. If t_1 and t_2 are balanced, then $s_1 = s_2$, $t_1 = t_2$, and $a_1 = a_2$.*

Proof. It suffices to show that $t_1 = t_2$. Assume for a contradiction that $t_1 \neq t_2$. Wlog. assume that t_1 is strictly longer than t_2, i.e., $t_1 = t_1'\ a_2\ t_2$. By Lemma A.2.10, $t_1'\ a_2$ is balanced, which is a contradiction; a balanced trace cannot end in a call (Lemma A.2.7). □

Lemma A.2.12 (Weak balance and case distinction)**.** *Let s be a weakly balanced trace. Then exactly one of the following three cases holds:*

1. *s is balanced.*
2. *$s = s_1\gamma_c?\ s_2$ for some call label γ_c, and where $\vdash s_2 : \mathit{balanced}^+$*
3. *$s = s_1\gamma_c!\ s_2$ for some call label γ_c, and where $\vdash s_2 : \mathit{balanced}^-$*

In case 2 and 3, the s_1, $\gamma_c?$ (resp., $\gamma_c!$), and s_2 are uniquely determined.

Proof. First of all it is clear that the three cases are mutually exclusive: When case 2 or 3 applies, the complete s is not balanced (using Lemma A.2.10), i.e., case 1 does not apply. Case 2 and 3 are mutually exclusive, as well, with the help of Lemma A.2.11. Now assume that s is weakly balanced. If it is not strictly balanced, the fact that s is of the form $s_1 a s_2$ for a call label a and where s_2 is balanced follows from Lemma A.2.7. The uniqueness of s_1, a, and s_2 follows also from Lemma A.2.11. □

Lemma A.2.13 (Functionality of pop)**.** *pop is a (partial) function on weakly balanced traces. Furthermore, $\mathit{pop}(t)$ is undefined iff t is balanced (assuming $\vdash t : \mathit{wbalanced}$).*

Proof. That *pop* returns at most one value is a consequence of Lemma A.2.11. If t is balanced, *pop* clearly is undefined. Now, if t is not balanced (but weakly balanced) the fact that *pop* is defined follows by Lemma A.2.12. □

Lemma A.2.14 (Sender and receiver)**.**

1. *Let $t = t_1\ t_2$ be a weakly balanced trace. Assume, t_2 is balanced and non-empty, i.e., $t_2 = \gamma_c?\ t_2'\gamma_r!$ and $\vdash t_2 : \mathit{balanced}^-$ (or dually, $\vdash t_2 : \mathit{balanced}^+$). Then $\mathit{receiver}(t_1\ t_2) = \mathit{sender}(t_1\ \gamma_c?)$.*
2. *Let $t\ \gamma?$ be a weakly balanced trace. If $t = \epsilon$, then $\mathit{sender}(t\ \gamma?) = \odot$. If otherwise $t = t'\ a$, then $\mathit{sender}(t'\ a\ \gamma?) = \mathit{receiver}(t'\ a)$ (and a is outgoing).*

Proof. Part 1 follows directly by definition of sender and of *pop*.

For part 2, proceed by induction on the length of t. The base case where $t = \epsilon$ and the induction case where $\gamma?$ is a call are immediate by the definition of *sender*. For returns, $\mathit{sender}(t'\ a\ \gamma_r?) = \mathit{receiver}(\mathit{pop}\ (t'\ a)))$. By the definition of *pop* and since $t'\ a\ \gamma_r?$ is alternating (Lemma A.2.1), $t'\ a = t'\ \gamma'! = t_1\ t_2 = t_1\ (\gamma_c^2?\ t_2'\gamma_r^2!)$ (i.e., $a' = \gamma_r^2!$) with $\vdash t_2 : \mathit{balanced}^-$, and furthermore, $\mathit{sender}(t'\ a\ \gamma_r?) = \mathit{receiver}(t_1)$. By induction, $\mathit{receiver}(t_1) = \mathit{sender}(t_1\ \gamma_c^2?)$. By part 1, $\mathit{sender}(t_1\ \gamma_c^2?) = \mathit{receiver}(t_1\ (\gamma_c^2?\ t_2'\gamma_r^2!))$ which equals $\mathit{receiver}(t'\ a)$, as required. □

Lemma A.2.15. *Assume* $\Xi_0 \vdash t : \mathit{balanced}$ *and* $\Xi_0 \vdash t \rhd \gamma?$*. Then* $\mathit{sender}(t\ \gamma?) = \odot$ *iff.* $\Xi_0 \vdash t : \mathit{balanced}^-$*,* $\Delta_0 \vdash \odot$*, and* γ *is a call. The same holds dually for* $\gamma!$*,* $\mathit{balanced}^+$*, and* $\Theta_0 \vdash \odot$*.*

Proof. An easy consequence of Lemma A.2.13. There are two directions to show. Assume first that $\mathit{sender}(t\ \gamma?) = \odot$. Assume further for a contradiction that the additional label is a return, i.e., $\gamma = \gamma_r$. By definition 3.3.4, the sender of γ_r equals $\mathit{receiver}(\mathit{pop}\, t)$ (with $t \neq \epsilon$). However, by Definition 3.3.1, $\mathit{pop}\ \ t$ is a call, and the receiver of that call cannot be $\odot$, yielding the contradiction.

Hence, the label is a call, say $\gamma = \gamma_c$. Since t is balanced, $\mathit{pop}\ \ t = \bot$ (Lemma A.2.13). Therefore, together with the call-enabledness assumption $\Xi_0 \vdash t \rhd \gamma?$ (Definition 3.3.3, equation (3.12)), $\Delta_0 \vdash \odot$, as required. The last claim $\Xi_0 \vdash t : \mathit{balanced}^-$ follows with Lemma A.2.12.

The reverse direction is similar: Since t is balanced, $\mathit{pop}\, t = \bot$ (Lemma A.2.13). The input-call enabledness assumption $\Xi_0 \vdash t \rhd \gamma_c?$ gives directly $\Delta_0 \vdash \odot$ (and the second clause of equation (3.12)) does not apply). □

Corollary A.2.16. *Assume* $\vdash t : \mathit{balanced}^+$*, and further* $\vdash s_1 : \mathit{wbalanced}^+$*. Then* $\vdash s_1\ t\ s_2 : \mathit{wbalanced}^+$ *iff.* $\vdash s_1\ s_2 : \mathit{wbalanced}^+$*. Dually* $\vdash s_1\ t\ s_2 : \mathit{wbalanced}^-$ *iff.* $\vdash s_1 s_2 : \mathit{wbalanced}^-$*. Two further dualizations hold where* $\vdash t : \mathit{balanced}^-$ *and* $\vdash s_1 : \mathit{balanced}^-$*.*

Proof. The two directions of the claim are covered by Lemma A.2.4(1), rule WB-INSERT, and Lemma A.2.2. □

The next lemma expresses the reverse characterization of weakly balanced traces of Lemma A.2.7 and Lemma A.2.1 (cf. page 57 for the definition of alternation).

Lemma A.2.17 (Number of calls and returns)**.** *Let* t *be an alternating trace. If for each prefix of* t*, the number of incoming returns is smaller or equal the number of outgoing calls, and dually for outgoing returns and incoming calls, then* t *is weakly balanced.*

Proof. Let $s \preccurlyeq t$, i.e., s is a prefix of t. We show by induction on the length of s, that it is weakly balanced, given the conditions on the number of calls and returns from the lemma. Let k_Δ be the number of outgoing calls minus the number of incoming returns, i.e., the number of calls unanswered by the environment. Dually, let k_Θ be the number of incoming calls minus the number of outgoing returns.

Case: Base case: $s = \epsilon$
Immediate by B-EMPTY$^+$ or B-EMPTY$^-$.

Case: Incoming call: $s = s'\gamma_c?$
By assumption, $k_\Theta \geq 0$ and $k_\Delta \geq 0$ after s' and furthermore (by induction), s' is weakly balanced. Now consider $s\gamma_r! = s'\gamma_c?\gamma_r!$, for some outgoing return $\gamma_r!$. The pair $\gamma_c?\gamma_r!$ is balanced, i.e., $\vdash \gamma_c?\gamma_r! : \mathit{balanced}^-$. By Corollary A.2.16, $s'\gamma_c?\gamma_r!$ is weakly balanced. Hence, being shorter, also $s'\gamma_c?$ is weakly balanced, as required.

Case: Outgoing return: $s = s'\gamma_r!$
By assumption, $k_\Theta \geq 1$ after s', and by induction, s' is weakly balanced (but not balanced). By Lemma A.2.11, s' is of the form $s'_1\gamma_c?s'_2$ with $\vdash s'_2 : \mathit{balanced}^+$, for some incoming call label $\gamma_c?$. This implies with B-IO that $\vdash \gamma_c?\ s'_2\ \gamma_r! : \mathit{balanced}^-$. By Corollary A.2.16, $s'_1\gamma_c?\ s'_2\ \gamma_r!$ is weakly balanced if s'_1 is. The latter is given by induction.

The remaining two cases for outgoing calls and incoming returns are dual. □

Lemma A.2.18 (Weak balance and enabledness)**.** *Assume* $\vdash t : \mathit{wbalanced}$.

1. *If* $\Xi_0 \vdash t \rhd \gamma_r?$*, then* $\Xi_0 \vdash t \rhd \gamma_c?$*. Dually for* $\gamma_c!$ *and* $\gamma_r!$*.*
2. *If* t *is non-empty, then either* $\Xi_0 \vdash t \rhd \gamma'?$ *or* $\Xi_0 \vdash t \rhd \gamma''!$*.*
3. $\vdash t : \mathit{wbalanced}^+$ *and* $\vdash t : \mathit{wbalanced}^-$ *iff* $t = \epsilon$*.*
4. *If* $\vdash t : \mathit{balanced}^-$ *and* $\Delta_0 \vdash \odot$*, then* $\Xi_0 \vdash t \rhd \gamma_c?$ *and* $\Xi_0 \nvdash t \rhd \gamma_r?$*. The case holds dually for* $\mathit{balanced}^+$*,* Θ_0*, and* $\gamma_c!$*, resp.,* $\gamma_r!$*.*
5. *Assume* $\vdash t : \mathit{wbalanced}$ *and* $\vdash t \rhd a$*. Then* $\vdash t : \mathit{wbalanced}^-$ *iff* a *is an incoming communication. Dually for* $\mathit{wbalanced}^+$*.*

Proof. See Definition 3.3.3 for the definition of enableness. Part 1 follows directly from the definition of enabledness.

For part 2: Because return enabledness implies call enableness by part 1, we need to consider only the case of two calls. By the case distinction of Lemma A.2.12, trace t is of exactly one of three possible forms. If t is strictly balanced, corresponding to A.2.12(1), $\mathit{pop}\ t = \bot$ (Lemma A.2.13), and therefore, either $\Xi_0 \vdash t \rhd \gamma_c?$ or $\Xi_0 \vdash t \rhd \gamma_c!$, by the first line of equation (3.12), resp., of (3.13) and the fact that either $\Delta_0 \vdash \odot$ or $\Theta_0 \vdash \odot$. Otherwise, either A.2.12(2) or A.2.12(3) applies, i.e., t is either of the form $t_1\ \tilde{\gamma}_c^1!\ t_2$ or of $t_1\ \tilde{\gamma}_c^2?\ t_2$, where t_2 is balanced, and furthermore $\mathit{pop}\ t = t_1\ \tilde{\gamma}_c^1!$ or else $\mathit{pop}\ t = t_1\ \tilde{\gamma}_c^2?$, depending on which of the two alternatives applies. Therefore, the second line of either (3.12) or of (3.13) applies, giving either $\Xi_0 \vdash t \rhd \gamma_c^1?$ or $\Xi_0 \vdash t \rhd \gamma_c^2!$, as required.

The two directions of part 3 are covered by part 2 and by the rules B-EMPTY$^+$ and B-EMPTY$^-$ from Table 3.3 (in combination with WB-B from Table 3.4).

For part 4, assume for one of the two dual cases that $\vdash t : \mathit{balanced}^-$ and $\Delta_0 \vdash \odot$. The judgment $\Xi_0 \vdash t \rhd \gamma_c?$ follows by Lemma A.2.13 and directly from the definition of enabledness, equation (3.12). The fact that $\Xi_0 \nvdash t \rhd \gamma_r?$ follows likewise by Lemma A.2.13 and the definition of input return enabledness. □

The next lemma basically shows that weak balance is an invariant of the traces of a component, respectively a legal trace: Adding an enabled label to a weakly balanced trace preserves weak balance. Remember that enabledness is one of the premises being checked for doing one step in the external semantics, resp., in the system for legal traces.

Lemma A.2.19 (Weak balance and enabledness)**.** *Assume a trace* t *with* $\Xi_0 \vdash t : \mathit{wbalanced}$ *and* $\Xi_0 \vdash t \rhd a$*, i.e.,* a *is enabled after* t*. Then the following holds:*

1. *If* $a = \gamma?$*, then* $\Xi_0 \vdash t\ \gamma? : \mathit{wbalanced}^+$*.*

2. *If* $a = \gamma!$, *then* $\Xi_0 \vdash t\ \gamma? : wbalanced^-$.

And as direct consequence:

3. $\Xi_0 \vdash t\ a : wbalanced$.

Proof. Part 3 is just a combination of the other two parts. For part 1 and 2 we exploit the characterization of weakly balanced traces in terms of numbers of calls and returns from Lemma A.2.7 and A.2.17. For the enabledness assertion $\Xi_0 \vdash t \rhd a$, see Definition 3.3.3. Let k_Θ and k_Δ be defined as in Lemma A.2.7.

By assumption, t is weakly balanced, i.e., Lemma A.2.7 gives $k_\Delta \geq 0$ and $k_\Theta \geq 0$.

Case: Incoming call ($a = \gamma_c?$)
If a is an incoming *call*, the two inequations still hold after the communication, hence by Lemma A.2.17, $t\ a$ is still weakly balanced, yielding $\Xi_0 \vdash t\ \gamma_c? : wbalanced$. That $\Xi_0 \vdash t\ \gamma_c? : wbalanced^+$ follows straightforwardly, yielding part 1 of the lemma.

Case: Incoming return ($a = \gamma_r?$)
In this case, we must show that in particular $k_\Delta - 1 \geq 0$ holds after the return; the value of k_Θ remains unchanged. By Definition 3.3.3 of input-return enabledness, $\Xi_0 \vdash t \rhd \gamma_r?$ means $pop\, t = t_1\ \gamma_c'!$ for some call label γ_c', were $t_1\ \gamma_c'!$ is a (not necessarily proper) prefix of t. By Definition 3.3.1 of pop, we stronger know that $t = t_1\ \gamma_c'!\ t_2$, where t_2 is balanced. By Lemma A.2.7(2), the difference of calls minus returns (in both directions) is 0 concerning the balanced t_2. Thus, the value of k_Δ after t equals that value after $t_1\gamma_c'!$, which implies $k_\Delta \geq 1$, and thus, the difference is still ≥ 0 after $t\ \gamma_r?$. The result therefore follows by Lemma A.2.17.

For outgoing communication, the argument is dual. □

The following is an easy observation in the definition of legal traces: Each label in the trace is enabled at the point in the trace before the label.

Lemma A.2.20 (Legality and enabledness)**.** *Assume* $\Xi_0 \vdash t\ a : trace$, *then* $\Xi_0 \vdash t \rhd a$.

Proof. By induction on the length of t and inspection of the rules from Table 3.5 (resp. Table 5.1): The enabledness judgment $\Xi_0 \vdash t \rhd a$ is a premise of each of the rules for legal traces (where it appears in the form of $\Xi_0 \vdash t \rhd o_1 \xrightarrow{a} o_2 : \vec{T} \to \_$ or $\Xi_0 \vdash t \rhd o_1 \xrightarrow{a} o_2 : \_ \to T$, where additionally the communication partners and the expected types are determined). □

Lemma A.2.21 (Legality and balance)**.** *If* $\Xi_0 \vdash t : trace$, *then* $\vdash t : wbalanced$.

Proof. In each step when checking legality of a trace, the enabledness of the next label a after r is checked by a premise of the form $\Xi_0 \vdash r \rhd a$.[2] Thus, the claim follows with preservation of weak balance when extending a trace by an enabled label (Lemma A.2.19). □

[2] In the rules of Table 3.5 or 5.1, the corresponding actual premise reads $\Xi_0 \vdash r \rhd o_1 \xrightarrow{a} o_2$, but this judgment contains $\Xi_0 \vdash r \rhd a$ as part of its definition, and the connectivity part referring to sender and receiver object does not concern us in this lemma.

A.2.2 Balance and swapping

The fact that the absolute order of certain labels is unobservable, when the observer is split into separate cliques, has been captured using the forward projection of a trace, i.e., considering the interaction from the perspective of single objects. Since the projection takes the merging of cliques into account, it captures the tree-like structure of the semantics (cf. Definition 3.1.7) Analogously, the effect of replay has been captured using the forward projection; see Definition 3.1.8 for the definition of $\preccurlyeq_\Theta$ and its symmetric variant $\succcurlyeq_\Theta$.

The mentioned definitions represent the tree-like structure of the semantics appropriately, which is emphasized also by the fact that the forward projection builds the core of the implementation of the observer in the completeness proof. The global nature of the definition of $s \succcurlyeq_\Theta t$, based on projections, however, makes it hard, to prove properties about s and t, two *linear* traces, not trees, and their relationship. Properties we are interested in are certain preservation properties, e.g.: If s is weakly balanced and $s \asymp_\Theta t$, then t is weakly balanced, as well.

As the implicit definition using local projections is ill-suited for proving this kind of properties, we present an alternative characterization of the swapping and replay relation, were $s \asymp_\Theta t$, resp., $s \succcurlyeq_\Theta t$ is represented by a number of elementary transformation steps, providing an "equational" representation of the relations, where interactions with different cliques can be swapped and new interactions can be added or removed due to replay. The corresponding transformation rules for traces resemble the informal examples from Section 1.4.

The swapping and replay rules are not literally an "equational" representation of the tree-like structure of the semantics. What makes it more complex than a plain equational or rewriting representation is that the equations cannot be applied to a subsequences of a trace in isolation, without taking (parts of) the whole trace into account. I.e., we cannot have an equation as follows: If s and t belong to two different cliques, then

$$s\ t \asymp_\Theta t\ s\ ,$$

and use this equation to conclude that

$$r\ s\ t\ u \asymp_\Theta r\ t\ s\ u\ .$$

There are two main reasons for this. First, the question whether s and t belong to two different cliques or not *depends* on the previous history r. Secondly (and related), the trace does not just consist of a sequence of labels, but must adhere to the balance requirements regulating the connections of the calls and returns. An additional more subtle point is that the possibility of swapping of s and t does not only depend on the history r, but also on the *future* u. In other words, we cannot in general conclude:

$$\frac{r\ s\ t \asymp_\Theta r\ t\ s}{r\ s\ t\ u \asymp_\Theta r\ t\ s\ u}$$

The failure of this property captures the fact that under certain circumstances, the *order* of interactions, here the order of s and t, can be observed *in retrospect*.

To sum up, $\asymp_\Theta$ and $\preccurlyeq_\Theta$ are not context-free equations on traces, but swapping of parts of a trace has to be considered in the context of the past behavior r as well as the future behavior u. To distinguish the "equational" representation of the swapping and the replay relation from their original definitions, we denote them by $\dot{\asymp}_\Theta$ and $\dot{\preccurlyeq}_\Theta$. We show that $\dot{\asymp}_\Theta$ and $\asymp_\Theta$, resp., $\dot{\preccurlyeq}_\Theta$ and $\preccurlyeq_\Theta$, coincide (see Corollary A.2.45, resp., A.2.55).

We start with the equational definition of swapping.

Definition A.2.22 (Swapping)**.** *The swapping relation $\dot{\asymp}_\Theta$ on traces is as the reflexive, transitive, and symmetric closure of the rules from Table A.1. In the rules, $|t|$ denotes the length of the trace t, i.e., the number of labels of the trace. The definition of $\dot{\asymp}_\Delta$ is dual.*

$$\frac{\Xi_0 \vdash_\Theta s \rhd t_1 \not\asymp t_2 \qquad \vdash u : \mathit{wbalanced} \qquad \vdash t_1, t_2 : {}^p\mathit{wbalanced}^p}{\Xi_0 \vdash s\ \nu(\Phi).t_1\ t_2\ u \asymp_\Theta s\ \nu(\Phi).t_2\ t_1\ u}\ \textsc{SwapW}_\Theta$$

$$\frac{\Xi_0 \vdash_\Theta s \rhd t_1 \not\asymp t_2 \qquad \vdash t_1 : \mathit{balanced} \qquad |t_2|\mathit{even}}{\Xi_0 \vdash s\ \nu(\Phi).t_1\ t_2\ u \asymp_\Theta s\nu(\Phi).t_2\ t_1\ u}\ \textsc{SwapB}_\Theta$$

Table A.1: Swapping

The next lemma states that balance is preserved by swapping.

Lemma A.2.23 (Swapping and balance)**.**

1. *If $\vdash t_1 : \mathit{wbalanced}^+$ and $\Xi_0 \vdash t_1 \dot{\asymp}_\Theta t_2$, then $\vdash t_2 : \mathit{wbalanced}^+$. Analogously for $\mathit{wbalanced}^+$ (and for $\dot{\asymp}_\Delta$).*
2. *If $\vdash t_1 : \mathit{balanced}^+$ and $\Xi_0 \vdash t_1 \dot{\asymp}_\Theta t_2$, then $\vdash t_2 : \mathit{balanced}^+$. Analogously for $\mathit{balanced}^+$ (and for $\dot{\asymp}_\Delta$).*

Proof. There are two parts to show and we start with part 1 for weak balance. So let $t_1 = s\ \nu(\Phi).t_1'\ t_2'\ u$. We need to show that the rules from Table A.1 preserve weak balance. If t_1' or t_2' equals ϵ, the argument is trivial. So assume, t_1' and t_2' are not empty.

Case: $\textsc{SwapW}_\Theta$
By Lemma A.2.1, $\vdash t_1 : \mathit{wbalanced}^+$ implies that t_1 is alternating. Furthermore, the alternation lemma implies that $\vdash t_1' : \mathit{wbalanced}^-$ iff. $\vdash t_2' : \mathit{wbalanced}^-$ (and analogously for $\mathit{balanced}^+$), since they are of even length, as required by the premises of $\textsc{SwapW}_\Theta$ and $\textsc{SwapB}_\Theta$.

Since $\Xi_0 \vdash_\Theta s \rhd t_1 \not\asymp t_2$, the situation $\vdash t_1' : \mathit{wbalanced}^+$ (and $\vdash t_1' : \mathit{wbalanced}^+$) cannot be the case (but see also Remark A.2.24). Hence we are given $\vdash t_1' : \mathit{wbalanced}^-$ and $\vdash t_2' : \mathit{wbalanced}^-$. Furthermore, $\vdash s : \mathit{wbalanced}^-$. The latter fact is justified as follows. First $\vdash s : \mathit{wbalanced}$ follows with Corollary A.2.6 from $s \preccurlyeq t$, where $\vdash t : \mathit{wbalanced}$ by assumption. If $s = \epsilon$, $\vdash s : \mathit{wbalanced}^-$ follows by B-EMPTY^-. If $s \neq \epsilon$, $\vdash s : \mathit{wbalanced}^-$, since the weakly balanced t alternating (Lemma A.2.1) and since t_1' and t_2' are not empty.

We distinguish whether the trailing u is empty or not. Assume first $u \neq \epsilon$. In the given situation, where $\vdash t_2' : wbalanced^-$, the premise $\vdash u : wbalanced$ implies that we know stronger that $\vdash u : {}^-wbalanced^p$, since t_1 is alternating. In the first of the two dual parts of the lemma, we additionally have $p = +$. Hence, the result follows using three times the concatenation Lemma A.2.4 for weakly balanced traces.

When $u = \epsilon$, the concatenation lemma needs to be applied only twice.

Case: SWAPB$_\Theta$
We are given that t_1' is strictly balanced. Note that u and t_2' need not even be weakly balanced. The premise $\Xi_0 \vdash_\Theta s \rhd t_1 \not\Rightarrow t_2$ implies that $\vdash t_1' : balanced^-$. The fact that t_1 is alternation gives, as in the case for SWAPW$_\Theta$, that $\vdash s : wbalanced^-$. Furthermore, t_2' is of the form

$$\gamma_1''?\, t_2''\, \gamma_2''! \tag{A.2}$$

(remember that we agreed that t_2' is not empty). For the end of the trace we distinguish whether u is empty or not. If $u \neq \epsilon$, $u = \gamma_3?\, u'$. By Lemma A.2.2 $\vdash s\, t_2' u : wbalanced$. By preservation of weak balance under prefixing of Corollary A.2.6, $\vdash s\, t_2' : wbalanced$ and by the form of t_2' from equation (A.2), $\vdash s\, t_2' : wbalanced^-$. Hence the result follows by Lemma A.2.4, rule WB-INSERT.

In part 2, we need to show the same property for strict balance instead of weak balance. Again, if t_1' and t_2' are empty, the result is trivial, so let t_1' and t_2' be different from ϵ. The rest of the argument is similar to the one for part 1

Case: SWAPW$_\Theta$
As above, we get $\vdash t_1' : wbalanced^-$ and $\vdash t_2' : wbalanced^-$. If the trailing u is not empty, $\vdash u : {}^-wbalanced^p$ (as above). Assuming further, for one of two possible cases, that $p = +$, we argue as follows. By the cut Lemma A.2.10(2), u is not just weakly balanced, but strictly balanced, i.e., here $\vdash u : balanced^-$. By Lemma A.2.10(1), this means $\vdash s\, t_1'\, t_2' : balanced^-$. Using the same argument twice more times gives $\vdash s : balanced^-$ and $\vdash t_1' : balanced^-$ and $\vdash t_2' : balanced^-$. Thus the result follows with rule B-OO and transitivity. For $u = \epsilon$, the argument is analogous.

As a remark: Effectively, the assumption that t_1 is balanced showed that the use of the swapping rule SWAPW$_\Theta$ for weakly balanced sub-sequences actually swapped balanced sub-sequences.

Case: SWAPB$_\Theta$
Similar. □

Remark A.2.24 (Preservation of balance)**.** *Note that the proof of Lemma A.2.23 used the premises $\Xi_0 \vdash s \rhd t_1 \not\Rightarrow t_2$ to exclude certain situations concerning the polarity of the swapped subsequences.*

Indeed, the excluded situations could have been proven in same, i.e., dual, manner than the possible ones. The pure preservation of balance and the alternation of the thread is independent of the connectivity information and the preservation of enabledness under swapping from Lemma A.2.23 holds analogously also, when omitting *the premises $\Xi_0 \vdash s \rhd t_1 \not\Rightarrow t_2$ from the rules of Table A.1.* □

Lemma A.2.25 (Independence)**.** *Assume $\vdash s\ t : wbalanced$. If $\vdash t : wbalanced$, then*

1. *If* $\vdash t : balanced$, *then* $pop(s\ t) = pop(s)$. *Otherwise,*
2. *if* $\not\vdash t : balanced$, *then* $pop(s\ t) = s\ \ pop(t)$.

Furthermore, if $\vdash t : balanced$ *(i.e., in the situation of part 1),* $pop(s\ t)$ *is undefined iff* $pop\ \ s$ *is undefined iff* $\vdash s\ t : balanced$.

Proof. For the definition of pop, see Definition 3.3.1. Let $pop(s\ t)$ be defined. Part 1 is immediate by definition. For part 2 we have $\not\vdash t : balanced$ but weakly balanced. This implies with the characterization from Lemma A.2.12 that $t = t_1\ \gamma_c\ t_2$ such that t_2 is balanced . Thus $pop(s\ t)$ = $pop(s\ t_1\ \gamma_c\ t_2) = s\ t_1\ \gamma_c = s\ pop(t_1\ \gamma_c\ t_2) = s\ \ pop(t)$.

For the claim about definedness: By Lemma A.2.13, $pop\ \ (s\ t)$ is undefined iff $s\ t$ is balanced. The fact that $pop(s\ t)$ is undefined iff $pop\ \ s$ is follows immediately from Lemma A.2.13 and Lemma A.2.10. □

The symbol $\odot$ represents the initial clique, where the thread starts its life. In the multi-threaded setting, it is the starting clique of the initial thread (in the multi-threaded setting, additionally $\odot_n$ represents the initial clique of the thread n). The $\odot$ is not a "real" clique, i.e., a collection of instantiated objects, but needed to represent the connectivity appropriately, in particular, to have a representative for the connectivity of the initial activity, even if no real object happens to be known.[3] The next lemma characterizes under which circumstances $\odot$ functions a communication partner, namely basically when the history before the communication step in question is balanced (for the sender of a call and for the receiver of a return).

Lemma A.2.26 ($\odot$ as communication partner)**.**

1. *Assume* $\vdash t\ \gamma_c? : wbalanced$
   (a) $receiver(t\ \gamma_c?) \neq \odot$.
   (b) $sender(t\ \gamma_c?) = \odot$ *iff* $\vdash t : balanced^-$.
2. *Assume* $\vdash t\ \gamma_r? : wbalanced$
   (a) $sender(t\ \gamma_r?) \neq \odot$.
   (b) $receiver(t\ \gamma_r?) = \odot$ *iff* $\vdash t\ \gamma_r? : balanced^+$.

For outgoing calls and incoming returns, the statements hold dually.

Proof. See 3.3.4 for the definition of sender and receiver. Proceed by induction on the length of the trace. Part 1a is immediate by definition: The receiver of a call, the callee, is directly mentioned in the label.

For part 1b, there are two directions two show. If $\vdash t : balanced^-$ and t is empty, then $sender(\gamma_c?) = \odot$ by definition. If otherwise, $t = t'\ a'$, $sender(t\ \gamma_c?) = receiver(t'\ a')$. Since $t = t'\ a'$ is balanced, a' is a return, and since $t'\ a'\ \gamma_c?$ is alternating (Lemma A.2.1), the return is outgoing, i.e., $a' = \gamma_r'!$, and $\vdash t'\ \gamma_r'! : balanced^-$. So the result follows by induction on part 2b. For the reverse direction of part 1b, we are given $sender(t\ \gamma_c?) = \odot$. If t is empty, the result is immediate by B-EMPTY$^-$ of Table 3.3. For $t \neq t'\ a'$, we are given

[3] This may happen, since the sender of call is not transmitted.

$sender(t\ \gamma_c?) = sender(t'\ a'\ \gamma_c?) = receiver(t'\ a') = \odot$. Again, the label a' is an outgoing return, and $t'\ a'$ is balanced by induction on (the dual of) part 2b.

For part 2a: By definition, $sender(t\ \gamma_r?) = receiver(pop(t))$. Since $pop(t)$ ends with an (outgoing) call (Lemma A.2.12), the case follows by induction on (the dual of) part 1a.

For part 2b, there are two directions to show. Assume $receiver(t\ \gamma_r?) = \odot$. By definition, $receiver(t\ \gamma_r?) = sender(pop(t))$. The non-empty trace $pop(t)$ ends in outgoing call (Lemma A.2.12), i.e., $t = t'_1\ \gamma'_c!\ t'_2$ s.t. t'_2 is balanced. This implies by B-IO that $\gamma'_c!\ t'_2\ \gamma_r?$ is balanced. With the cut Lemma A.2.10, t'_1 is balanced, as well, from which the result follows by rule B-II of Table 3.3. For the reverse direction, assume $\vdash t\ \gamma_r? : \mathit{balanced}^+$. By definition, $receiver(t\ \gamma_r?) = sender(pop(t))$, where $pop(t)$ ends in an outgoing call, i.e., as above, $t = t'_1\ \gamma'_c!\ t'_2$ s.t. t'_2 is balanced. As in the previous direction, Lemma A.2.10 yields that the shorter t'_1 is balanced, more precisely, $\vdash t'_1 : \mathit{balanced}^+$. By induction on (the dual of) part 1b, $sender(pop(t)) = \odot$, as required. □

The next lemma covers a crucial property for showing that the swapping relation $\doteq_\Theta$ preserves sender and receiver of the labels in a trace. The informal discussion at the beginning of Section A.2.2 mentioned that the possibility of swapping of subsequences of a trace may depend on the rest of the trace, the past as well as the future. This dependence is (amongst other reasons) caused by the fact that the communication partners of a trace may depend on the prior parts being swapped. The next lemma thus characterizes when the sender and receiver of a trace depends on a past subsequence: Given a trace $s\ t\ a$ ending in a weakly balanced $t\ a$, then the sender and receiver of the label a are determined by t (and thus do not depend on s), or equal $\odot$, in which case, they may depend on s.

Lemma A.2.27 (Independence)**.**

1. *Assume* $\vdash s\ t\ a : \mathit{wbalanced}$. *If* $\vdash t\ a : \mathit{wbalanced}$, *then*
   (a) $sender(s\ t\ a) = sender(t\ a)$, *or* $a = \gamma_c$ *and* $sender(t\ a) = \odot$, *and*
   (b) $receiver(s\ t\ a) = receiver(t\ a)$, *or* $a = \gamma_r$ *and* $receiver(t\ a) = \odot$.
2. *Assume* $\vdash s\ t\ u\ a : \mathit{wbalanced}$. *If* $\vdash t\ \ : \mathit{balanced}$, *then* $sender(s\ t\ u\ a) = sender(s\ u\ a)$ *and* $receiver(s\ t\ u\ a) = receiver(s\ u\ a)$.

Proof. For sender and receiver, see Definition 3.3.4.

Part 1 (weak balance)
If $s = \epsilon$, the result is immediate. So proceed by induction on the length of $s\ t$, assuming that $s \neq \epsilon$. We distinguish according to the form of a.

Case: Outgoing return: $a = \gamma_r!$
We have:

$$
\begin{array}{lll}
receiver(s\ t\ \gamma_r!) & = sender(pop(s\ t)) & \text{(Definition 3.3.4)} \\
& = sender(s\ \ pop(t)) & \text{(Lemma A.2.25(2))} \\
& = sender(s\ t'\ a') & \text{(by definition of } pop\text{)} \\
& = sender(t'\ a') & \text{(induction on part 1a, case (i))} \\
& = sender(pop\ t) & \\
& = receiver(t\ \gamma_r!)\ . &
\end{array}
$$

To apply Lemma A.2.25(2) in the above chain note that $\vdash t\ \gamma_r! : \mathit{wbalanced}$ implies that t is not strictly balanced (Lemma A.2.7), since $t\ \gamma_r!$ ends with a return. To apply the induction hypothesis, not that $s\ t'$ is strictly shorter than $s\ t$, and furthermore that $t'\ a'$ and $s\ t'\ a'$ are weakly balanced (Lemma A.2.5).

Alternatively, (ii) $\mathit{sender}(t'\ a') = \odot$ and a' is a call, i.e., in particular an incoming call, say $\gamma_c'?$ because of alternation. This implies with Lemma A.2.26 that $\vdash t' : \mathit{balanced}^-$, and this gives by Definition 3.3.4 that $\mathit{receiver}(t\ \gamma_r!) = \odot$, as required.

Concerning the sender: We are given $\mathit{sender}(s\,t\,\gamma_r!) = \mathit{sender}(s\ (t'\ a')\ \gamma_r!) = \mathit{receiver}(\mathit{pop}(s\ (t'\ a')))$, i.e., $t = t'\ a'$, where $a' = \gamma'!$, as before. Lemma A.2.25 gives, $\mathit{receiver}(\mathit{pop}(s\ (t'\ a'))) = \mathit{receiver}(s\ \ \mathit{pop}(t'\ a'))$, from which the case follows by induction.

Case: Incoming call: $a = \gamma_c?$
For the receiver, the case is immediate by definition. Concerning the sender: If s and t are empty (and hence the sender equals $\odot$), the statement holds trivially. Otherwise, $s\ t\ \gamma_c? = u'\ a'\ \gamma_c?$ (where $a = \gamma'!$ because of alternation, see Lemma A.2.1) and $\mathit{sender}(u'\ \gamma'!\ \gamma_c?) = \mathit{receiver}(u'\gamma'!)$. Now, if $t = t'\gamma'!$, i.e., $s\ t = s\ (t'\gamma!)$, the result follows by induction on part 1b. Otherwise, $t = \epsilon$ and $s = s'\gamma'!$. Hence, $\mathit{sender}(t\ \gamma_c?) = \mathit{sender}(\gamma_c?) = \odot$, as covered by part 1a.

The cases for outgoing returns and incoming calls are dual.

Part 2 (balance)
Straightforward, by the definition of sender and receiver and using the *pop*-function. □

The next lemma shows that swapping, in most cases, preserves the information about sender and receiver, but not always. In Lemma A.2.28 below, formulated for $\doteq_\Theta$, the communication partners in the environment may not be preserved (see part 1 of the lemma). This change can affect the sender of an incoming call or the receiver of an outgoing return (for $\doteq_\Delta$, the situation is dual). In the light of our intention, that traces in $\doteq_\Delta$-relation are observably equivalent, or dually, that $\doteq_\Theta$ captures a closure condition on the set of traces, this seems odd. Critical in this context is especially rule SWAPW$_\Delta$, resp., its dual SWAPW$_\Theta$. The swapping of a strictly balanced part, in contrast, preserves sender and receiver. For SWAPW$_\Theta$, however, the trailing u is required to be weakly balanced *in isolation.* The property that certain communication partners are not preserved by swapping reflects the fact that the sender of a call is not transmitted in a label and thus cannot be observed by the callee (and indirectly, neither can the receiver of returns).

Lemma A.2.28 (Swapping and communication partners)**.** *Assume* $\Xi_0 \vdash t_1\ a \doteq_\Theta t_2\ a$, *where* $\vdash t_1\ a : \mathit{wbalanced}$.

1. *communication partner in* Δ*:*
   (a) $\mathit{receiver}(t_1\ \gamma_c!) = \mathit{receiver}(t_2\ \gamma_c!)$.
   (b) $\mathit{sender}(t_1\ \gamma_r?) = \mathit{sender}(t_2\ \gamma_r?)$.
2. *communication partner in* Θ*:*
   (a) $\mathit{sender}(t_1\ \gamma!) = \mathit{sender}(t_2\ \gamma!)$

(b) $receiver(t_1\ \gamma?) = receiver(t_2\ \gamma?)$

If $\doteq_\Theta$ *is justified by instances of* SWAPB$_\Theta$ *alone, also for* Δ*-objects in part 1, additionally sender of incoming calls and the receiver of outgoing returns are preserved, i.e., combining 1 and 2 and merging the cases, which gives:*

3. $sender(t_1\ a) = sender(t_2\ a)$ *and* $receiver(t_1\ a) = receiver(t_2\ a)$.

The lemma holds dually for $\doteq_\Delta$.

Proof. For the sender and receiver of a label, see Definition 3.3.4. First, by Lemma A.2.23, swapping preserves weak balance; hence, sender and receiver of a after t_2 are well-defined (Definition 3.3.4 insists on its argument to be weakly balanced).

Case: SWAPW$_\Theta$: $t_1\ a = s\ \nu(\Phi).t_1^1\ t_1^2\ u'\ a$ with $\vdash u'\ a : wbalanced$
We distinguish, whether the communication partner is part of the environment or of the component.

Subcase: Communication partner in the environment (part 1)
The case where $a = \gamma_c!$ in part 1a is covered by Lemma A.2.27(1b). If $a = \gamma_r?$ in part 1b is covered by Lemma A.2.27(1a).

Subcase: Communication partner in the component (part 2)
In this case we need to show preservation of the communication partners in all situations. We distinguish according to the form of a, where the argument in the first two cases corresponds to the one for dealing with the communication partners in the environment.

Subsubcase: Incoming call ($a = \gamma_c?$)
Part 2b is covered by Lemma A.2.27(1b), since the receiver of the call is independent of the swapped part.

Subsubcase: Outgoing return ($a = \gamma_r!$)
Part 2a is covered by analogously by Lemma A.2.27(1a).

Subsubcase: Outgoing call ($a = \gamma_c!$)
If t_1' or t_2' are empty, the case is trivial. Thus assume that both subsequences are non-empty. By Lemma A.2.27(1a), there are two cases to consider. In the first case, if $sender(s\ \nu(\Phi).t_1^1\ t_1^2\ u'\ a') = sender(u'\ \gamma_c!)$, we are done.

Otherwise,

$$sender(u'\ \gamma_c!) = \odot\ . \tag{A.3}$$

We argue that this case cannot occur. The premise $\Xi_0 \vdash_\Theta s \rhd t_1^1 \not\leftrightharpoons t_1^2$ of rule SWAPW$_\Theta$, implies $\vdash t_1^1 : {}^-wbalanced^-$ and $\vdash t_1^2 : {}^-wbalanced^-$, and furthermore,

$$\vdash u'\ \gamma_c! : {}^-wbalanced^-\ . \tag{A.4}$$

Equation A.3 implies with (the dual of) Lemma A.2.26(1b), $\vdash u' : balanced^+$. Now, that contradicts the polarity information in equation A.4, refuting the assumption from equation (A.3).

Subsubcase: Incoming return ($a' = \gamma_r?$)
Analogously.

Case: SWAPB$_\Theta$: $t_1 = s\ \nu(\Phi).t_1^1\ t_1^2\ u$ with $\vdash t_1^1 : \mathit{balanced}$
Note first that stronger $\vdash t_1^1 : \mathit{balanced}^-$, since the trace is alternating, since balanced sub-sequences are of even length, and because of the requirement $\vdash s \rhd t_1^1 \not\eqsim t_1^2$. The case is covered by Lemma A.2.27(2). □

The next example illustrates the lemma, in particular using the swapping of weakly balanced traces from the perspective of the component, i.e., we illustrate mainly SWAPW$_\Theta$ and the role of the sender and receiver of a communication in that context.

Example A.2.29 (Swapping)**.** *Consider the following trace $s_1 = s_1^1\ s_1^2$:*

$$s_1 = \nu(o_1{:}c).\langle \mathit{call}\ o_1.l()\rangle?\ \langle \mathit{call}\ o.l()\rangle!\ \nu(o_2{:}c).\langle \mathit{call}\ o_2.l()\rangle?\ \langle \mathit{call}\ o.l()\rangle!\ , \quad \text{(A.5)}$$

where s_1^1 consists of the first two calls and s_1^2 of the remaining two. The interactions with the two component cliques, represented by o_1 and o_2 are not strictly balanced, but weakly balanced. Clearly, the component can perform the two interactions also in the reversed order

$$s_2 = \nu(o_2{:}c).\langle \mathit{call}\ o_2.l()\rangle?\ \langle \mathit{call}\ o.l()\rangle!\ \nu(o_1{:}c).\langle \mathit{call}\ o_1.l()\rangle?\ \langle \mathit{call}\ o.l()\rangle!\ , \quad \text{(A.6)}$$

i.e., $s_1 \dot{\asymp}_\Theta\ s_2$.

Assume now that the two cliques are merged by a further incoming call, where s_1 is extended by $\gamma_c? = \langle \mathit{call}\ o_1.l'(o_2)\rangle?$. Now, does the following equality hold?

$$s_1\ \gamma_c? \dot{\asymp}_\Theta\ s_2\ \gamma_c? \quad \text{(A.7)}$$

According to SWAPW$_\Theta$*, the equality indeed holds. The intuitive justification for that equation is, that each component showing the left-hand trace of (A.7) shows also the one on the left-hand side (and vice-versa). In other words, the component code executed after $\gamma?$ must not be able to react differently after s_1 and s_2. This can be seen by looking at the behavior of the component more closely.*

Assuming that we start with an empty stack, the reduction looks as follows, where we show only the thread-part of the component. Let us abbreviate the blocked stack-frames as

$$t_1 \triangleq \mathit{let}\ x{:}T = o_1\ \mathit{blocks\ for}\ o\ \mathit{in}\ t_1'\ \mathit{and}\ t_2 \triangleq \mathit{let}\ x{:}T = o_2\ \mathit{blocks\ for}\ o\ \mathit{in}\ t_2'\ .$$

Then:

$$\begin{array}{ll} \natural\langle \mathit{stop}\rangle & \stackrel{s_1^1}{\Longrightarrow} \\ \natural\langle t_{1\,\mathit{blocked}}^1;\ \mathit{stop}\rangle & \stackrel{s_1^2}{\Longrightarrow} \\ \natural\langle t_{2\,\mathit{blocked}}^2;\ t_{1\,\mathit{blocked}}^1;\ \mathit{stop}\rangle & \stackrel{\gamma_c?}{\Longrightarrow} \\ \natural\langle \mathit{let}\ x{:}T = o_1\ \mathit{blocks\ for}\ o\ \mathit{in}\ t;\ t_{2\,\mathit{blocked}}^2;\ t_{1\,\mathit{blocked}}^1;\ \mathit{stop}\rangle\ . & \end{array} \quad \text{(A.8)}$$

Instead of the derivation shown, the alternative sequence $s_2\ \gamma_c?$ can unavoidably be taken as well, only that in the end configuration, the blocked method bodies of the two incoming calls are stacked in reversed order:

$$\natural\langle \mathit{let}\ x{:}T = o_1\ \mathit{blocks\ for}\ o\ \mathit{in}\ t;\ t_{1\,\mathit{blocked}}^1;\ t_{2\,\mathit{blocked}}^2;\ \mathit{stop}\rangle\ . \quad \text{(A.9)}$$

Note, however, that the sender *of $\gamma_c?$, an environment object, is different in $s_1\ \gamma_c?$ and in $s_2\ \gamma_c?$. This is reflected in Lemma A.2.28 in that sender of an incoming communication is preserved for incoming returns (see part 1b), but not necessarily for incoming calls.*

Now, when the merging is not done by a call but by a return, the situation changes! The return either deblocks t_2 *after* s_1 *(replacing the call-step at the end of (A.8)), or it deblocks* t_1 *after* s_2*. Clearly, the component can react differently two these two situations, since it executes two different pieces of code after the return. Concerning the swapping relation, the two trace are* not *equal, i.e.,* $s_1\ \gamma_r?\not\doteq_\Theta\ s_1\ \gamma_r?$*. In particular, rule* SWAPB$_\Theta$ *does* not *apply, as* $u = \gamma_r?$ *is* not *weakly balanced. Still another way to interpret the example is that the swapping with the trailing return is* not *allowed, which means that the sender of the incoming call must be preserved. For incoming calls, the sender needs not be preserved, as the sender of a call as such is not transmitted; hence, swapping of* s_1 *by rule* SWAPW$_\Theta$ *is allowed for* $s_1\gamma_c?$*.* □

Next we prove that swapping preserves enabledness. As just discussed, swapping does not preserve sender and receiver under all circumstances. The judgment $\Xi_0 \vdash t \rhd a$ of enabledness, however, ignores the sender and receiver of a and depends on the pure call and return structure of t. In Lemma A.2.38 later, we will extend Lemma A.2.30 to deal with enabledness judgments of the form $\Xi_0 \vdash t \rhd o_1 \xrightarrow{a} o_2$, which include information about the communication partners.

Lemma A.2.30 (Swapping and enabledness)**.** *Assume* $t_1 : \mathit{wbalanced}$*. If* $\Xi_0 \vdash t_1 \rhd a$ *for some label* a *and* $\Xi_0 \vdash t_1 \doteq_\Theta t_2$ *then* $\Xi_0 \vdash t_2 \rhd a$*.*

Proof. Enabledness is given in Definition 3.3.3. For one case, we assume $\vdash t_1 : \mathit{wbalanced}^-$ (the one for $\mathit{wbalanced}^+$ is dual). By preservation of balance under swapping from Lemma A.2.23, $\vdash t_2 : \mathit{wbalanced}^-$, as well.

We show that one application of SWAPW$_\Theta$, resp., of SWAPB$_\Theta$, preserves enabledness. The claim follows by transitivity of $\doteq_\Theta$/induction.

The definition of enabledness distinguishes call and return labels. By Lemma A.2.18(5), a must be an incoming communication.

Case: Incoming call: $a = \gamma_c?$
We further distinguish according to the instance of the rules from Table A.1.

Subcase: SWAPW$_\Theta$
According to equation (3.12) of input enabledness for calls, there are two cases to consider. In the first case, $\mathit{pop}\, t_1 = \bot$ and $\Delta_0 \vdash \odot$. Hence by Lemma A.2.13, t_1 is balanced, more precisely, $\vdash t_1 : \mathit{balanced}^-$. By Lemma A.2.23(2), $\vdash t_2 : \mathit{balanced}^-$. Hence, $\Xi_0 \vdash t_2 \rhd a$ by the definition of pop (equation (3.12)).

In the second case, $\mathit{pop}\, t_1 = t_1'\ \gamma_c!$, which means $t_1 = t_1'\ \gamma_c!\ t_1''$ with $\vdash t_1'' : \mathit{balanced}^-$. There are a few cases to distinguish, namely concerning which part(s) of $t_1'\ \gamma_c!\ t_1''$ are affected by $\doteq_\Theta$.

Subsubcase: Swapping inside t_1' or inside t_1''
If $t_2 = t_1'\ \gamma_c!\ t_2''$ with[4] $t_1'' \doteq_\Theta t_2''$, then by preservation of balance under swapping from Lemma A.2.23(2), t_2'' is balanced, as well, and the case follows from the definition of enabledness. The case for $t_2 = t_2'\ \gamma_c!\ t_1''$ with $t_1' \doteq_\Theta t_2'$ is simpler, as the balanced end-sequence t_1'' is not affected by the swapping.

Subsubcase: Swapping neither in t_1' nor in t_1''
Remain the situations where the swapping *moves* $\gamma_c!$. In one case, we have

$$t_1 = r_1'\ \underline{s_1'\gamma_c!r_1''}\ \overline{s_1''}u_1'' \quad \doteq_\Theta \quad r_1'\ \overline{s_1''}\ \underline{s_1'\gamma_c!r_1''}\ u_1'' = t_2\ , \tag{A.10}$$

[4] We write in the proof in situations as the current one $\vdash t_1'' \doteq_\Theta t_2''$ short for $\Xi_0 \vdash t_1'\ \gamma_c!\ t_1'' \doteq_\Theta t_1'\ \gamma_c!\ t_2''$ with $\doteq_\Theta$ changing only the t_1''-part to t_2''.

where the swap (from left to right) of the underlined and the overlined subsequence is justified by rule SWAPW$_\Theta$. By the premise of that rule, the trailing u_1'' is weakly balanced. Since additionally $r_1''\ s_1''\ u_1''$ is balanced, the cut Lemma A.2.10(2) and (3) yields that $r_1''\ s_1''$ and u_1'' must be balanced, as well. By a second premise of rule SWAP$_\Theta$, s_1'' is weakly balanced, hence applying Lemma A.2.10(2) once more gives, that also r_1'' is balanced, from which the result follows. Alternatively, SWAPW$_\Theta$ gives rise to the following situation, reversed compared to (A.10):

$$t_1 = r_1'\ \overline{s_1'}\ \underline{u_1'\ \gamma_c!\ r_1''}\ u_1'' \quad \doteq_\Theta \quad r_1'\ \underline{u_1'\ \gamma_c!\ r_1''}\ \overline{s_1'}\ u_1'' = t_2\ . \tag{A.11}$$

By the premises of the swap-rule, the overlined and the underlined parts s_1' and $u_1'\ \gamma_c!\ r_1''$ are both weakly balanced, and additionally the trailing u_1'' is weakly balanced, as well as. Furthermore, $r_1''\ u_1''$ is balanced, which implies with the cut lemma Lemma A.2.10(2) and (3) that both r_1'' and u_1'' are balanced (in the given situation, $\vdash r_1''\ u_1'' : \mathit{balanced}^-$, $\vdash r_1'' : \mathit{balanced}^-$ and $\vdash u_1'' : \mathit{balanced}^-$).

The sequence s_1', however, is *not* guaranteed to be strictly balanced, only weak balance is assured! If s_1' is balanced, as well, the case follows straightforwardly from the definition of enabledness and pop, since the trace after $\gamma_c!$ on the right-hand side, i.e., $r_1''\ s_1'\ u_1''$, is balanced then.

If s_1' is not balanced, we know at least $\vdash s_1' : {}^-\mathit{wbalanced}^-$ in the given situation (because the trace is alternating, we can conclude, that in particular u_1' is of odd length). By the characterization of weakly balanced traces from Lemma A.2.8, s_1' is of the form $s_1^{1'}\ \gamma_c'!\ s_1^{2'}$ such that $\vdash s_1^{2'} : \mathit{balanced}^-$. By rule B-OO, $\vdash s_1^{2'}\ u_1'' : \mathit{balanced}^-$, from which the case follows, using equation (3.12).

Subcase: SWAPB$_\Theta$
Easier. As in the previous case, there are two cases to consider (see equation (3.12)). The one for $\mathit{pop}\ t_1 = \bot$ works as for SWAPW$_\Theta$. If otherwise, $\mathit{pop}\, t_1 \neq \bot$, the case follows by the easy observation that $\mathit{pop}(s\ u) = \mathit{pop}(s\ t\ u)$ for balanced traces t (where alternation is respected, i.e., $s\ t\ u$ must be alternating). See Lemma A.2.2 and A.2.4.

Case: Incoming return: $a = \gamma_r?$
In this case, return enabledness means that $\mathit{pop}(t_1) = t_1'\ \gamma_c!$, i.e., the argument works analogously to the part for $\gamma_c?$ where $\mathit{pop}\, t_1 \neq \bot$, i.e., to be input enabled, the case where t_1 is balanced cannot occur.

Case: Outgoing communication: $a = \gamma!$
Similar. □

Remark A.2.31 (Swapping and enabledness)**.** *Concerning Lemma A.2.30 and its proof. The lemma shows preservation of $\Xi_0 \vdash t_1 \rhd a$, when t_1 is replaced by t_2 by swapping. The interesting cases are the ones justified by* SWAPW$_\Theta$.

In the situation for incoming calls, described by equation (A.11) in the proof, the subsequence s_1' is weakly balanced, but may not be balanced, as treated in the proof in one subcase. Also in that situation, swapping preserves enabledness *of the incoming call. The* sender *of $\gamma_c?$ however,* changes! *Before the swap, $\mathit{sender}(t_1\ \gamma_c?) = \mathit{receiver}(t_1)$, afterwards, have $\mathit{sender}(t_2\ \gamma_c?) = \mathit{receiver}(t_2)$, which might not be the same.*

The same holds also for returns, i.e., the sender *of $\gamma_r?$ may change (caused by* SWAPW$_\Theta$*, but not by* SWAPB$_\Theta$*). This does not contradict Lemma A.2.28, in partic-*

ular not part 1b, which stipulates that Θ-swapping preserves the sender of incoming returns. In the mentioned situation that $sender(t_1\ \gamma_r?) \neq sender(t_2\ \gamma_r?)$, *where* $\Xi_0 \vdash t_1 \doteq_\Theta t_2$ *and both* $\Xi_0 \vdash t_1 \rhd \gamma_r?$ *and* $\Xi_0 \vdash t_2 \rhd \gamma_r?$. *However,* $\Xi_0 \not\vdash t_1\ \gamma_r? \doteq_\Theta t_2\ \gamma_r?$, *since the condition* $\vdash u : wbalanced$ *for* SWAPB_Θ *does not hold. For calls* $\gamma_c?$, *on the other hand* $\Xi_0 \vdash t_1 \asymp_\Theta t_2$ *implies* $\Xi_0 \vdash t_1\ \gamma_c? \asymp_\Theta t_2\ \gamma_c?$, *even if that might change the sender of* $\gamma_c?$. □

Later we prove in the extension Lemma A.2.40 that $\doteq_\Theta$ is preserved under extending the two compared traces under certain circumstances, but not in general. However, prefixing always preserves $\doteq_\Theta$:

Lemma A.2.32 (Swapping and prefix)**.** *Assume* $\Xi_0 \vdash s\ a : wbalanced$. *If* $\Xi_0 \vdash s\ a \doteq_\Theta t\ a$ *(i.e., where* a *is not affected by the swapping), then* $\Xi_0 \vdash s \doteq_\Theta t$.

Proof. Immediate by inspection of the rules from Table A.1, and the fact that weak balance is preserved under prefixing (see Corollary A.2.6). □

The next two lemmas show preservation of enabledness under swapping, similar to Lemma A.2.30, but additionally taking sender and receiver of the next label into account.

Lemma A.2.33 (Swapping and partners)**.** *Assume* $\Xi_0 \vdash t_1\ \gamma_1?\ \gamma_2! : wbalanced$ *and furthermore* $\Xi_0 \vdash t_1\ \gamma_1? \doteq_\Theta t_2\ \gamma_1?$ *(i.e., the swapping affects only* t_1 *and* t_2*). Then* $sender(t_1\ \gamma_1?\ \gamma_2!) = sender(t_2\ \gamma_1?\ \gamma_2!)$. *The property holds dually for* $\doteq_\Delta$, *i.e., where incoming and outgoing communication is reversed.*

Proof. First, we know stronger that $\Xi_0 \vdash t_1\ \gamma_1?\ \gamma_2! : wbalanced^-$. By preservation of balance under swapping (Lemma A.2.23), $\Xi_0 \vdash t_1\ \gamma_1?\ \gamma_2! : wbalanced^-$, as well. Clearly, this implies for the prefixes $\Xi_0 \vdash t_1\ \gamma_1? \ : wbalanced^+$ and $\Xi_0 \vdash t_2\ \gamma_1? \ : wbalanced^+$ and, in particular,

$$\Xi_0 \vdash t_1\ \gamma_1? \doteq_\Theta t_2\ \gamma_1? \,. \tag{A.12}$$

That weak balance is preserved under prefixing follows from Corollary A.2.6; preservation of $\doteq_\Theta$ under prefixing by Lemma A.2.32. Furthermore we have $sender(t_1\ \gamma_1?\ \gamma_2!) = receiver(t_1\ \gamma_1?)$ and analogously $sender(t_2\ \gamma_1?\ \gamma_2!) = receiver(t_2\ \gamma_1?)$ by (the dualization of) Lemma A.2.14(2).

Equation (A.12) implies with Lemma A.2.28(2b), that the receiver of $\gamma_1?$ is preserved under swapping, i.e., $receiver(t_1\ \gamma_1?) = receiver(t_2\ \gamma_1?)$, independent of whether the label is a call or a return. Therefore, $sender(t_1\ \gamma_1?\ \gamma_2!) = sender(t_2\ \gamma_1?\ \gamma_2!)$, as required for the sender of $\gamma_2!$. □

In the legal trace system of Table 3.5, the conditions to extend a trace t by an additional label a to $t\ a$ can be split into three conditions (resp., into four conditions in the deterministic setting): (1) *enabledness*, i.e., whether alternation between incoming and outgoing communication is respected and whether, in case of a return, it is an answer to a matching call (cf. Definition 3.3.3). The enabledness condition is combined in judgments of the form $\Xi_0 \vdash t \rhd o_s \xrightarrow{a} o_r : \vec{T} \to T$ with the calculation of the sender and receiver and the calculation of the expected types (see equation (3.14)). Then (2) *typing*, i.e., basically that the transmitted values are of the expected types. In Definition 2.6.11, this is formulated in the context after *updating* the context before the label with the

fresh information carried by the label. Finally, (3) the connectivity information is checked. Part (2) and (3) use the sender and receiver plus the type calculated in (1) for the check. In the deterministic setting, additionally (4) a condition ensuring determinism is required. The next definition combines the mentioned conditions into a single judgment, for convenience.

Definition A.2.34 (Legality)**.** *In context Ξ_0 and after trace t, the next label a is legal, written*

$$\Xi_0 \vdash t \rhd a : ok, \tag{A.13}$$

if

1. $\Xi_0 \vdash t \rhd o_s \xrightarrow{a} o_r : \vec{T} \to T$ *for some o_s and o_r (cf. Definition 3.3.3 on page 57)*
2. $\acute{\Delta}, \acute{\Theta} \vdash \lfloor a \rfloor : \vec{T} \to \_$*, resp.,* $\acute{\Delta}, \acute{\Theta} \vdash \lfloor a \rfloor : \_ \to T$*, depending on whether a is a call, resp., a return and with $\vec{T}$ and T determined by part 1 (cf. Definition 2.6.11 on page 36, equation (2.16)).*
3. $\acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} \_ : wc$ *(cf. Definition 2.6.7 on page 34, equation (2.10)).*
4. $\Xi \vdash t \rhd a : det$ *(cf. Definition 3.1.10 on page 52).*

In part 2 and 3, $\acute{\Xi}$ is given by $\Xi_0 \xRightarrow{t} \Xi$ and $\acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r$ (see equation (2.13), combining the context updates from Definition 2.6.8 and 2.6.9).

Lemma A.2.35. *Assume $\Xi_0 \vdash t\ a : trace$, then $\Xi_0 \vdash t \rhd a : ok$.*

Proof. Straightforward, as the definition of $\Xi_0 \vdash t \rhd a : ok$ collects the premises used to check $\Xi_0 \vdash t\ a : trace$, given a proof of $\Xi_0 \vdash t : trace$. By Definition A.2.34, there are four conditions to be checked. The enabledness of part A.2.34 of the definition is covered by Lemma A.2.20. Well-typedness and well-connectedness for part 2 and 3 are covered by the respective premises of the rule for legality applied to derive $\Xi_0 \vdash t\ a : trace$. Part 4 for determinism is immediate. □

The next lemma shows that two traces are swapping equal using the tree representation if they are swapping equal using the equational representation. The reverse direction is proven later in Lemma A.2.44.

Lemma A.2.36 ($\doteq_\Theta$ implies $\asymp_\Theta$)**.** *Assume $\vdash s : wbalanced^-$ and $\vdash t : wbalanced^-$. If $\Xi_0 \vdash s \doteq_\Theta t$, then $\Xi_0 \vdash s \asymp_\Theta t$. The property holds analogously for $wbalanced^+$ and dually for $\doteq_\Delta$ and $\asymp_\Delta$.*

Proof. We show the implication for one application of a rule for $\doteq_\Theta$. The result then follows by induction/transitivity.

Proceed by induction on the length of s. Note first that s and t are of equal length. Furthermore, using the alternation Lemma A.2.1, if $\vdash s : balanced^-$, then $\vdash t : balanced^-$ (and analogously for $balanced^+$). The base case for $s = \epsilon$, and hence $t = \epsilon$, is immediate with reflexivity of $\asymp_\Theta$. For the induction step we distinguish according to the last label of s.

Case: Input call: $s = s'\ \gamma_c? \doteq_\Theta t$
We distinguish whether the swap-step $\doteq_\Theta$ affects the end label $\gamma_c?$ or not.

Subcase: $\Xi_0 \vdash s'\ \gamma_c? \dot{\asymp}_\Theta\ t'\ \gamma_c?$, with $\Xi_0 \vdash s' \dot{\asymp}_\Theta t'$
The shorter traces are weakly balanced, as well (Corollary A.2.6). Thus by induction, $\Xi_0 \vdash s' \asymp t'$. By the preservation of communication partners from Θ under swapping from Lemma A.2.28(2b), $receiver(s'\ \gamma_c?) = receiver(t'\ \gamma_c?)$ (the receiver, i.e., the callee, is a component object). I.e., the label $\gamma_c?$ belongs to the same component clique in s' and in t'. Therefore, $\Xi_0 \vdash s' \asymp_\Theta t'$ implies $\Xi_0 \vdash s'\ \gamma_c? \asymp_\Theta t'\ \gamma_c?$.

Subcase: $s = s'\ \gamma_c? = r'\ s_1'\ s_2'\ \gamma_c?$ and $t = r'\ s_2'\ \gamma_c?\ s_1'$
This case cannot happen (unless $s_1' = \epsilon$, in which case the claim is trivial). Since t is alternating, s_1' must start with an outgoing label. This, however, contradicts $\Xi_0 \vdash r \rhd s_1' \not\leftrightharpoons s_2'\ \gamma_c?$.

Case: Input return: $s = s'\ \gamma_r? \dot{\asymp}_\Theta t'$
As for incoming calls, the case where the swap affects $\gamma_r?$ implies that the case is trivial, i.e., $s = t$. I.e., we have to consider only the following situation:

Subcase: $\Xi_0 \vdash s'\ \gamma_r? \dot{\asymp}_\Theta t'\ \gamma_r?$, with $\Xi_0 \vdash s' \dot{\asymp}_\Theta t'$
The case works similar to the corresponding one for incoming calls. The shorter traces s' and t' are weakly balanced, as well. Therefore, by induction, $\Xi_0 \vdash s' \asymp_\Theta t'$. Lemma A.2.28(2b) gives that also for returns $receiver(s'\ \gamma_r?) = receiver(t'\ \gamma_r?)$, a component object. Therefore, $\Xi_0 \vdash s'\ \gamma_r? \asymp_\Theta t'\ \gamma_r?$, as required.

Case: Output call: $s = s'\ \gamma_c!$
We distinguish whether the swap-step $\dot{\asymp}_\Theta$ affects the end label $\gamma_c!$ or not.

Subcase: $\Xi_0 \vdash s'\ \gamma_c! \dot{\asymp}_\Theta t'\ \gamma_c!$, with $\Xi_0 \vdash s' \dot{\asymp}_\Theta t'$
Induction yields $\Xi_0 \vdash s' \asymp t'$. Lemma A.2.28(2a) gives that the *sender* object of the call, a component object is not affected by the swap, i.e., the $\gamma_c!$ belongs to the same component clique, comparing s' and t', and hence $\Xi_0 \vdash s'\ \gamma_c! \asymp_\Theta t'\ \gamma_c!$, as required.

Subcase: $s = s'\ \gamma_c! = r'\ s_1'\ s_2'\ \gamma_c!$ and $t = r'\ s_2'\ \gamma_c!\ s_1'$
Unlike the situation for incoming communication we cannot argue away this case. However, the premises for both SWAPB$_\Theta$ and SWAPW$_\Theta$ require that $\Xi_0 \vdash_\Theta r \rhd s_1' \not\leftrightharpoons s_2'\ \gamma_c!$, from which the case follows by definition of $\asymp_\Theta$.

Case: Output return: $s = s'\ \gamma_r!$
Analogous to the case for outgoing calls. Also for outgoing return, Lemma A.2.28(2a) assures that the sender of the return is not affected by the swapping. □

Lemma A.2.37 (Swapping and contexts)**.** *If* $\Xi_0 \overset{s}{\Longrightarrow} \Xi_1$ *and* $\Xi_0 \vdash s \dot{\asymp}_\Theta t$*, then* $\Xi_0 \overset{t}{\Longrightarrow} \Xi_2$*, where* $(\Delta_1, \Theta_1) = (\Delta_2, \Theta_2)$ *and* $E_{\Theta_1} = E_{\Theta_2}$*. The property holds dually for* $\dot{\asymp}_\Delta$*, where* $E_{\Delta_1} = E_{\Delta_2}$ *instead.*

Proof. Straightforward, using in particular the preservation of communication partners under $\dot{\asymp}_\Theta$ from Lemma A.2.28(2), and the definition of connectivity update (Definition 2.6.9, in particular, part 1 for incoming communication and the *dual* of part 2, both updating E_Θ). □

The next lemma shows that the judgment $\Xi_0 \vdash s \rhd a : ok$ is preserved under swapping according to the cliques of Θ, provided, s ends in an *incoming* communication.

Lemma A.2.38 (Swapping). *Assume* $\vdash s'\ \gamma'? : wbalanced$ *and* $\Xi_0 \vdash s'\ \gamma'? \doteq_\Theta t$.

1. *If* $\Xi_0 \vdash s'\ \gamma'? \rhd o_s \xrightarrow{\gamma!} \_$, *then* $\Xi_0 \vdash t \rhd o_s \xrightarrow{\gamma!} \_$. *The "*$\_$*" is a place holder, indicating in particular, that the receiver might not be preserved. In short:*

$$\frac{\vdash s'\ \gamma'? : wbalanced \quad \Xi_0 \vdash s'\ \gamma'? \rhd o_s \xrightarrow{\gamma!} \_ \quad \Xi_0 \vdash s'\gamma'? \doteq_\Theta t}{\Xi_0 \vdash t \rhd o_s \xrightarrow{\gamma!} \_}$$

2. *If* $\Xi_0 \vdash s'\ \gamma'? \rhd \gamma! : wt$, *then* $\Xi_0 \vdash t \rhd \gamma! : wt$.
3. *If* $\Xi_0 \vdash s'\ \gamma'? \rhd \gamma! : wc$, *then* $\Xi_0 \vdash t \rhd \gamma! : wc$.
4. *If* $\Xi_0 \vdash s'\ \gamma'? \rhd \gamma! : det$, *then* $\Xi_0 \vdash t \rhd \gamma! : det$.

The properties hold dually for $\doteq_\Delta$ *and for* $\gamma'!$ *instead of* $\gamma'?$.

Proof. There are four parts to show; in the multithreaded setting, the condition dealing with determinism is not needed, but the implication would hold nonetheless.

Part 1 (enabledness and sender)
By preservation of enabledness under swapping from Lemma A.2.30,

$$\Xi_0 \vdash t \rhd \gamma! \,. \tag{A.14}$$

It remains to be checked, that the sender is preserved, as well. First, we know stronger that $\vdash\ s'\ \gamma'? : wbalanced^+$, since weakly balanced traces are alternating (Lemma A.2.1). By preservation of balance under swapping (Lemma A.2.23), $\Xi_0 \vdash t : wbalanced^+$, as well. As t is alternating, as well, again with Lemma A.2.1, t is of the form $t'\ \gamma'?$ (with $\Xi_0 \vdash s' \doteq_\Theta t'$). Hence, Lemma A.2.33 applies, yielding the result.

For the next two parts, let Ξ be given by $\Xi_0 \overset{s'\ \gamma'?}{\Longrightarrow} \Xi_1$, and $\acute{\Xi}_1 = \Xi_1 + o_s \xrightarrow{\gamma!} o_r$, as given by equation (2.13), combining Definition 2.6.8 and 2.6.9.

Part 2 (typing)
Well-typedness of a label is given in Definition 2.6.11 on page 36. We need to distinguish, where $\gamma!$ is a call or a return, according to the two rules of Table 2.10, more precisely, the duals of the two rules.

Case: LT-CALLO
First, the receiver of the call is preserved by swapping. Secondly, $\Xi_0 \overset{t}{\Longrightarrow} \Xi_2$ implies with Lemma A.2.37 that $(\Delta_2, \Theta_2) = (\Delta_1, \Theta_1)$, and furthermore $(\acute{\Delta}_2, \acute{\Theta}_2) = (\acute{\Delta}_1, \acute{\Theta}_1)$. Therefore, Definition 3.3.5 determines the same expected type and the premises of rule LT-CALLO apply unchanged also for the situation after t.

Case: LT-RETO
Similarly, plus the fact from part 1 of the lemma that the *sender* is preserved. The receiver, which in contrast might not be preserved, is not relevant for LT-RETO.

Part 3 (connectivity)
See Definition 2.6.7 for the check of connectivity, where we need the dual of equation (2.10), i.e., for *outgoing* core labels. The connectivity check is based on the *sender,* only. Thus, the part works analogously to part 2 for typing and rests again on preservation of the sender from part 1.

Part 4 (determinism)
See Definition 3.1.10 for the definition of deterministic extension. Note first, that the lemma is dealing with the extension by an outgoing label and hence we need to check only for preservation of det_Θ, not for det_Δ, which is preserved automatically. Let us abbreviate $s'\ \gamma'!$ by s. So we are given one of two possible situations (cf. equation (3.11)). If $\Xi_0 \vdash s\ \gamma!\ \preccurlyeq_\Theta\ s$, we argue as follows. By assumption, $\Xi_0 \vdash s \doteq_\Theta t$. Hence by Lemma A.2.36, $\Xi_0 \vdash s \asymp_\Theta t$, i.e., by the tree-based definition of swapping (Definition 3.1.7), $\underline{s} = \underline{t}$, which further means that s and t are equal when projected to the behavior of all mentioned component objects. Since the sender of $\gamma!$ is preserved under swapping by part 1, also $\Xi_0 \vdash t\ \gamma! \preccurlyeq_\Theta t$, as required. The alternative form (3.11)), that there does not exists a label b with $\Xi_0 \vdash s\ \gamma! \preccurlyeq_\Theta s$, works analogously. □

Corollary A.2.39. *Assume* $\vdash s'\ \gamma'? : wbalanced$ *and* $\Xi_0 \vdash s'\ \gamma'? \doteq_\Theta t$. *If* $\Xi_0 \vdash s'\ \gamma'? \rhd \gamma! : ok$, *then* $\Xi_0 \vdash t \rhd \gamma! : ok$.

Proof. Directly by Lemma A.2.38. □

Lemma A.2.32 showed that swapping is preserved under prefixing. The reverse preservation, under extension, does not hold in general. The intuitive reason is, that in particular extending the trace by a *merging* action may reveal information about the order of interaction in the past, which has not been observable as long as the cliques had been separate. The reconstruction of past orderings by merging, however, is not in general possible. In particular, it is not possible if them merging is done by a call. More technically, extending two traces by an additional label may break $\doteq_\Theta$ (and dually $\doteq_\Delta$), since extending a trace may break weak balance, namely when adding a *return*. This additional return may invalidate the corresponding premise from SWAPW$_\Theta$. See, also the informal discussion at the beginning of Section A.2.2.

Lemma A.2.40 (Swapping and extension)**.**

1. *Assume* $\vdash s\ \gamma_c? : wbalanced^+$. *If* $\Xi_0 \vdash s \doteq_\Theta t$, *then* $\Xi_0 \vdash s\ \gamma_c? \doteq_\Theta t\ \gamma_c?$.
2. *Assume* $\vdash s\ \gamma_r? : wbalanced^+$. *If* $\Xi_0 \vdash s \doteq_\Theta t$ *and* $receiver(s\ \gamma_r?) = receiver(t\ \gamma_r?)$, *then* $\Xi_0 \vdash s\ \gamma_r? \doteq_\Theta t\ \gamma_r?$.
3. *Assume* $\vdash s\ \gamma! : wbalanced^-$. *If* $\Xi_0 \vdash s \doteq_\Theta t$, *then* $\Xi_0 \vdash s\ \gamma! \doteq_\Theta t\ \gamma!$.

The three statements[5] *are summarized by the rules of Table A.2. The property holds dually for* $\doteq_\Delta$.

Proof. We show that the property holds for a single application of one of the rules from Table A.1. The result follows by transitivity/induction.

Rule SWAP$^1_\Theta$*: Extension by an incoming call*
First note that $\Xi_0 \vdash t\ \gamma_c? : wbalanced^+$ (by preservation of weak balance under swapping from Lemma A.2.23). There are two cases to distinguish.

[5]It is not necessary, for instance in the first statement to assume that $\vdash s\ \gamma_c? : wbalanced^+$, it would suffice to require $\vdash s\ \gamma_c? : wbalanced$; analogously in the other two parts. The polarity is spelled out for clarity.

$$\frac{\Xi_0 \vdash s \dot{\asymp}_\Theta t}{\Xi_0 \vdash s\ \gamma_c? \dot{\asymp}_\Theta t\ \gamma_c?}\ \textsc{Swap}^1_\Theta$$

$$\frac{\Xi_0 \vdash s \dot{\asymp}_\Theta t \qquad \mathit{receiver}(s\ \gamma_r?) = \mathit{receiver}(t\ \gamma_r?)}{\Xi_0 \vdash s\ \gamma_r? \dot{\asymp}_\Theta t\ \gamma_r?}\ \textsc{Swap}^2_\Theta$$

$$\frac{\Xi_0 \vdash s \dot{\asymp}_\Theta t}{\Xi_0 \vdash s\ \gamma! \dot{\asymp}_\Theta t\ \gamma!}\ \textsc{Swap}^3_\Theta$$

Table A.2: Swapping and extension

Case: SWAPW$_\Theta$: $\Xi_0 \vdash s = r\ (s_1\ s_2)\ u \dot{\asymp}_\Theta r\ (s_2\ s_1)\ u = t$,
where $\vdash u : \mathit{wbalanced}^-$ (by Lemma A.2.18(5)). Clearly, extending the weakly balanced u by a call preserves weak balance (Lemma A.2.7). Hence, $\vdash u\ \gamma_c? : \mathit{wbalanced}^+$, from which the claim follows.

Case: SWAPB$_\Theta$: $\Xi_0 \vdash s = r\ (s_1\ s_2)\ u \dot{\asymp}_\Theta r\ (s_2\ s_1)\ u = t$,
with s_1 (or s_2) balanced. The claim $\Xi_0 \vdash s\ \gamma_c? \asymp_\Theta t\ \gamma_c?$ is justified directly by SWAPB$_\Theta$.

Rule SWAP$_2$*: Extension by an incoming return*
Analogous to the previous case, preservation of enabledness under swapping from Lemma A.2.23 gives $\Xi_0 \vdash t\ \gamma_r? : \mathit{wbalanced}^+$.

Case: SWAPB$_\Theta$: $\Xi_0 \vdash s = r\ (s_1\ s_2)\ u \dot{\asymp}_\Theta r\ (s_2\ s_1)\ u = t$,
where s_1 or s_2 is balanced. The case follows directly by SWAPB$_\Theta$. Note that in this case of SWAPB$_\Theta$, the premise of SWAP$^2_\Theta$ requiring $\mathit{receiver}(s\ \gamma_r?) = \mathit{receiver}(t\ \gamma_r?)$ is not a restriction; swapping of a balanced subsequence preserves the receiver of the return (cf. the part of Lemma A.2.28 dealing with strictly balanced sub-sequences).

Case: SWAPW$_\Theta$: $\Xi_0 \vdash s = r\ (s_1\ s_2)\ u \dot{\asymp}_\Theta r\ (s_2\ s_1)\ u = t$,
where $\vdash u : \mathit{wbalanced}^-$ (by alternation), the only interesting case. Unlike in the case for incoming calls, we cannot immediately conclude that $\vdash u\ \gamma_r? : \mathit{wbalanced}^+$. By definition of the receiver and the *pop*-function (Definition 3.3.4 and 3.3.1), $\mathit{receiver}(s\ \gamma_r?) = \mathit{sender}(s'\ \gamma_c!)$, where $s = s'\ \gamma_c!\ s''$ and where s'' is the uniquely determined postfix of s with $\vdash s'' : \mathit{balanced}^-$. We distinguish whether $\gamma_c!$ is part of r, of s_1, of s_2, or of u, yielding 4 cases:

1. $s = s'\ \gamma_c! s'' = (r'\ \gamma_c!\ r'')\ s_1\ s_2\ u$, with s_1, s_2, and u balanced.
2. $s = s'\ \gamma_c! s'' = r\ (s_1'\ \gamma_c!\ s_1'')\ s_2\ u$, with s_2 and u balanced.
3. $s = s'\ \gamma_c! s'' = r\ s_1\ (s_2'\ \gamma_c!\ s_2'')\ u$ with u balanced.
4. $s = s'\ \gamma_c!\ s'' = r\ s_1\ s_2'\ (u'\ \gamma_c!\ u'')$.

The fact that in the different case the trailing sub-sequence are balanced follows by the cut Lemma A.2.10 from the premises of rule SWAPW$_\Theta$. In case 1, the claim follows by the swapping rule SWAPB$_\Theta$ for a *balanced* subsequence. Similar in case 2, since again s_2 is balanced.

Interesting is case 3. It is the only case where we use the additional premise $\mathit{receiver}(r\ s_1\ s_2\ u\ \gamma_r?) = \mathit{receiver}(r\ s_2\ s_1\ u\ \gamma_r?)$, which is the sender of $\gamma_c!$, a component object. We additionally know by the last premise of SWAPW$_\Theta$ that $\Xi_0 \vdash_\Theta r\ \rhd\ s_1 \not\leftrightharpoons s_2$. This implies that for the swapped sequence we have

$$t = r\ s_2\ s_1\ u = r\ s_2'\ \gamma_c!\ s_2''\ s_1\ u\ .$$

The cut Lemma A.2.10 implies again that s_1 and u are balanced, and thus the case follows once more by SWAPB$_\Theta$. Case 4 finally follows by directly by SWAPW$_\Theta$, since $\vdash u\ \gamma_r? : \mathit{wbalanced}^+$.

Rule SWAP$^3_\Theta$*: Extension by an outgoing communication*
We are given that $s\ \gamma!$ is weakly balanced and thus alternating. The means, s is either empty or ends in an incoming communication. For $s = \epsilon$, the case is immediate. Otherwise, $s = s'\ \gamma?$. Independent of whether $\Xi_0 \vdash s \doteq_\Theta t$ is justified by SWAPB$_\Theta$ or SWAPW$_\Theta$, the traces are of the form $r\ s_1\ s_2\ u$, resp., $r\ s_2\ s_1\ u$, where $u = u'\ \gamma?$. In case of SWAPW$_\Theta$, the fact that $\vdash u : \mathit{wbalanced}^+$ implies that also $u\ \gamma!$ is weakly balanced (more precisely, $\vdash u\ \gamma! : \mathit{wbalanced}^-$) even in the case that $\gamma! = \gamma_r!$. In case of SWAPB$_\Theta$, the claim follows straightforward by SWAPB$_\Theta$ and the observation that the last interaction (being incoming) of s, resp., of t, cannot be affected by the swapping. □

Note that in SWAP$^1_\Theta$, the *sender* of $\gamma_c?$ may be different after s and after t. This can be understood that a call in isolation makes no observable difference, since the sender of the call is not transmitted; see Lemma A.2.28, where preservation of the sender of an incoming call is *not* assured. Note further that the preservation of the sender (or receiver) does not hold for returns (which why it is explicitly required in SWAP$^2_\Theta$), i.e., it is possible that $\Xi_0 \vdash s\ \doteq_\Theta\ t$ but $\Xi_0 \not\vdash s\ \gamma_r? \doteq_\Theta\ t\ \gamma_r?$. The smallest illustrating example are the two traces

$$s = s_1\ s_2 = \gamma_c^1?\ \gamma_c^{1\prime}!\ \ \gamma_c^2?\ \gamma_c^{2\prime}! \quad \text{and} \quad t = s_2\ s_1 = \gamma_c^2?\ \gamma_c^{2\prime}!\ \ \gamma_c^1?\ \gamma_c^{1\prime}!\ .$$

The equation $s \doteq_\Theta t$ is justified by SWAPW$_\Theta$, but not by SWAPB$_\Theta$, with the trailing u in the premise of the rule empty. As mentioned for the call, the sender of and additional incoming return changes comparing the situation after s with the one after t. The difference between extending the trace by a call, resp., by a return, is that the trailing u, extended by the additional interaction, remains *weakly balanced* in isolation when adding a call, whereas with the additional return it might not. Consequently, SWAPW$_\Theta$ applies to $s\ \gamma_c?$ but not necessarily to $s\ \gamma_r?$. Since the swapping with the additional $\gamma_r?$ is (in certain cases) *not* possible, the sender of an incoming return is *preserved* by swapping (see Lemma A.2.28(1b)), in contrast to the sender of incoming calls.

The next lemma is a straightforward extension of Lemma A.2.40 for the preservation of swapping under an extension by a trace longer than a single label.

Lemma A.2.41 (Swapping and extension)**.** *Assume* $\Xi_0 \vdash s\ u : \mathit{wbalanced}$, $\Xi_0 \vdash t\ u : \mathit{wbalanced}$, *and* $\Xi_0 \vdash u : \mathit{wbalanced}$. *If* $\Xi_0 \vdash s \doteq_\Theta t$, *then* $\Xi_0 \vdash s\ u \doteq_\Theta t\ u$

Proof. By straightforward induction on the length of u, using Lemma A.2.40 in the induction step and with the help Lemma A.2.28(2b) which assures that the receiver for incoming returns remains unchanged by the swapping. □

The next lemma allows to reorder the labels in a trace by swapping in such a way that the ones belonging to a chosen clique are grouped together, uninterrupted by labels interacting with other cliques. This will be helpful when proving equivalence of the tree representation $\asymp_\Theta$ and the equational representation $\dot{\asymp}_\Theta$ of the swapping and replay relation. The lemma chooses to shift all interaction of a given clique to the end of the global trace (which is the form helpful in later lemmas). By the side conditions for weakly balanced, especially alternating, traces, and assuming that the property is formulated for Θ-cliques, one cannot move the globally first interaction to the end, if the thread starts at the component side. Similarly, if the trace ends in an incoming label, i.e., it is $wbalanced^+$, entering some component clique, one cannot move the interactions of *another* clique to the end to the trace.

Lemma A.2.42. *Assume* $\Delta_0 \vdash \odot$.

1. *Let* $\Xi_0 \vdash t\ \gamma! : wbalanced$ *and furthermore* $\Xi_0 \overset{t\ \gamma!}{\Longrightarrow} \Xi$ *and* $[o']$ *be an arbitrary component clique after* $t\ \gamma!$*, i.e.,* $\Theta \vdash o'$ *and* $[o'] = [o']_{/\Xi}$*. Assume further* $t_2' = t\ \gamma! \downarrow_{[o']}$*. Then*
$$\Xi_0 \vdash t\ \gamma! \mathrel{\dot{\asymp}_\Theta} t_1'\ t_2'\ , \tag{A.15}$$
for some trace t_1'.

2. *For* $\Xi_0 \vdash t\ \gamma? : wbalanced$*, the property holds only for the component clique of* $\gamma?$*, i.e. for* $[o]_{/\Xi}$*, where* $receiver(t\ \gamma?) = [o]$.

If $\Theta_0 \vdash \odot$*, the trace* $t\ a$ *more precisely is of the form* $\gamma'!\ t\ a$ *or just* $\gamma'!$*, where* $\gamma'!$ *is the initial interaction of the trace (a call). Then, equation (A.15) is adapted to* $\Xi_0 \vdash \gamma'!\ t\ \gamma! \mathrel{\dot{\asymp}_\Theta} \gamma'!\ t_1'\ t_2'$ *(i.e., the initial interaction cannot be moved by swapping).*

The lemma holds dually for $\dot{\asymp}_\Delta$.

Proof. Proceed by induction on the length of t. For $t = \epsilon$, the statement holds vacuously.

Case: $t = s\ \gamma?$

The case corresponds to part 2, where we need to consider the clique $[o]$ of $\gamma?$. If the additional $\gamma?$ creates a new component clique, the result is immediate. Otherwise, we know about the form of the trace:

$$s\ \gamma? = s_1\ s_2\ s_3\ s_4\ \gamma?\ . \tag{A.16}$$

The shorter trace $s_1\ s_2\ s_3$, if not empty, ends in an outgoing communication, as s_3 and $s_4\ \gamma?$ belong to different component cliques. Thus we can apply the induction hypothesis of part 1, reordering the shorter $s_1\ s_2\ s_3$ as follows (if $s_1\ s_2\ s_3$ is empty, we are already done)

$$\Xi_0 \vdash s_1\ s_2\ s_3 \mathrel{\dot{\asymp}_\Theta} s_1'\ s_2'\ , \tag{A.17}$$

where s_2' contains the complete interaction with the component clique $[o]$ of the last incoming call. Since the projection s_2' is weakly balanced (Lemma A.3.3), This implies with the prefix Lemma A.2.32 and the extension Lemma A.2.41 that

$$\Xi_0 \vdash s_1\ s_2\ s_3\ s_4\ \gamma? \mathrel{\dot{\asymp}_\Theta} s_1'\ s_2'\ s_4\ \gamma?\ , \tag{A.18}$$

as required.

Case: $t = s\ \gamma!$
In part 1 for output, we need to consider all component cliques after t, i.e., the sender clique $[o]$ of $\gamma!$ as well as other cliques.

Subcase: sender clique $[o]$ of $\gamma!$
If $s\ \gamma!$ is of the form $s_1\ s_2\ \gamma!$ where $s_2\ \gamma! = s\ \gamma!\downarrow_{[o]}$, the case is immediate. Otherwise, the trace is of the form

$$s\ \gamma! = s_1\ s_2\ s_3\ s_4\ \gamma!\,, \tag{A.19}$$

i.e., it corresponds to the one from equation (A.16), and the case follows analogously.

Case: clique $[o'] \neq [o]$
The trace is of the form $s\ \gamma! = r\ u\ \gamma!$, where $u\ \gamma!$ is the last interaction with $[o]$. The shorter trace r ends in an outgoing communication. Hence, by induction on part 1, we obtain

$$\Xi_0 \vdash r \mathrel{\dot{\asymp}_\Theta} r'_1\ r'_2\,. \tag{A.20}$$

Furthermore, r'_2 and $u\ \gamma!$ are weakly balanced (see the projection Lemma A.3.3), i.e., $\vdash r'_2 : {}^{-}\mathit{wbalanced}^{-}$ and $u\ \gamma! : {}^{-}\mathit{wbalanced}^{-}$, and the result follows with SWAPW$_\Theta$ from Table A.1. □

Lemma A.2.43. *Assume* $\Xi_0 \vdash t'_1\ \gamma? : \mathit{wbalanced}$ *and* $\Xi_0 \vdash t'_1\ \gamma? : \mathit{wbalanced}$. *Furthermore* $\Xi_0 \vdash t'_1\ \gamma? \asymp_\Theta t'_2\ \gamma?$. *Then* $\mathit{receiver}(t'_1\ \gamma?) = \mathit{receiver}(t'_2\ \gamma?)$.

Proof. Straightforward by definition of $\asymp_\Theta$ (Definition 3.1.7), which requires that $\underline{t'_1\ \gamma?} = \underline{t'_2\ \gamma?}$. □

The next two lemma cover the reverse direction of the property from Lemma A.2.36, both together showing that the tree representation and the equational representation of the swapping relation coincide.

Lemma A.2.44 ($\asymp_\Theta$ implies $\mathrel{\dot{\asymp}_\Theta}$)**.** *Assume* $\vdash s : \mathit{wbalanced}$ *and* $\vdash t : \mathit{wbalanced}$. *If* $\Xi_0 \vdash s \asymp_\Theta t$, *then* $\Xi_0 \vdash s \mathrel{\dot{\asymp}_\Theta} t$. *The property holds dually for* $\mathrel{\dot{\asymp}_\Delta}$ *and* $\asymp_\Delta$.

Proof. Proceed by induction on the length of s. The base case for $s = \epsilon$ is immediate by reflexivity of $\mathrel{\dot{\asymp}_\Theta}$. In the induction case we distinguish according to the nature of the last interaction in $s = s'\ a$.

Case: Incoming call: $s = s'\ \gamma_c? \asymp_\Theta t$
The trace t is weakly balanced and thus alternating by Lemma A.2.1. Hence $t = t'\ \gamma_c?$, i.e., the last incoming call cannot be swapped inside. Since the receiver of $\gamma_c?$ concerns the same clique after s' and t', the definition of $\asymp_\Theta$ implies that also for the shorter traces we have $\Xi_0 \vdash s' \asymp_\Theta t'$. By induction, therefore, $\Xi_0 \vdash s' \mathrel{\dot{\asymp}_\Theta} t'$. Furthermore, as weak balance is preserved under prefixing (Lemma A.2.5), $\Xi_0 \vdash s' : \mathit{wbalanced}^{+}$ and $\Xi_0 \vdash t' : \mathit{wbalanced}^{+}$. By the extension Lemma A.2.40, rule SWAP$^1_\Theta$, $\Xi_0 \vdash s'\ \gamma_c? \mathrel{\dot{\asymp}_\Theta} t'\ \gamma_c?$, as required.

Case: Incoming return: $s = s'\ \gamma_r?$
Again, since t is alternating, we have $t = t'\ \gamma_r?$ for some t'. Furthermore, by Lemma A.2.43 $\mathit{receiver}(s'\ \gamma_r?) = \mathit{receiver}(t'\ \gamma_r?)$. As in the previous subcase, this implies also for the shorter $\Xi_0 \vdash s' \asymp_\Theta t'$ and thus by induction, $\Xi_0 \vdash s' \mathrel{\dot{\asymp}_\Theta} t'$. The case follows then by SWAP$^2_\Theta$ from the extension Lemma A.2.40.

Case: Outgoing communication: $s = s'\ \gamma!$
We distinguish the case where there exists *one* component clique after s, or more than one. Since s is not empty, there exists at least one clique. First note that $\Xi_0 \vdash s \asymp_\Theta t$ implies that the number of component cliques implies after t coincides with the number of component cliques after s.

Subcase: One component clique after s and after t
Now, since we have only one component clique after s and after t, we know further that $t = t'\ \gamma!$, i.e., the $\gamma!$ must occur at the end of t. Furthermore we know that $sender(s'\ \gamma!) = sender(t'\ \gamma!)$. Hence, $\Xi_0 \vdash s' \asymp_\Theta t'$ by definition of $\asymp_\Theta$ and thus by induction $\Xi_0 \vdash s' \dot{\asymp}_\Theta t'$. The case then follows by $\textsc{Swap}^3_\Theta$.

Subcase: More than one component clique after s and after t
So consider the component clique of $sender(s) = sender(s'\ \gamma!)$, say $[o]$. Consider the complete interaction of s with that clique, resp., the rest, i.e., let $s_1 = s'\ \gamma!\downarrow_{[o]}$, and s_2. Analogously, $t_1 = s'\ \gamma!\downarrow_{[o]}$, and t_2. By induction, $\Xi_0 \vdash s_1 \dot{\asymp}_\Theta t_1$ and $\Xi_0 \vdash s_2 \dot{\asymp}_\Theta t_2$. As the mentioned projections are all weakly balanced (Lemma A.3.3)), the extension Lemma A.2.41 applies, yielding $\Xi_0 \vdash s_2\ s_1 \dot{\asymp}_\Theta t_2\ t_1$. Then the result follows by (twice) Lemma A.2.42 and transitivity of $\dot{\asymp}_\Theta$:

$$\Xi_0 \vdash s \dot{\asymp}_\Theta s_2\ s_1 \dot{\asymp}_\Theta t_2\ t_1 \dot{\asymp}_\Theta t\ .$$

□

Corollary A.2.45. *Assume s and t being weakly balanced. Then $\Xi_0 \vdash s \dot{\asymp}_\Theta t$ iff $\Xi_0 \vdash s \asymp_\Theta t$.*

Proof. The conjunction of Lemma A.2.36 and A.2.44. □

Next a straightforward observation, connecting the swapping relations $\asymp_\Delta$ and $\asymp_\Theta$ (resp. $\dot{\asymp}_\Delta$ and $\dot{\asymp}_\Theta$) from the perspective of the environment and from the component. See Section 3.1 for the definition of $\bar{t}$, the complementary trace of t.

Lemma A.2.46 (Dualizing)**.** *If $\Xi_0 \vdash s \asymp_\Theta t$, then $\Xi_0 \vdash \bar{s} \asymp_\Delta \bar{t}$. The same holds for the relationship of $\dot{\asymp}_\Theta$ and $\dot{\asymp}_\Delta$.*

Proof. Obvious. All definitions leading to $\asymp_\Theta$ are dually defined for $\asymp_\Delta$. Analogously for $\dot{\asymp}_\Theta$ and $\dot{\asymp}_\Delta$. □

Lemma A.2.47 (Projection and dualizing)**.** *Assume two legal traces s and t, i.e., $\Xi_0 \vdash s : trace$ and $\Xi_0 \vdash t : trace$. Let furthermore $[o]$ be an arbitrary component clique after s and after t, i.e., $\Xi_0 \overset{s}{\Longrightarrow} \Xi_1$, such that $\Xi_0 \overset{t}{\Longrightarrow} \Xi_2$ and $[o] = [o]_{/\Xi_1} = [o]_{/\Xi_2}$. If $\Xi_0 \vdash {}_{[o]}\!\downarrow s = {}_{[o]}\!\downarrow t$, then $\bar{\Xi}_0 \vdash {}_{[o]}\!\downarrow \bar{s} = {}_{[o]}\!\downarrow \bar{t}$ (where in the complementary trace, $[o]$ becomes an environment clique). The property holds dually for environment cliques.*

Proof. Straightforward. Especially, when dualizing a trace r to $\bar{r}$, the future projection from Definition 3.1.3 works on component objects of r the same way as it works on environment objects of $\bar{r}$, and vice versa. □

The next property is mainly a technical lemma needed to show that under certain conditions, swapping preserves the conditions for the judgment $\Xi_0 \vdash t \rhd a : ok$ (cf. Definition A.2.34), which implies that after swapping,

a is still possible. The lemma is used in Lemma A.2.49 afterwards, which covers the case of "input enabledness" in Lemma 3.3.26. Note that the mentioned Lemma A.2.49 deals with preservation of an output action. The proof of Lemma 3.3.26, an important part of completeness, uses the $\preccurlyeq^\bullet$-relation from the perspective of the observer, i.e., $\preccurlyeq^\bullet_\Delta$ instead of $\preccurlyeq^\bullet_\Theta$, and from that perspective, the output label becomes an input label.

Lemma A.2.48. *Assume $\Delta_0 \vdash \odot$ and two legal traces $\Xi_0 \vdash s_1\ s_2 : trace$ and $\Xi_0 \vdash t_1\ t_2 : trace$, where $s_2 = s_2'\ \gamma'?$ and $t_2 = t_2'\ \gamma'?$ for some incoming label $\gamma'?$. Let $[o]$ be a component clique after $s_1\ s_2$ as well as after $t_1\ t_2$. Let furthermore be s_2, resp., t_2 be the complete interaction of $s_1\ s_2$, resp., of $t_1\ t_2$ with the component clique $[o]$, i.e., $s_1\ s_2 \downarrow_{[o]} = s_2$ and $t_1\ t_2 \downarrow_{[o]} = t_2$. Furthermore, assume ${}_{[o]}\!\downarrow s_1\ s_2 = {}_{[o]}\!\downarrow t_1\ t_2$. Then $\Xi_0 \vdash s_1\ s_2 \rhd a : ok$ implies $\Xi_0 \vdash t_1\ t_2 \rhd a : ok$. The lemma holds analogously for $\Theta_0 \vdash \odot$, and furthermore dually for the situation considering an environment clique and $s_1\ s_2$, resp., $t_1\ t_2$ ending in an outgoing communication.*

Proof. First it is easy to see that $\Xi_0 \vdash s_1\ s_2 \rhd a : ok$ implies that a is an outgoing communication, since s_2 ends in an incoming one, i.e., $a = \gamma!$. By the assumption that s_2, resp., t_2 contains the complete interaction with the component clique $[o]$, we have that

$$ {}_{[o]}\!\downarrow s_1\ s_2 = {}_{[o]}\!\downarrow s_2 \quad \text{and} \quad {}_{[o]}\!\downarrow t_1\ t_2 = {}_{[o]}\!\downarrow t_2, \tag{A.21} $$

i.e., the initial sequences s_1, resp., t_2, are not contained in the future projections. By Definition 3.1.7, this implies $\Xi_0 \vdash s_2 \asymp_\Theta t_2$, i.e., s_2 and t_2 are swapping equivalent, using the tree-representation of swapping equality. By Lemma A.2.44,

$$ \Xi_0 \vdash s_2 \mathrel{\dot{\asymp}}_\Theta t_2\ . \tag{A.22} $$

The fact that $\Xi_0 \vdash s_1\ s_2 \rhd a : ok$ and the left-hand equation of (A.21) imply

$$ \Xi_0 \vdash s_2 \rhd a : ok\ . \tag{A.23} $$

Now, s_2 ends with an incoming communication.[6] Hence, by equation (A.22) and Corollary A.2.39, $\Xi_0 \vdash t_2 \rhd a : ok$. With the right-hand equation of (A.21), this yields $\Xi_0 \vdash t_1\ t_2 \rhd a : ok$, as required. □

Lemma A.2.49 ($\preccurlyeq^\bullet_\Theta$ and enabledness)**.** *Assume two weakly balanced traces $\Xi_0 \vdash s : wbalanced$ and $\Xi_0 \vdash t : wbalanced$. If $\Xi_0 \vdash t \rhd o_s \xrightarrow{\gamma!} o_r$ and $\Xi_0 \vdash s \preccurlyeq^\bullet_\Theta t$, then $\Xi_0 \vdash s \rhd o_s \xrightarrow{\gamma!} o_r'$.*

Proof. The relation $\preccurlyeq^\bullet_\Theta$ is given in Definition 3.3.25. We distinguish whether t is empty or not.

If $t = \epsilon$, Definition 3.3.25 requires —only part 2 applies— that $s \preccurlyeq_\Theta t$, which implies $s = \epsilon$, from which the result follows trivially.

If otherwise $t \neq \epsilon$, we argue as follows. It is easy to see that $\Xi_0 \vdash t : wbalanced^+$ and that the *last* label of t is incoming (it is, e.g., a consequence of the combination of Lemma A.2.19 and Lemma A.2.1). So t is of the form $t'\ \gamma'?$ for some label γ'. Furthermore, by Lemma A.2.14, $sender(t'\ \gamma'?\ \gamma!) = receiver(t'\ \gamma'?)$. Let $sender(t'\ \gamma'?\ \gamma!) = o_s$, a component object (potentially $\odot$),

[6] The trace t_2 ends with the same incoming communication, but we don't need the fact to continue the argument.

and let $[o_s]_{/\acute{\Xi}}$, or $[o_s]$ for short, be its component clique, where $\acute{\Xi}$ is the context after $t'\gamma'?\gamma!$, i.e., where $\acute{\Xi}$ is given by $\Xi_0 \overset{t'\gamma'?\gamma!}{\Longrightarrow} \acute{\Xi}$.

Now, by Definition 3.3.25(1), the assumption $\Xi_0 \vdash s \preccurlyeq^{\bullet}_{\Theta} t$ implies

$$ {}_{[o_s]}\!\downarrow s = {}_{[o_s]}\!\downarrow t \,. \tag{A.24}$$

The trace t is of the form $t_1\ t_2$, where t_2 is the end-piece of t projected to the component clique $[o_s]$. Since $t_2 = t_2'\ \gamma'?$ and since the receiver of $\gamma'?$ is o_s, the sequence t_2 is non-empty and ends in an incoming interaction, entering the component clique $[o_s]$.

As s is weakly balanced and thus alternating (Lemma A.2.1), equation (A.24) implies that s, too, is of the form $s = s_1\ s_2$, where s_2 is a non-empty interaction with the clique of o_s, ending with an incoming communication.

For one of two possible situations, assume $\Delta_0 \vdash \odot$, i.e., the thread starts initially in the environment. By Lemma A.2.42, in particular part 2 for traces ending in an incoming communication, we can re-order the two traces via swapping into

$$\Xi_0 \vdash t_1\ t_2 \doteq_{\Theta} t_1'\ t_2' \quad \text{and} \quad \Xi_0 \vdash s_1\ s_2 \doteq_{\Theta} s_1'\ s_2' \,, \tag{A.25}$$

where t_2' contains the *complete* interaction of $t_1\ t_2$ with the clique of o_s, and analogously for s_2'.

By the preservation Lemma A.2.38, the assumption $\Xi_0 \vdash t_1\ t_2 \rhd \gamma! : ok$ and the swapping on the left-hand of (A.25) imply

$$\Xi_0 \vdash t_1'\ t_2' \rhd \gamma! : ok \tag{A.26}$$

Note that the receiver of $\gamma!$ is *not* preserved by the re-arrangement, but the sender, a component object, is (see in particular part 1 of Lemma A.2.38). Remains to be argued that $\gamma!$ is possible after $s_1'\ s_2$, as well, which in particular means that the sender remains unchanged.

The two end-traces t_2' and s_2' are given by $t_1'\ t_2'\ \downarrow_{[o_s]}$ and $s_1'\ s_2'\ \downarrow_{[o_s]}$. By Lemma A.2.48, equation (A.26) from above implies that

$$\Xi_0 \vdash s_1'\ s_2' \rhd \gamma! : ok \ . \tag{A.27}$$

Now once again by the preservation Lemma A.2.38 and using the right-hand judgment of equation (A.25),

$$\Xi_0 \vdash s_1\ s_2 \rhd \gamma! : \ o_s \xrightarrow{\gamma!} o_r^2 \,, \tag{A.28}$$

which finishes the case. □

A.2.3 Replay

Next we present an equational characterization of replay, i.e., of the relation $\preccurlyeq_{\Theta}$ combining swapping and replay. As given in Definition 3.1.8, the relation $\preccurlyeq_{\Theta}$ is given based on the future projection of a trace to objects or rather cliques occurring in the trace (cf. Definition 3.1.3 for ${}_{o}\!\downarrow t$). The phenomenon of replay ("what can be done once, can be done twice") was covered in the definition of $\Xi_0 \vdash s \preccurlyeq_{\Theta} t$ in that for each component clique of s (i.e., after s), there must be a corresponding one after t (after potentially renaming t) such that the behavior of the clique of s is covered by the behavior of the clique of t, and vice

versa. Definition 3.1.8 included "swapping" in that it was based, via the future projection, on the tree representation of the merging clique structure.

Now that we have represented $\asymp_\Theta$ equivalently by an equational characterization $\dot{\asymp}_\Theta$ and since we intend to do the same for $\preccurlyeq_\Theta$ combining swapping and replay, we cannot (or rather should not) base the characterization of replay on the future projection. Instead, we use the *past projection* (see Definition A.3.1), which does not represent the swapping or the tree-like structure of the semantics, but simply contains the part of the global, linear trace relevant for the clique projected onto.

Definition A.2.50 (Swapping and replay)**.** *The relation $\dot{\preccurlyeq}_\Theta$ on traces is given by the reflexive, transitive, and symmetric closure of the rules of Table A.3. The relation $\dot{\preccurlyeq}_\Delta$ is defined dually.*

$$\frac{\Xi_0 \vdash s \mathrel{\dot{\asymp}_\Theta} t}{\Xi_0 \vdash s \mathrel{\dot{\preccurlyeq}_\Theta} t}\ \textsc{Swap}_\Theta$$

$$\frac{\Xi_0 \vdash s \rhd \gamma! \quad \mathit{sender}(s\,\gamma!) = o \qquad s\,\gamma! \downarrow_{[o]} \preccurlyeq s' \downarrow_{[o]} \quad s =_\alpha s'}{\Xi_0 \vdash s\,\gamma! \mathrel{\dot{\preccurlyeq}_\Theta} s}\ \textsc{ReO}_\Theta \qquad \frac{\Xi_0 \vdash s \rhd \gamma! \quad \mathit{receiver}(s\,\gamma?) = o \qquad s\,\gamma? \downarrow_{[o]} \preccurlyeq s' \downarrow_{[o]} \quad s =_\alpha s'}{\Xi_0 \vdash s\,\gamma? \mathrel{\dot{\preccurlyeq}_\Theta} s}\ \textsc{ReI}_\Theta$$

Table A.3: Swapping and replay

Question A.2.51 (Replay and alternation)**.** *In the current definition of replay from Definition A.2.50, Table A.3, alternation is not covered. Is it a problem, what is the problem if there is one, and how can we solve it, and is alternation the only problem?*

Answer: First of all, the definition is for the sequential, single-threaded case. Similar problems occur also in the multi-threaded case, of course.

Is it a problem, first of all? I think, yes. The premise currently requires, that one can simply add $\gamma!$, if $s\,\gamma!$, projected to the sender clique of o is a prefix of a renaming of s, projected to the same clique. If now s itself ends with an outgoing call, we clearly cannot extend it by another $\gamma!$ as this would not be alternating.

One way to remedy it is to require

$$\Xi_0 \vdash r \rhd \gamma!$$

That's basically the same as alternation. This is what we need at least. That's already required such that the result of sender and $\mathit{receiver}$ is defined.

Other potential problems are

1. typing
2. connectivity
3. determinism

But they seem no problems. □

Rule SWAP$_\Theta$ incorporates swapping as subset of $\dot{\asymp}_\Theta$. As for the replay, i.e., repetition of an action already witnessed in the past, we distinguish between incoming and outgoing labels. For an outgoing label $\gamma!$ (cf. rule REO$_\Theta$), the *sender* clique of the communication (a component clique), is relevant. The projection $s\ \gamma!\downarrow_{[o]}$ in the premise of the rule in particular contains the (projection of the) label $\gamma!$. If that projection $s\ \gamma!\downarrow_{[o]}$ is already contained in the shorter trace s, then the new $\gamma!$ constitutes no new behavior. Of course, we cannot expect the $s\gamma!\downarrow_{[o]}$ to occur *literally* in s. Hence, we rename s to an appropriate α-variant s' in the premise of the rule.

For incoming labels, rule REI$_\Theta$ works analogously, considering the clique of the receiver of the action, instead of the sender.

Next we have to show that the relation $\asymp_\Theta$ from Definition 3.1.8 and the equational definition $\dot{\asymp}_\Theta$ coincide.

Lemma A.2.52 ($\dot{\asymp}_\Theta$ implies $\asymp_\Theta$)**.** *Assume s and t to be weakly balanced. Then $\Xi_0 \vdash s \dot{\asymp}_\Theta t$ implies $\Xi_0 \vdash s \asymp_\Theta t$. The lemma holds analogously for $\dot{\asymp}_\Delta$ and $\asymp_\Delta$.*

Proof. See Table A.3 for the definition of $\dot{\asymp}_\Theta$. We show the implication for one instance of a rule for $\dot{\asymp}_\Theta$. The result then follows by induction/transitivity. For swapping with rule SWAP$_\Theta$, the result follows by the corresponding Lemma A.2.36 for $\dot{\asymp}_\Theta$.

Case: REO$_\Theta$: $\Xi_0 \vdash s\ \gamma!\ \dot{\asymp}_\Theta\ s$
$\Xi_0 \vdash s\ \gamma!\ \not\succcurlyeq_\Theta\ s$ as one half of the claim is immediate by definition. For the reverse direction we argue as follows. By the premise of the rule, $s\ \gamma!\downarrow_{[o]} \preccurlyeq s'\downarrow_{[o]}$, where o is the sender of $\gamma!$ and s' an appropriate α-renaming of s. For component cliques $[o']$ other than the sender clique $[o]$, we have $s\ \gamma!\downarrow_{[o']} = s\downarrow_{[o']}$, i.e., in particular $s\ \gamma!\downarrow_{[o']} \preccurlyeq s\downarrow_{[o]}$. By Lemma A.3.12, connecting forward and backward projection, we therefore have for all component cliques $[o'']$ that also for the forward projection ${}_{[o'']}\!\downarrow s\ \gamma! \preccurlyeq {}_{[o'']}\!\downarrow s$. Hence, $\Xi_0 \vdash s\ \gamma!\ \not\preccurlyeq_\Theta\ s$, finishing the case.

Case: REI$_\Theta$: $\Xi_0 \vdash s\ \gamma?\ \dot{\asymp}_\Theta\ s$
Analogously. □

Lemma A.2.53 ($\asymp_\Theta$ implies $\dot{\asymp}_\Theta$)**.** *Assume s and t to be weakly balanced. Then $\Xi_0 \vdash s \asymp_\Theta t$ implies $\Xi_0 \vdash s \dot{\asymp}_\Theta t$. The property holds analogously for $\asymp_\Delta$ and $\dot{\asymp}_\Delta$.*

Proof. For the definition of $\asymp_\Theta$, see Definition 3.1.8. Proceed by induction on the length of s. In the base case, where s is empty, $\epsilon \asymp_\Theta t$ implies by definition that $t = \epsilon$, and the case follows by reflexivity of $\dot{\asymp}_\Theta$. For the induction step, we are given $s = r\ a$ and distinguish according to the nature of a.

Case: $s = r\ \gamma_c?$ (incoming call)
We distinguish further, whether the additional label is already a replay wrt. the shorter r or not.

Subcase: $\Xi_0 \vdash r\ \gamma_c?\ \not\preccurlyeq_\Theta\ r$
Since clearly $\Xi_0 \vdash r\ \gamma_c?\ \not\succcurlyeq_\Theta\ r$, we immediately get $\Xi_0 \vdash r\ \gamma_c?\ \asymp_\Theta\ r$. By induction we get $\Xi_0 \vdash r\ \dot{\asymp}_\Theta\ r\ \gamma_c?$. Furthermore, by transitivity of $\asymp_\Theta$, $\Xi_0 \vdash r \asymp_\Theta t$. Hence, again by induction, $\Xi_0 \vdash r\ \dot{\asymp}_\Theta\ t$, which implies the required $\Xi_0 \vdash r\ \gamma_c?\ \dot{\asymp}_\Theta\ t$ by transitivity of $\dot{\asymp}_\Theta$.

Subcase: $\Xi_0 \not\vdash r\ \gamma_c? \preccurlyeq_\Theta r$
This means the extension of the component clique by the additional $\gamma_c?$ is *not* already covered by the behavior of any other component clique. The assertion $\Xi_0 \vdash r\ \gamma_c? \asymp_\Theta t$ implies that t is of the form $t =_\alpha t'$ s.t. $t' = t'_1\ \gamma_c?\ t'_2$.

We first argue that the subsequence t'_2 at the end is empty. If not empty, the trailing t'_2 must be of the form $\gamma!\ t''_2$ for some *outgoing* label $\gamma!$, sent by the component clique of $\gamma_c?$ since all weakly balanced traces are alternating (Lemma A.2.1). Now, the sender of $\gamma!$ corresponds to the receiver of $\gamma_c?$, in particular, $\gamma!$ belongs to the same clique as the receiver of $\gamma_c?$. This contradicts the assumption that $\Xi_0 \not\vdash r\ \gamma_c? \preccurlyeq_\Theta r$ and the assumption $\Xi_0 \vdash r\ \gamma_c? \succcurlyeq_\Theta t$ (as one direction of $\Xi_0 \vdash r\ \gamma_c? \asymp_\Theta t$). Therefore we are given $\Xi_0 \vdash r\ \gamma_c? \asymp_\Theta t'_1\ \gamma_c?$.[7]

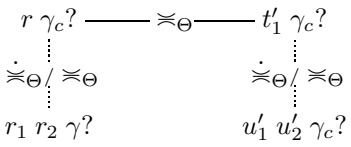
Assume that the thread starts in the environment, i.e., $\Delta_0 \vdash \odot$. As the trace ends with an input we have $\vdash r\ \gamma? : \mathit{wbalanced}^+$, hence both

$$\Xi_0 \vdash r\ \gamma? \mathrel{\dot\asymp_\Theta} r_1\ r_2\ \gamma? \quad \text{and} \quad \Xi_0 \vdash r\ \gamma? \asymp_\Theta r_1\ r_2\ \gamma?\ ,$$

(reorganizing the thread with Lemma A.2.42 for the left-hand judgment and additionally with the help of Lemma A.2.52 for the judgment on the right), where $r_2\ \gamma?$ contains the complete interaction with the receiver clique of $\gamma?$. We can apply the same lemmas to $\Xi_0 \vdash t'_1\ \gamma_c?$, yielding $\Xi_0 \vdash t'_1\ \gamma_c? \mathrel{\dot\asymp_\Theta} u'_1\ u'_2\ \gamma_c?$ and analogously for $\asymp_\Theta$. Since $\asymp_\Theta$ implies by definition $\preccurlyeq_\Theta$ and analogously $\mathrel{\dot\asymp_\Theta}$ implies $\preccurlyeq_\Theta$ (rule $\textsc{Swap}_\Theta$), we have by transitivity of $\preccurlyeq_\Theta$ that

$$\Xi_0 \vdash r_1\ r_2\ \gamma_c? \preccurlyeq_\Theta u'_1\ u'_2\ \gamma_c?\ , \tag{A.29}$$

where neither r_1 nor u'_1 contains labels belonging to $[o]$. The situation is summarized in the following diagram:



Equation (A.29) and the fact that $r_2\ \gamma_c?$, resp., $u'_2\ \gamma_c?$ contains all the behavior projected to $[o]$ clearly imply $\Xi_0 \vdash r_1 \preccurlyeq_\Theta u'_1$, whence by induction $\Xi_0 \vdash r_1 \mathrel{\dot\preccurlyeq_\Theta} u'_1$. Furthermore we know by preservation of weak balance by past projection (Lemma A.3.3) that $\vdash r_2\ \gamma_c? : \mathit{wbalanced}^+$ and $\vdash u'_2\ \gamma_c? : \mathit{wbalanced}^+$. Thus, $\Xi_0 \vdash r_1 r_2 \gamma? \mathrel{\dot\preccurlyeq_\Theta} u'_1 u'_2 \gamma_c?$ by the extension Lemma A.2.41.

Case: $s = r\ \gamma_c!$
Analogous to the case for input. □

Remark A.2.54. *Concerning the proof of Lemma A.2.53, note that the used "extension" Lemma A.2.41 is formulated for the swapping relation* $\mathrel{\dot\asymp_\Theta}$*, and not for* $\mathrel{\dot\preccurlyeq_\Theta}$*. Indeed, we do* not *have an analog of Lemma A.2.41 for* $\mathrel{\dot\preccurlyeq_\Theta}$*. In general, it is difficult to formulate a corresponding property for* $\mathrel{\dot\preccurlyeq_\Theta}$*. The complication comes from the fact that* $\mathrel{\dot\preccurlyeq_\Theta}$ *has renaming built-in (to formulate replay), whereas* $\mathrel{\dot\asymp_\Theta}$ *just reorders the labels without renaming. Even in the simplest case of just renaming, an "extension property" like "if* $s =_\alpha t$*, then* $s\ a \mathrel{\dot\preccurlyeq_\Theta} t\ a$*" makes no sense.* □

[7] Note that additionally we know $\Xi_0 \vdash r \preccurlyeq_\Theta t'_1$, and thus by induction, $\Xi_0 \vdash r \mathrel{\dot\preccurlyeq_\Theta} t'_1$. We do not use this fact. Due to renaming, we cannot generally conclude that extending $r \mathrel{\dot\preccurlyeq_\Theta} t'_1$ by an additional $\gamma_c?$ preserves $\mathrel{\dot\preccurlyeq_\Theta}$.

Corollary A.2.55. *Assume s and t to be weakly balanced. Then $\Xi_0 \vdash s \asymp_\Theta t$ iff $\Xi_0 \vdash s \dot{\asymp}_\Theta t$. The property holds analogously for $\asymp_\Delta$ and $\dot{\asymp}_\Delta$.*

Proof. The conjunction of Lemma A.2.52 and A.2.53. □

Lemma A.2.56 (Dualizing)**.** *If $\Xi_0 \vdash s \preccurlyeq_\Theta t$, then $\Xi_0 \vdash \bar{s} \preccurlyeq_\Delta \bar{t}$. The same holds for the relationship of $\dot{\preccurlyeq}_\Theta$ and $\dot{\preccurlyeq}_\Delta$, and also for the symmetric variants of the replay relation, i.e., $\asymp_\Theta$ and $\asymp_\Delta$, resp., $\dot{\asymp}_\Theta$ and $\dot{\asymp}_\Delta$.*

Furthermore, $\vdash t : det_\Delta$, then $\vdash \bar{t} : det_\Theta$; the analogous property holds for $\vdash t : det_\Theta$.

Proof. Obvious. All definitions leading to $\dot{\preccurlyeq}_\Theta$ are dually defined for $\dot{\preccurlyeq}_\Delta$ (see also Lemma A.2.46 for the corresponding property for swapping). Analogously for the other mentioned relations. □

A.3 Traces, cliques, and projections

In the following we prove a number facts about traces, the tree structure of merging cliques, and projections of the global trace onto a local clique. In particular, we define a *past* projection of a trace onto a clique.

A.3.1 Past projection

We define the *projection* of a trace onto a clique as the part of the sequence interacting with that clique. Remember that the clique of a component object $\Theta \vdash o$ (where o can also represent $\odot$ or —in the multithreaded setting— $\odot_n$) consists of all objects from Θ acquainted with o (dual definitions apply to environment objects). Thus the equivalence $\leftrightharpoons$ partitions Θ into equivalence classes, and we write $[o]_{/\Xi}$ for that equivalence class. For simplicity, we often just write $[o]$, when Ξ is clear from the context.

Earlier, we provided a *different* notion of projection of a global trace onto objects and cliques. Both definitions are similar insofar that both take as input a global, linear trace, keep all labels interacting with the clique being projected on, and jettison all interaction not interacting with the clique. In short, both provide a clique-local view of a given, global trace.

There is, however, a crucial conceptual difference between both notions of projection. In Definition 3.1.3, ${}_{[o]}\!\downarrow t$ is given by recording the interaction of each single object $o' \in [o]$, taking the tree-like, *evolving* clique structure into account. Indeed, the projection was used to formalize the swapping relation $\asymp_\Theta$ (cf. Definition 3.1.7). In contrast, the projection $t \downarrow_{[o]}$ (see below) does *not* take into account the *evolving* clique structure, but just takes the clique $[o]$ after t as *fixed.* This latter implies, that the projection $t \downarrow_{[o]}$ is *linear* in nature, it is just the portion of the linear, global t, that interacts with one chosen, fixed clique. For distinction, we call ${}_{[o]}\!\downarrow t$ the *future* or forward projection, $t \downarrow_{[o]}$, defined below, the *past* or backward projection.

The definition of projection of an trace onto a clique of environment objects is straightforward (and a bit simpler that the future projection from Definition 3.1.3): One simply jettisons all actions not belonging to that clique. One only has to be careful dealing with exchange of bound names. The trace to start from is global and thus scope extrusion of fresh names to the environment is

accounted for on a global level, namely whether the outside in its entirety has been told the name or not. From the *local* perspective of the environment clique we project onto, a name which has been given before to another clique nevertheless is *locally new,* since the clique has no way of *comparing* the name with the identity previously sent to other cliques.

Given a global trace, its projection onto one particular clique of objects as given at the end of the trace is defined straightforwardly by induction on the length of the trace, appending actions at the end (for the definition of sender and receiver of a label after a history t, see Definition 3.3.4. Remember also the conventions from Notation 2.6.5):

Definition A.3.1 (Past projection)**.** *Assume as trace $\Xi \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ and let $\acute{\Delta} \vdash o$ for some object reference o. Then the projection of t onto the clique $[o]$ according to $\acute{\Delta}; \acute{E}_\Delta$, written $t \downarrow_{[o]}$, is defined as follows: $t' = t \downarrow_{[o]}$, if there exists a derivation according to Table A.4 starting with $\Phi_0 \vdash \epsilon \rhd_{[o]} t$ and with $\Phi \vdash t' \rhd_{[o]} \epsilon$ as axiom. The projection onto a component clique is defined dually. We use the same definition*

$$\frac{}{\Phi \vdash r \rhd_{[o]} \epsilon}\ \text{P-EMPTY}$$

$$\frac{\mathit{receiver}(t\,\gamma!) \notin [o] \quad \Phi \vdash r \rhd_{[o]} s}{\Phi \vdash r \rhd_{[o]} \gamma!\, s}\ \text{P-OUT}_1$$

$$\frac{\mathit{receiver}(t\,\gamma!) \in [o] \quad \Phi_2' = \mathit{fn}(\nu(\Phi_1').\gamma) \setminus \Phi \quad \Phi, \Phi_1', \Phi_2' \vdash r\ \nu(\Phi_1', \Phi_2').\gamma! \rhd_{[o]} s}{\Phi \vdash r \rhd_{[o]} \nu(\Phi_1').\gamma!\, s}\ \text{P-OUT}_2$$

$$\frac{\mathit{sender}(t\,\gamma?) \notin [o] \quad \Phi \vdash r \rhd_{[o]} s}{\Phi \vdash r \rhd_{[o]} \gamma?\, s}\ \text{P-IN}_1 \qquad \frac{\mathit{sender}(t\,\gamma?) \in [o] \quad \Phi, \Phi' \vdash r\ \nu(\Phi').\gamma? \rhd_{[o]} s}{\Phi \vdash r \rhd_{[o]} \nu(\Phi').\gamma?\, s}\ \text{P-IN}_2$$

Table A.4: Past projection onto an environment clique

analogously for legal traces.

The projection of the empty trace remains empty (rule P-EMPTY). For output actions in P-OUT$_1$ and P-OUT$_2$ we distinguish according to the receiver. If the receiver is not involved in the communication, the label is "projected out"; dually for incoming communication. More interesting is P-OUT$_2$: Fresh names are not only the globally fresh ones Φ_1', but also the locally fresh ones Φ_2'. The situation for incoming new names is *not symmetric!* It is simpler as we need not distinguish between locally and globally new names: Everything that the clique has created in isolation is globally new as well as locally new.

Remark A.3.2 (Projection and new identities)**.** *The projection uses the sender, resp., the receiver, of a label. Furthermore, the definition of projection keeps track of* locally *new names; in particular in the rule* P-OUT$_2$ *dealing with names received freshly by an environment clique. Note that the cliques in the definitions are those at*

the end of the given trace. As a consequence, merging *two environment cliques is* not *reflected in bound exchange of environment names, even if the names of environment objects of the cliques being merged are guaranteed to be mutually unknown. A different way of phrasing it is that the locally new names* Φ_2' *mentioned in* P-OUT$_2$ *concern* only *component objects, which means that projection does not "introduce" new occurrences of lazily instantiated objects.* □

As mentioned, we also define the *future* projection of a trace onto a clique, in contrast to Definition A.3.1, which defines the *backward* projection of a trace, i.e., the past interaction of a trace with a clique. Technically, the definition of the forward projection is slightly more complex as one has to take the changing clique structure into account, while on the past interaction, one simply collects all interactions with the current clique, irrespective of the past evolution of the clique structure. Thus the rules of Table 3.2 uses the full contexts including connectivity, whereas the rules of Table A.4 can be defined using only the name contexts. Otherwise, the treatment of new names is analogous as in the backward projection, i.e., a name which is globally known in the past trace, but not *locally* known to the clique onto which it is projected, is counted as new in the projection.

The following lemma shows that the past projection to a clique is weakly balanced. Note that the *future* projection ${}_o\!\downarrow t$ of a trace to an object clearly need *not* be weakly balanced. What breaks weak balance is the situation, when in the course of t, the clique of o is *merged* by a return whose matching call originated not from the clique of o but from the partner clique being merged.

Lemma A.3.3 (Weak balance and projection)**.** *Let t be a weakly balanced trace and $[o]$ be a component clique after t. Then $\vdash t \downarrow_{[o]}$: $wbalanced$. The lemma holds dually for environment cliques.*

Proof. Proceed by induction on the rules of Table A.4, showing that for all judgments $\Phi \vdash r \rhd_{[o]} s$ in the derivation, the r is weakly balanced, using $\Phi_0 \vdash \epsilon \rhd_{[o]} s$ as base case. So for $r = \epsilon$, the claim holds trivially. For the induction step, the only two interesting cases are P-OUT$_2$ and P-IN$_2$ (the induction step for P-OUT$_1$ and P-IN$_1$ is trivial).

In case of P-OUT$_2$ it is easy to check that the condition $sender(t\ \gamma!) \in [o]$ of the rule implies $\Xi_0 \vdash r \rhd \gamma!$. Analogously for P-INT$_2$. So the result follows by Lemma A.2.19. □

A.3.2 Tree structure

The evolving connectivity gives rise to a tree structure (a forest), concerning the cliques as equivalence classes of objects. The tree structure lies at the core of the semantics and is important in the completeness construction: The tree data structure needs to be represented in the code of the observer. This section proves a few properties concerning the tree-structured cliques.

In particular, it shows invariants of the names and their relationship occurring in a legal trace t, transforming the initial context Ξ_0 into a post-context Ξ, written $\Xi_0 \overset{t}{\Longrightarrow} \Xi$ (see Definition A.5.3). Since all traces of a component are legal (Lemma A.5.9), and since furthermore $\Xi_0 \vdash C_0 \overset{t}{\Longrightarrow} \Xi \vdash C$ implies $\Xi_0 \overset{t}{\Longrightarrow} \Xi$,

the corresponding lemmas hold analogously also for traces from reductions $\Xi_0 \vdash C_0 \stackrel{t}{\Longrightarrow} \Xi \vdash C$.

The following lemma expresses that during evolution, a given clique only grows larger, resp., that the object names of separate cliques are disjoint.

Lemma A.3.4 (Tree structure of cliques)**.** *Assume $\Xi_0 \stackrel{r}{\Longrightarrow} \Xi_1$ and $\Xi_0 \stackrel{r\ u}{\Longrightarrow} \Xi_2$. Let $[o_1]_{/\Xi_1}$ (or $[o_1]$ for short) be a clique of component objects according to Ξ_1, i.e., after r, analogously for $[o_2]_{/\Xi_2}$ (resp. $[o_2]$ for short) after $r\ u$. Then one of the following three disjoint cases applies:*

1. *$[o_1] \subset [o_2]$, or*
2. *$[o_1] = [o_2]$, or*
3. *$[o_1] \cap [o_2] = \emptyset$.*

For environment cliques, the lemma holds dually.

Proof. Note that the equivalence classes contain always a *non-empty* set of object names; hence the three cases are mutually exclusive. Proceed by induction on the length of u. If $u = \epsilon$, immediately case (2) or (3) applies, since E_Θ implies a partitioning of the Θ-objects. In the induction case, $u = u'\ a$ for some label a. An outgoing label only enlarges *one* existing component clique —the one of the sender— by scope extrusion, i.e., by adding *new* object references, and thus preserves the invariant. An incoming label may *merge* existing cliques and/or add new references by lazy instantiation. It suffices to consider the binary merge, the n-ary merge then follows easily (since the operation of adding sets of fresh names is associative and commutative). So consider two component cliques $[o_2^1]$ and $[o_2^2]$ being merged by a. If both are disjoint with $[o_1]$ according to case (3), then the combined clique is disjoint. If wlog. $[o_2^1] \cap [o_1] = \emptyset$ and $[o_2^2] \supset [o_1]$, then after the merge, $[o_2^1, o_2^2] \supset [o_1]$, i.e., case (1) applies. If both $[o_2^1] \supset [o_1]$ and $[o_2^2] \supset [o_1]$, then clearly $[o_2^1, o_2^2] \supset [o_1]$. If wlog. $[o_2^1] = [o_1]$ (and consequently $[o_2^2] \cap [o_1] = \emptyset$), then after the merge, case (1) applies. □

Corollary A.3.5 (Tree structure of cliques)**.** *Assume $\Xi_0 \stackrel{r_1}{\Longrightarrow} \Xi_1$ and $\Xi_0 \stackrel{r_2}{\Longrightarrow} \Xi_2$ with $r_1 \preccurlyeq t$ and $r_2 \preccurlyeq t$ for some legal trace t ($\Xi_0 \vdash t : trace$). Let $[o_1]_{/\Xi_1}$ (or $[o_1]$ for short) be the clique of component objects according to Ξ_1 i.e., after r_1, analogously for $[o_2]_{/\Xi_2}$ (resp. $[o_2]$ for short) after r_2. Then one of the following 4 disjoint cases applies: $[o_1] \subset [o_2]$, $[o_1] \supset [o_2]$, $[o_1] = [o_2]$, or else $[o_1] \cap [o_2] = \emptyset$. For environment cliques, the lemma holds dually.*

Proof. A direct consequence of Lemma A.3.4, since $r_1 \preccurlyeq r_2$ or $r_2 \preccurlyeq r_1$. □

Next a few straightforward properties of the (forward and backward) projections of a trace, relating them with the evolving clique structure. The lemmas are formulated wrt. component cliques, but the properties apply in dual form to environment cliques, as well. The next lemma simply states that a clique consists of all the names of component objects encountered in the past, as projected to that clique.

Lemma A.3.6 (Projection and names)**.** *Assume $\Xi_0 \stackrel{r}{\Longrightarrow} \Xi$, and let o be a component object after r, i.e., $\Theta \vdash o$, where $[o]_{/\Xi}$ (or $[o]$ for short) denotes o's clique according to Ξ. Then*

$$[o] = names_\Theta(r \downarrow_{[o]}) = bn_\Theta(r \downarrow_{[o]}) .$$

Proof. By induction on the length of r, using Definition A.3.1 and Table A.4 (resp., the dual variant for component cliques). For $r = \epsilon$, i.e., in case of P-EMPTY, the property is trivially satisfied: with Ξ_0 containing only classes, there is no object reference o with $\Theta_0 \vdash o$. For an incoming communication, which does not affect the clique under consideration (cf. rule P-OUT$_1$)[8] the set of known object names is not changed and neither is the projection of the trace prolonged; hence the case follows by induction. In case of (the dual of) rule P-OUT$_2$ with the trace of the form $t\ \nu(\Phi_1').\gamma? = t\ a$, we are facing the following situation: $\Xi_0 \stackrel{t}{\Longrightarrow} \Xi \stackrel{a}{\Longrightarrow} \acute{\Xi}$. By induction, the projection t', as given by $\Phi' \vdash t' \rhd_{[o]} a$, contains the names of o's clique according to the assertion Ξ before the step, i.e.,

$$[o]_{/\Xi} = names_\Theta(t') = bn_\Theta(t') \ .$$

Now the derivation rule dual to P-OUT$_2$ extends Φ' by all (free and bound) names of a not yet contained in Φ'. This corresponds to the extension of Ξ to $\acute{\Xi}$ via $\acute{\Xi} = \Xi + o_r \stackrel{a}{\leftarrow} o_s$ wrt. component objects, i.e., wrt. the extension of the receiver clique (see in particular Definition 2.6.8 for the update of the name contexts). The cases for outgoing communication, corresponding to (the duals of) P-IN$_1$ and P-IN$_2$ and dealing with lazy instantiation, are simpler. □

The next lemma is a simple consequence. We denote by $\preccurlyeq^t$ the "prefix" relation on trees, i.e., $t_1 \preccurlyeq^t t_2$ means, t_1 is a sub-tree of t_2. Furthermore, we need the suffix relation on (linear) sequences which we denote by $\succcurlyeq^s$. We use $\prec^t$ and $\succ^s$ for the respective "strict" variants of the order relations.

Lemma A.3.7. *Assume $\Xi_0 \stackrel{r_1}{\Longrightarrow} \Xi_1$ and $\Xi_0 \stackrel{r_2}{\Longrightarrow} \Xi_2$ with $r_1 \preccurlyeq t$ and $r_2 \preccurlyeq t$, i.e., $t = r_1\ s_1 = r_2\ s_2$ for some s_1, resp., s_2. Let $[o_1]_{/\Xi_1}$ (or $[o_1]$ for short) be the clique of component objects according to Ξ_1 i.e., after r_1, analogously for $[o_2]_{/\Xi_2}$ (or $[o_2]$ for short).*

1. *If $names_\Theta(r_1 \downarrow_{[o_1]}) = names_\Theta(r_2 \downarrow_{[o_2]})$ then $r_1 \downarrow_{[o_1]} \preccurlyeq^t r_2 \downarrow_{[o_2]}$ or $r_1 \downarrow_{[o_1]} \succcurlyeq^t r_2 \downarrow_{[o_2]}$.*
2. *If $names_\Theta(r_1 \downarrow_{[o_1]}) \subset names_\Theta(r_1 \downarrow_{[o_1]})$ then $r_1 \downarrow_{[o_1]} \prec^t r_2 \downarrow_{[o_2]}$.*

Proof. Straightforward. □

The following lemma relates the forward and the backward projection.

Lemma A.3.8 (Forward and backward projection)**.** *Let t be a legal trace, and $r\ s = t$ for some r and s. Furthermore, let $[o]$ be a component clique after r. Assume further $s \downarrow_{[o]} \neq \epsilon$. Then*

$$r \downarrow_{[o]} = t - {}_{[o]}\!\downarrow s \ . \tag{A.30}$$

Proof. See Definition 3.1.5, equation (3.7), for the definition of $t - {}_{[o]}\!\downarrow s$. Straightforward by induction on the length of r. □

The next lemma is a "dynamic" variant of Lemma B.4.10 later. It states the not too surprising fact that each clique of component objects consists exactly of the references mentioned in the corresponding subtree, where the subtree is given by the interaction path to one of the roots which represents the future behavior of that clique (cf. Definition 3.1.5). The connection between the cliques

[8] Remember that the rules of Table A.4 are formulated for the dual case of environment cliques.

of objects and the future interaction is important for the implementation later: When realizing a trace t, each object of the implementation will have a representation of the futures $t - s_o$ together with the matching current clique $[o]$, which identifies a possible current state in the execution of t.

Lemma A.3.9. *Let t be a legal trace, and $rs = t$ for some r and s. Furthermore, let $[o]$ be some component clique $[o]$ after r. Assume further that ${}_{[o]}\!\downarrow s \neq \epsilon$. Then*

$$[o]_{/\Xi} = names_\Theta(t - {}_{[o]}\!\downarrow s) \; . \tag{A.31}$$

Proof. By Lemma A.3.8 and Lemma A.3.6. □

The next two lemmas connect the past projection, resp. the set of objects of a clique, with the future projection.

Lemma A.3.10 (Forward and backward)**.** *Assume $\Xi_0 \vdash C_0 \stackrel{r_1}{\Longrightarrow} \Xi_1 \vdash C_1$ and $\Xi_0 \vdash C_0 \stackrel{r_2}{\Longrightarrow} \Xi_2 \vdash C_2$ with $r_1 \preccurlyeq t$ and $r_2 \preccurlyeq t$, i.e., $t = r_1\ s_1 = r_2\ s_2$ for some s_1 resp. s_2. Let $[o_1]_{/\Xi_1}$ (or $[o_1]$ for short) be the clique of component objects according to Ξ_1 i.e., after r_1, analogously for $[o_2]_{/\Xi_2}$ (resp. $[o_2]$ for short).*

1. $r_1 \downarrow_{[o_1]} = r_2 \downarrow_{[o_2]}$ *iff.* ${}_{[o_1]}\!\downarrow s_1 = {}_{[o_2]}\!\downarrow s_2$.
2. $r_1 \downarrow_{[o_1]} \prec^t r_2 \downarrow_{[o_2]}$ *iff.* ${}_{[o_1]}\!\downarrow s_1 \succ^s {}_{[o_2]}\!\downarrow s_2$.

Proof. Straightforward. □

Lemma A.3.11 (Projection and cliques)**.** *Assume $\Xi_0 \stackrel{r_1}{\Longrightarrow} \Xi_1$ and $\Xi_0 \stackrel{r_2}{\Longrightarrow} \Xi_2$ with $r_1 \preccurlyeq t$ and $r_2 \preccurlyeq t$, i.e., $t = r_1\ s_1 = r_2\ s_2$ for some s_1, resp., s_2. Let $[o_1]_{/\Xi_1}$ (or $[o_1]$ for short) be the clique of component objects according to Ξ_1 i.e., after r_1, analogously for $[o_2]_{/\Xi_2}$ (or $[o_2]$ for short).*

1. (a) *If* ${}_{[o_1]}\!\downarrow s_1 = {}_{[o_2]}\!\downarrow s_2 \neq \epsilon$, *then* $[o_1] = [o_2]$.
   (b) *If* $\epsilon \neq {}_{[o_1]}\!\downarrow s_1 \prec^s {}_{[o_2]}\!\downarrow s_2$, *then* $[o_1] \supseteq [o_2]$.
   (c) *If not* ${}_{[o_1]}\!\downarrow s_1 \preccurlyeq^s {}_{[o_2]}\!\downarrow s_2$ *nor* ${}_{[o_2]}\!\downarrow s_2 \preccurlyeq^s {}_{[o_1]}\!\downarrow s_1$, *then* $[o_1] \cap [o_2] = \emptyset$.
2. (a) *If* $[o_1] = [o_2]$, *then* ${}_{[o_1]}\!\downarrow s_1 \preccurlyeq^s {}_{[o_2]}\!\downarrow s_2$ *or* ${}_{[o_2]}\!\downarrow s_2 \preccurlyeq^s {}_{[o_1]}\!\downarrow s_1$.
   (b) *If* $[o_1] \supset [o_2]$, *then* ${}_{[o_1]}\!\downarrow s_1 \prec^s {}_{[o_2]}\!\downarrow s_2$.

Proof. Straightforward, by the uniqueness of names, and the fact that communication only *adds* information to Θ and E_Θ.

Part 1a follows by Lemma A.3.10(1) and Lemma A.3.6; part 1b analogously by Lemma A.3.10(2) and again Lemma A.3.6. Part 2 for the reverse direction follows by Lemma A.3.7 and Lemma A.3.10. For part 1c, we know (using part 2) that neither $[o_1] \subseteq [o_2]$ nor $[o_2] \subseteq [o_1]$ can hold. This leaves $[o_1] \cap [o_2] = \emptyset$ as only alternative (cf. Corollary A.3.5). □

Part (1) and part (2) of Lemma A.3.11 can be seen as inverse aspects of the connection between component cliques and the future projection. Note, however, that neither the implication of part (1) nor of part (2) holds in inverse direction. In particular, the equality $[o_1] = [o_2]$ does not imply that the corresponding projections ${}_{[o_1]}\!\downarrow s_1$ and ${}_{[o_2]}\!\downarrow s_2$ are equal.

We introduced two kinds of projections on a linear trace, the future projection from Definition 3.1.3 and the projection on the past from Definition A.3.1.

As mentioned, given a (e.g., component) clique $[o]$ after a trace t, a conceptual difference between the two projections is that the future ${}_{[o]}\!\downarrow t$ represents the *tree* structure of the clique, whereas $t \downarrow_{[o]}$ just contains the linear subsequence of t which concerns $[o]$. So, the past projection contains more global ordering information than the future projection. Both kinds of projections are needed in the definition of replay; the future projection is used for $\asymp_\Theta$ and the past projection of the equational analog $\dot{\asymp}_\Theta$. In the proof that $\asymp_\Theta$ and $\dot{\asymp}_\Theta$ coincide, we need following property. Obviously, the reverse implication does not hold.

Lemma A.3.12 (Forward and backward)**.** *Assume two weakly balanced traces s and t, and let $[o]$ be a component clique after both s and t. Then, if $s \downarrow_{[o]} = t \downarrow_{[o]}$, then ${}_{[o]}\!\downarrow s = {}_{[o]}\!\downarrow t$. Analogously, if $s \downarrow_{[o]} \preccurlyeq t \downarrow_{[o]}$, then ${}_{[o]}\!\downarrow s \preccurlyeq {}_{[o]}\!\downarrow t$.*

Proof. Straightforward. □

A.4 Soundness

A crucial part of the argument is to determine the common behavior of program and environment acting complementary concerning the interface behavior, and dually, to determine the complementary external behavior from a given common reduction. The lemmas and proofs hold analogously in the concurrent setting for *augmented* traces. In this section, we write also $t, s_1, \ldots$ (instead of $t^+, s_1^+, \ldots$) for augmented traces in the concurrent setting. All proofs are carried through in the more complex setting of augmented traces, but work analogously when removing the sender augmentation from the respective labels (cf. Definition 5.1.4).

The distinction between component and environment concerns not only classes and objects, which cleanly split between both sides, but also the thread. Once the activity of the thread has crossed the interface via a method call, its code is contained both at both sides. Therefore, in order to define the common behavior of two parallel components, one must determine a representation of the *common* code. After all, a component containing the two parts of the thread $n\langle t_1\rangle \parallel n\langle t_2\rangle$ is not even well-typed, and also intuitively, the component and the environment part of a given thread do not "run in parallel". The *merge* $n\langle t_1\rangle \between n\langle t_2\rangle$ is defined in the following section, and used for defining the behavior composition and decomposition.

A.4.1 Merging

Given two pieces $n\langle t_1\rangle$ and $n\langle t_2\rangle$ of the thread, split between program and environment, we need to determine the *common* representation of t_1 and t_2. A thread $n\langle t\rangle$ consists of the stack of method bodies. When split into $n\langle t_1\rangle$ and $n\langle t_2\rangle$ belonging to the two parts of the program, t_1 and t_2 are of specific form: At most one of t_1 or t_2 may be enabled by an internal operational step, while all other stack frames are of the form

$$\mathit{let}\ x{:}T = o_1\ \mathit{blocks\ for}\ o_2\ \mathit{in}(\mathit{let}\ y : T' = t'\ \mathit{in}\ o_1\ \mathit{returns}\ y\ \mathit{to}\ o_3)\ .$$

The only exception is the "oldest" stack frame, i.e., the one where the thread started executing, which is not terminated by a return-statement. See also the

rules for incoming and outgoing method calls from Table 2.11 (resp. 4.8).[9] Now, t_1 and t_2 can be merged or "zipped" together into a common thread $t_1 \between t_2$ by canceling out matching block/return pairs in both stacks.

The following definition and the corresponding proofs carry over to a large extent from the object-based setting [82]. The classes and the connectivity do not impose much additional complication here. The merge of two components, resp., threads is defined as follows:

Definition A.4.1 (Merge). *For a pair of components, respectively, for a pair of threads, $\between$ is the symmetric, partial operator up-to $\equiv$ defined by Table A.5, where*

$$t_{blocked} = let\ x{:}T = \_\ blocks\ for\ o_2\ in\ t_1$$

and furthermore in the last case for $t_{blocked} \between t_2$, the expression e is block/return free and $y \notin fv(t_2)$. Let furthermore t_{stop} abbreviate $stop; t$, a thread where the top-most frame is stopped. The augmentation for the object whose outgoing method call blocked, is irrelevant, and thus indicated by "$\_$". Dually, in the clauses for returns in Table A.5, the object to return to is not important and either left unspecified.

$$\begin{array}{rcll}
\mathbf{0} \between C & \equiv & C & \\
(\nu(n{:}T).C_1) \between C_2 & \equiv & \nu(n{:}T).(C_1 \between C_2) & n \notin fn(C_2) \\
(o[c,F] \parallel C_1) \between C_2 & \equiv & o[c,F] \parallel (C_1 \between C_2) & \\
(c[\![O]\!] \parallel C_1) \between C_2 & \equiv & c[\![O]\!] \parallel (C_1 \between C_2) & \\
(n\langle t\rangle \parallel C_1) \between C_2 & \equiv & n\langle t\rangle \parallel (C_1 \between C_2) & n \notin dom(C_2) \\
(n\langle t_1\rangle \parallel C_1) \between (n\langle t_2\rangle \parallel C_2) & \equiv & n\langle t_1 \between t_2\rangle \parallel (C_1 \between C_2) &
\end{array}$$

$$\begin{array}{rcl}
t_{blocked} \between t_{stop} & \equiv & t_{stop} \\
t_{blocked} \between v & \equiv & v \\
t_{blocked} \between (let\ y{:}T' = o_2\ returns\ v\ to\ _; t'_2) & \equiv & t_1[v/x] \between t'_2 \\
t_{blocked} \between (let\ y{:}T' = o_2\ returns\ v\ to\ _) & \equiv & t_1[v/x] \\
t_{blocked} \between (let\ y{:}T' = e\ in\ t_2) & \equiv & let\ y{:}T' = e\ in \\
 & & \quad (t_{blocked} \between t_2)
\end{array}$$

Table A.5: Merge

The definition is essentially the same as in [82]. Note that the $\between$-operation is partial, i.e., it fails when the two participating components and in particular the two parts of the thread cannot be combined into a common stack.

The rules for $C_1 \between C_2$ in the first part of the table are straightforward. For the empty component, there is nothing to merge. Objects and classes do not participate in the merging, and likewise the scoping operator is ignored. The two parts of the thread are merged and the rest of the components are merged recursively in the last equation. Note that we do not need $n \notin dom(C_1)$ and

[9] The form of the two stacks is determined by the difference of incoming calls minus outgoing returns (the component stack) and the difference of outgoing calls minus incoming returns (for the environment stack). The difference between these two differences ranges always over $\{-1, 0, 1\}$, depending on where the thread is currently active. Lemma A.2.7, but also Lemma A.5.2 later, which connects the balance of a trace with the form of the thread in a component.

$n \notin dom(C_2)$ as side condition in that last equation. By writing $n\langle t\rangle \parallel C$ for the arguments, we implicitly state that $n \notin dom(C)$.

For two pieces of the thread, the $\between$-operator is defined by case analysis on the outermost let-construct (if the thread is not completely stopped). This corresponds to the top-most part of the common stack for both threads. If one of the threads is stopped, the combination of both is stopped, as well, since the stopped thread can never return. If a part of a thread is blocked in its topmost construct, it indicates that it waits for a return of the activity from its partner (cf. rule CALLO, resp., CALLO_0 and RETI). In case the partner is a *value* (in Section 2.6.1, the $return$-syntax is defined as augmentational expression, but not as value), it means, the return will never happen, and the blocked part can be discarded. If otherwise the partner is about to perform the return, the value is handed over, the two topmost let-bindings are thereby popped off, and the merge-operator recurs through the rest of the two threads. The last equation, finally, deals with the situation that the partner is not (yet) a value or a return statement and rearranges the let-binding appropriately.

As far as objects and classes are concerned, merging of two components behaves like their parallel composition. Note that we assume that the only thread occurs at most once in C. The same will apply later analogously in the multi-threaded case for each thread.Thus in the following lemma, the merge of C_1 and C_2 does not concern the thread.

Lemma A.4.2 ($\parallel$ and $\between$)**.** *If* $\Xi \vdash C_1 \parallel C_2$*, then* $C_1 \parallel C_2 \equiv C_1 \between C_2$.

Proof. By induction on the definition of $\between$, i.e., the left-hand sides of the equations as given in Table A.5. The crucial fact underlying the property and the proof is that the thread is contained not both in C_1 and C_2 (in the single-threaded case this is by convention, in the multithreaded case, the type system assures that). This in turn means that the merge-operator is applied only trivially to threads.

Case: $\mathbf{0} \between C$
0 is the neutral element both for $\between$ and for $\parallel$ (cf. Table 2.6 and A.5).

Case: $((\nu n{:}T).C_1) \between C_2$,
where $n \notin fn(C_2)$. The merge is then given by $(\nu n{:}T).(C_1 \between C_2)$, which by induction is equivalent to $(\nu n{:}T).(C_1 \parallel C_2)$ which furthermore yields $((\nu n{:}T).C_1) \parallel C_2)$ by the rules for structural congruence from Table 2.6, since $n \notin fn(C_2)$.

Case: $(o[c, F] \parallel C_1) \between C_2$, $(c[O] \parallel C_1) \between C_2$, and $(n\langle t\rangle \parallel C_1) \between C_2$
All three cases by straightforward induction, using associativity of the $\parallel$-operator.

Case: $(n\langle t_1\rangle \parallel C_1) \between (n\langle t_2\rangle \parallel C_2)$
In this case, the component $(n\langle t_1\rangle \parallel C_1) \parallel (n\langle t_2\rangle \parallel C_2)$ is not well-formed: for the $\parallel$-operator, we do not allow parallel composition of named threads. □

For a thread, occurring on both sides of the $\between$-operator, the respective stack frames are "zipped" into a common stack, if possible. As a consequence: If $C = C_1 \between C_2$ is defined, then an external step of the thread is *enabled* for C exactly if the step is enabled for C_1 or for C_2. For the special case of barbing, this is expressed in the following lemma: $C_1 \between C_2$ strongly barbs on c_b if one of the constituents strongly barbs on c_b. We cannot conclude, however, that *either* C_1 or C_2 strongly barbs on c_b, as two *different* threads in C_1, resp., C_2 may be

about to report success. In the sequential setting, we know stronger that *exactly* one of C_1 and C_2 barbs on c_b, as the other component must be blocked.

Lemma A.4.3 (Merging and barbing)**.** *Assume $C \equiv C_1 \between C_2$. Then $C \downarrow_{c_b}$ iff. $C_1 \downarrow_{c_b}$ or $C_2 \downarrow_{c_b}$. In the sequential setting, the "or" can be strengthened to an "either-or".*

Proof. By induction on the definition of $C_1 \between C_2$, where we omit symmetric cases. Before we start, recall the definition of barbing in equation (2.3), stipulating that $C \downarrow_{c_b}$ if C is structurally congruent to the component $C_b \triangleq \nu(\vec{n}{:}\vec{T}, b{:}c_b).\ C' \parallel n'\langle \mathit{let}\ x{:}\mathit{none}\ \mathit{in}\ b.\mathit{succ}()\ \mathit{in}\ t\rangle$.

Case: $C_1 \between \mathbf{0}$
Since by definition of structural congruence, resp., by assumption, $C_1 \between \mathbf{0} \equiv C_1 \equiv C_b$, the case is immediate.

Case: $C = (\nu(n{:}T).C_1) \between C_2 \equiv \nu(n{:}T).(C_1 \between C_2)$,
where $n \notin \mathit{dom}(C_2)$. Assume $C \downarrow_{c_b}$, i.e., $\nu(n{:}T).(C_1 \between C_2) \equiv C_b$. The case follows straightforwardly by induction, using the properties of $\equiv$.

Case: $C = (o[c, F] \parallel C_1) \between C_2 \equiv o[c, F] \parallel (C_1 \between C_2)$
Assume $C \downarrow_{c_b}$, i.e., $o[c, F] \parallel (C_1 \between C_2) \equiv C_b$. This implies that $(C_1 \between C_2) \downarrow_{c_b}$. Hence by induction, $C_1 \downarrow_{c_b}$ or $C_2 \downarrow_{c_b}$, and thus $(C_1 \parallel o[c, F]) \downarrow_{c_b}$ or $C_2 \downarrow_{c_b}$, as required.

For the reverse direction, assume $(o[c, F] \parallel C_1) \downarrow_{c_b}$, which implies $C_1 \downarrow_{c_b}$, and thus by induction $(C_1 \between C_2) \downarrow_{c_b}$, from which the case follows. The argument for the second alternative, starting from $C_1 \downarrow_{c_b}$, is similar.

Case: $(c[\![O]\!] \parallel C_1) \between C_2 \equiv c[\![O]\!] \parallel (C_1 \between C_2)$
Analogous.

Case: $(n\langle t\rangle \parallel C_1) \between C_2 \equiv n\langle t\rangle \parallel (C_1 \between C_2)$,
where $n \notin \mathit{dom}(C_1, C_2)$. Analogous.

Case: $(n\langle t_1\rangle \parallel C_1) \between (n\langle t_2\rangle \parallel C_2) \equiv n\langle t_1 \between t_2\rangle \parallel (C_1 \between C_2)$,
where $n \notin \mathit{dom}(C_1, C_2)$. First assume $(C_1 \between C_2) \downarrow_{c_b}$. Then the case follows by induction, as in the previous cases. The only interesting case is when $C' = C_1 \between C_2$, i.e.,

$$n\langle t_1 \between t_2\rangle = n\langle \mathit{let}\ x{:}\mathit{none}\ \mathit{in}\ b.\mathit{succ}()\ \mathit{in}\ t\rangle\ .$$

In this case, the last clause of Table A.5 applies, so that

$$n\langle t_1\rangle = n\langle \mathit{let}\ y{:}T = o_1\ \mathit{blocks\ for}\ o_2\ \mathit{in}\ t_1'\rangle$$

and

$$n\langle t_2\rangle = n\langle \mathit{let}\ y{:}\mathit{none} = b.\mathit{succ}()\ \mathit{in}\ t_2'\rangle,$$

which means $(n\langle t_2\rangle \parallel C_2) \downarrow_{c_b}$. The reverse direction is analogous. □

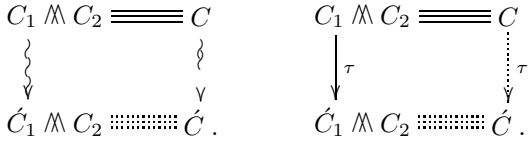
A.4.2 Trace composition

The communication labels for external behavior are strictly dual, and also each rule from Table 2.11 (resp. Table 4.8) has a dual counterpart. Since the labeled transitions describe exactly the interface behavior, two component fitting together (in the sense of being mergeable and engaging in exactly dual sequences of external steps) can perform together a sequence of internal steps.

Lemma A.4.4 (⩚ and $\rightsquigarrow$-step)**.** *Assume* $C_1 \between C_2 \equiv C$

1. *If* $C_1 \rightsquigarrow \acute{C}_1$*, then* $C \rightsquigarrow \acute{C}$ *with* $\acute{C}_1 \between C_2 \equiv \acute{C}$*, for some component* $\acute{C}$*.*
2. *If* $C_1 \xrightarrow{\tau} \acute{C}_1$*, then* $C \xrightarrow{\tau} \acute{C}$ *with* $\acute{C}_1 \between C_2 \equiv \acute{C}$ *for some component* $\acute{C}$*.*

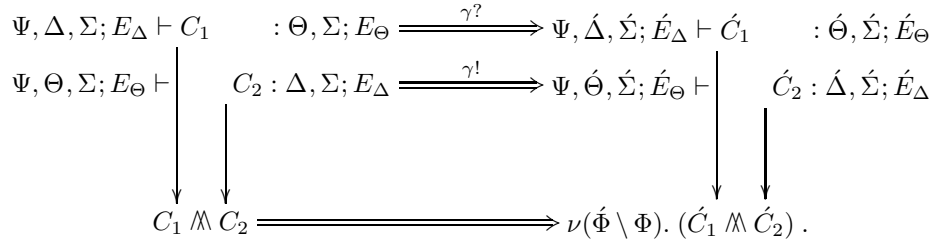
Moreover, the reduction step $C \rightsquigarrow C'$ *concerns the same redex as the step* $C_1 \rightsquigarrow C'_1$*. The same applies to* $\xrightarrow{\tau}$*-steps in part 2. Pictorially:*



Proof. For $\rightsquigarrow$ in part 1, by induction on the length of derivation for $C_1 \rightsquigarrow \acute{C}_1$, using the operational axioms from Table 4.5 (resp. 2.5 in the single-threaded case) and the rules from Tables 2.7 and 2.6, covering structural congruence. The only interesting case for merging is the one where $C_1 \equiv n\langle t_1 \rangle \parallel C'_1$ and $C_2 \equiv n\langle t_2 \rangle \parallel C'_2$, i.e., when the two stacks of the thread, responsible for the $\rightsquigarrow$-step, are actually merged. The fact that both can be merged means, either t_1 can do an internal step and t_2 is blocked waiting for return, or symmetrically (where the symmetrical case cannot be true, since a blocked thread cannot do a $\rightsquigarrow$-step):

Case: $n\langle t_1 \rangle = n\langle \mathit{let}\ y{:}T' = e\ \mathit{in}\ t'_1 \rangle$ and $n\langle t_2 \rangle = n\langle \mathit{let}\ x{:}T = o_1\ \mathit{blocks\ for}\ o_2\ \mathit{in}\ t'_2 \rangle$ where $n\langle \mathit{let}\ y{:}T' = e\ \mathit{in}\ t'_1 \rangle \rightsquigarrow n\langle \mathit{let}\ y{:}T' = e'\ \mathit{in}\ t'_1 \rangle$. The merge $n\langle t_1 \rangle \between n\langle t_2 \rangle$ is given by $n\langle \mathit{let}\ y{:}T' = e\ \mathit{in}\ ((\mathit{let}\ x{:}T = o_1\ \mathit{blocks\ for}\ o_2\ \mathit{in}\ t'_2) \between t'_1) \rangle$, from which the case follows. The argument for $\xrightarrow{\tau}$ in part 2 works analogously. □

Lemma A.4.5 (⩚ and communication step)**.** *Assume* $\Psi, \Xi \vdash C_1$ *and* $\Psi, \bar{\Xi} \vdash C_2$ *where* $C_1 \between C_2 \equiv C$*. If* $\Psi, \Xi \vdash C_1 \xrightarrow{\gamma?} \acute{\Delta}, \Psi, \acute{\Xi} \vdash \acute{C}_1$ *and* $\Psi, \bar{\Xi} \vdash C_2 \xrightarrow{\gamma!} \Psi, \bar{\acute{\Xi}} \vdash \acute{C}_2$*, then* $C \equiv \nu(\acute{\Phi} \setminus \Phi).\ \acute{C}_1 \between \acute{C}_2$*. Pictorially:*



Proof. By case analysis on the form of the communication label γ.

Case: rule CALLI$_0$/CALLO, i.e., $\gamma = \nu(\Delta', \Theta', n{:}\mathit{thread}).n\langle [o]\mathit{call}\ o_r.l(\vec{v}) \rangle$ i.e., $\Sigma \not\vdash n$. This case applies only to the multithreaded setting. By CALLI$_0$ and CALLO from Table 4.8 (plus the augmentation from Definition 5.1.4) and the typing rules, the two components are of the following form: C_1 as the receiver does not contain the thread n and C_2 is of the form

$$C_2 \equiv \nu(n{:}\mathit{thread}, \Delta', \Theta', \vec{n}{:}\vec{T}).(C'_2 \parallel n\langle \mathit{let}\ x{:}T = o_s\ o_r.l(\vec{v})\ \mathit{in}\ t \rangle)\ .$$

Note that the sender augmentation o of the label needs not equal the "actual" caller o_s, but some representative of the sender clique such that $\Delta \vdash o$ from the perspective of C_1. Furthermore, the sender to return to in the callee code in C_2, $\odot_n$, need not match the o_s; the connectivity premises of the operational rules, however, guarantee that o, o_s, and $\odot_n$ belong to the same clique. By convention, Δ' contains objects from C_2 and Θ' those lazily instantiated to be contained in C_1 from this communication step on. After $\gamma?$, resp., after $\gamma!$, we get

$$\acute{C}_1 = C_1 \parallel n\langle \mathit{let}\ y{:}T = o_r.l(\vec{v})\ \mathit{in}\ o_r\ \mathit{returns}\ y\ \mathit{to}\ \odot_n\rangle$$

and

$$\acute{C}_2 = \nu(\vec{n}{:}\vec{T}).\ (C_2' \parallel n\langle \mathit{let}\ x{:}T = o_s\ \mathit{blocks\ for}\ o_r\ \mathit{in}\ t\rangle)\ .$$

and for the contexts *after* the step $\acute{\Delta} = \Delta, \Delta'$, $\acute{\Theta} = \Theta, \Theta'$, and $\acute{\Sigma} = \Sigma, n{:}\ \mathit{thread}$. Since the thread n is not contained in C_1 before the communication step and likewise the names from Δ', Θ', and $\vec{n}$ are new for C_1, we get by definition of the merge operator: $C_1 \between C_2 \equiv$

$$\nu(\Delta', \Theta', n{:}\ \mathit{thread}, \vec{n}{:}\vec{T}).\ (C_1 \between C_2' \parallel n\langle \mathit{let}\ x{:}T = o_s o_r.l(\vec{v})\ \mathit{in}\ t\rangle)\ ,$$

as the situation before the step. For the components $\acute{C}_1$ and $\acute{C}_2$ after the common step, the last two clauses in the definition of merging yield

$$\acute{C}_1 \between \acute{C}_2 \equiv \nu(\vec{n}{:}\vec{T}).(C_1 \between C_2' \parallel n\langle \mathit{let}\ y{:}T = o_1 o_2.l(\vec{v})\ \mathit{in}\ t[y/x]\rangle)\ ,$$

which means

$$C_1 \between C_2 \equiv \nu(\acute{\Phi} \setminus \Phi).(\acute{C}_1 \between \acute{C}_2)\ ,$$

as required. The cases for non-initial calls work similarly.

Case: $\gamma = \nu(\Delta', \Theta').n\langle \mathit{return}(v)\rangle$

Note that unlike the cases for method calls, the thread name cannot be transmitted boundedly. Furthermore, as we only transmit a single (non-compound) value v and not a vector as for method calls, Δ' or Θ' is empty (or both). For uniformity, we treat them both in one case. By the operational rules for external steps from Table 4.8 (resp. Table 2.11 in the sequential setting), the components must be of the following forms:

$$\begin{array}{rcl} C_1 & \equiv & \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle t_1\rangle \\ & = & \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle \mathit{let}\ x{:}T = o_r^1\ \mathit{blocks\ for}\ o_s\ \mathit{in}\ t_1'\rangle \end{array}$$

and

$$\begin{array}{rcl} C_2 & \equiv & \nu(\Delta', \Theta', \vec{n}_2{:}\vec{T}_2).\ C_2' \parallel n\langle t_2\rangle \\ & \equiv & \nu(\Delta', \Theta', \vec{n}_2{:}\vec{T}_2).\ C_2' \parallel n\langle \mathit{let}\ x{:}T = o_s\ \mathit{returns}\ v\ \mathit{to}\ o_r^2; t_2'\rangle \end{array}$$

After the step, the components look as follows (cf. rules RETI and RETO):

$$\acute{C}_1 \equiv \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle t_1'[v/x]\rangle \quad \text{and} \quad \acute{C}_2 \equiv \nu(\vec{n}_2{:}\vec{T}_2).\ C_2' \parallel n\langle t_2'\rangle\ .$$

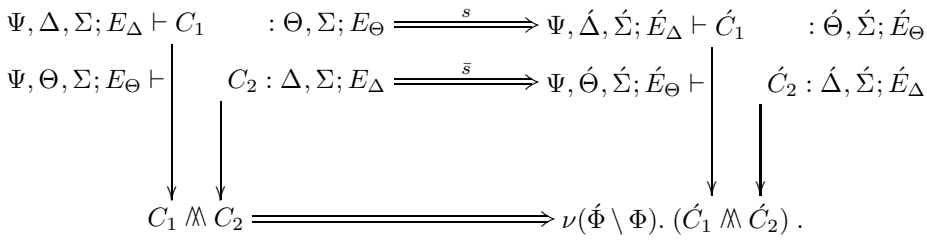
For the update of contexts, we get $\acute{\Delta} = \Delta, \Delta'$, $\acute{\Theta} = \Theta, \Theta'$, and Σ containing the thread names remains unchanged, i.e., $\acute{\Sigma} = \Sigma$. Since the names from $\vec{n}_2$, Δ', and from Θ' are new for C_1 and $\vec{n}_1$ new for C_2, the definition of $\between$ for

components and for threads (and using symmetry) gives (we assume that $\between$ has a stronger binding power than $\|$):

$$\begin{array}{rcl} C_1 \between C_2 & \equiv & \nu(\Delta', \Theta', \vec{n}_1{:}\vec{T}_1, \vec{n}_2{:}\vec{T}_2).\ (C_1' \between C_1' \parallel n\langle t_1 \between t_2\rangle) \\ & \equiv & \nu(\Delta', \Theta', \vec{n}_1{:}\vec{T}_1, \vec{n}_2{:}\vec{T}_2).\ (C_1' \between C_1' \parallel n\langle t_1'[v/x] \between t_2'\rangle) \\ & \equiv & \nu(\Delta', \Theta').\ \nu(\vec{n}_1{:}\vec{T}_1, \vec{n}_2{:}\vec{T}_2).\ (C_1' \between C_1' \parallel n\langle t_1'[v/x] \between t_2'\rangle) \\ & \equiv & \nu(\acute{\Phi} \setminus \Phi).(\nu(\vec{n}_1{:}\vec{T}_1).(C_1' \parallel n\langle t_1'[v/x]\rangle) \between \nu(\vec{n}_2{:}\vec{T}_2).(C_2' \parallel n\langle t_2'\rangle)) \\ & = & \nu(\acute{\Phi} \setminus \Phi).\ (\acute{C}_1 \between \acute{C}_2)\ , \end{array}$$

which concludes the case. □

Lemma A.4.6 (Trace composition)**.** *Assume $\Psi, \Xi \vdash C_1$ and $\Psi, \bar{\Xi} \vdash C_2$ with $C_1 \between C_2 \equiv C$. If $\Psi, \Xi \vdash C_1 \stackrel{s}{\Longrightarrow} \Psi, \acute{\Xi} \vdash \acute{C}_1$ and $\Psi, \bar{\Xi} \vdash C_2 \stackrel{\bar{s}}{\Longrightarrow} \Psi, \bar{\acute{\Xi}} \vdash \acute{C}_2$, then $C \Longrightarrow \acute{C}$ where $\acute{C} \equiv \nu(\acute{\Phi} \setminus \Phi).\ \acute{C}_1 \between \acute{C}_2$ (remember: Φ is the name-binding part of Ξ, and $\acute{\Phi}$ that of $\acute{\Xi}$). Pictorially:*



Proof. By induction on the length of reduction (cf. Table 3.1), using subject reduction from Lemma A.1.1 and the two parts of Lemma A.4.4 dealing with $\rightsquigarrow$-steps resp. τ-steps of one of the partners, and Lemma A.4.5, dealing with communication between the partners, resolved in a common τ-step. □

A.4.3 Trace decomposition

This section contains the opposite property of the previous section: A reduction sequence of a component consisting of two sub-constituents can be decomposed or "torn apart" into two complementary reduction sequences. In the concurrent setting, we are working with *augmented traces*, and in decomposition, both partners must agree also in the sender augmentation (as we required also for trace composition). As for trace composition, the development is basically equivalent to the one in the object-based setting of [82].

Lemma A.4.7 (Expansion)**.** *Let e_1 be an expression containing neither block nor return statements and let t_2 be of the form let $x_2{:}T_2 = o_1$ blocks for o_2 in t_2'. If $t_1 \between t_2 \equiv t$, then $n\langle let\ x_1{:}T_1 = e_1\ in\ t_1\rangle \between n\langle t_2\rangle = n\langle let\ x_1{:}T_1 = e_1\ in\ t\rangle$.*

Proof. Immediate, by inspection of the rules, in particular the last one from the second part of Table A.5. □

Lemma A.4.8 (Expansion)**.** *Assume $\Psi, \Xi \vdash C_1' \parallel n\langle t_1\rangle$ and $\Psi, \bar{\Xi} \vdash C_2$ and let e be an expression containing neither block nor return statements. If $(C_1' \parallel n\langle t_1\rangle) \between C_2 \equiv C' \parallel n\langle t\rangle$, then $C_1' \parallel n\langle let\ x{:}T = e\ in\ t_1\rangle \between C_2 \equiv C' \between n\langle let\ x{:}T = e\ in\ t\rangle$.*

Proof. If C_2 does not contain n, then $n\langle t_1\rangle \equiv n\langle t\rangle$, and the result holds trivially.

Otherwise, $C_2 \equiv C_2' \parallel n\langle t_2\rangle$, where the thread n does not occur in the rest C_2'. Let furthermore C_1 abbreviate $C_1' \parallel n\langle t_1\rangle$. In this case, the derivation of the assumption $C_1 \between C_2 \equiv C' \parallel n\langle t\rangle$ implies $C_1' \between C_2' \equiv C'$ and $n\langle t_1\rangle \between n\langle t_2\rangle \equiv n\langle t\rangle$. Thus we obtain with the help of Lemma A.4.7

$$\begin{array}{lcl} C_1' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ \ t_1\rangle \between C_2 & \equiv & (C_1' \parallel (n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ \ t_1\rangle) \between (C_2' \parallel n\langle t_2\rangle) \\ & \equiv & (C_1' \between C_2') \parallel (n\langle \mathit{let}\ x{:}T = e\ \ \mathit{in}\rangle\ t_1 \between n\langle t_2\rangle) \\ & \equiv & (C_1' \between C_2') \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ \ t\rangle \\ & \equiv & C' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ \ t\rangle\ , \end{array}$$

as required. □

Lemma A.4.9 (Decomposition and top redex)**.** *If* $\Psi, \Xi \vdash C_1$ *and* $\Psi, \bar{\Xi} \vdash C_2$ *where* $C_1 \between C_2 \equiv \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t\rangle)$*, then*

$$\Psi, \Xi \vdash C_1 \stackrel{s}{\Longrightarrow} \Psi, \acute{\Xi} \vdash \acute{C}_1 \quad \textit{and} \quad \Psi, \bar{\Xi} \vdash C_2 \stackrel{\bar{s}}{\Longrightarrow} \Psi, \bar{\acute{\Xi}} \vdash \acute{C}_2$$

(or symmetrically) with

$$\acute{C}_1 = \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ \ t_1\rangle\ ,$$

and where $\nu(\acute{\Phi} \setminus \Phi).\nu(\vec{n}_1{:}\vec{T}_1).(C_1' \parallel n\langle t_1\rangle) \between \acute{C}_2 \equiv \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle t\rangle)$*, with* $\Phi = \Delta, \Theta$ *and* $\acute{\Phi} = \acute{\Delta}, \acute{\Theta}$*.*

Proof. By induction on the definition of $\between$. We show a few exemplary cases, especially the one of matching block/return.

Case: $C_1 \between C_2 = C_1 \between \mathbf{0} \equiv C_1 = \nu(\vec{n}{:}\vec{T}).\ (C_1' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t\rangle)$
In this case, s and $\bar{s}$ are empty, $\vec{n}_1 = \vec{n}$, and $t_1 = t$.

Case: $C_1 \between C_2 = (\nu(n{:}T).\tilde{C}_1) \between C_2 \equiv \nu(n{:}T).(C_1 \between C_2)$,
where $n \notin \mathit{fn}(C_2)$. By induction.

Case: $C_1 \equiv n\langle \mathit{let}\ x_1{:}T_1 = o\ \mathit{blocks\ for}\ o_2\ \mathit{in}\ t_1\rangle$ and
$C_2 \equiv n\langle \mathit{let}\ x_2{:}T_2 = o_2\ \mathit{returns}\ v\ \mathit{to}\ o'; t_2\rangle$,
with $C_1 \between C_1 \equiv n\langle t_1[v/x]\rangle \between n\langle t_2\rangle \equiv \nu(\vec{n}{:}\vec{T}).(C \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t\rangle)$. Note that o' and o need not be equal for the merge. By the two complementary operational rules RETI and RETO of Table 4.8, resp., of 2.11:

$$\Psi, \Xi \vdash C_1 \stackrel{\gamma_r?}{\longrightarrow} \Psi, \Xi \vdash n\langle t_1[v/x]\rangle \quad \text{and} \quad \Psi, \bar{\Xi} \vdash C_2 \stackrel{\gamma_r!}{\longrightarrow} \Psi, \bar{\Xi} \vdash n\langle t_2\rangle\ ,$$

with $\gamma_r = n\langle \mathit{return}(v)\rangle$. Applying the induction hypothesis to the two post-configurations after the return-step gives:

$$\begin{array}{ll} & n\langle t_1[v/x]\rangle \quad \stackrel{s}{\Longrightarrow} \Psi, \acute{\Xi} \vdash \nu(\vec{n}_1{:}\vec{T}_1).(C_1' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t_1\rangle) \\ \text{and} & \\ & \Psi, \bar{\Xi} \vdash n\langle t_2\rangle \quad \stackrel{\bar{s}}{\Longrightarrow} \Psi, \bar{\acute{\Xi}} \vdash \acute{C}_2\ , \end{array}$$

where $\nu(\acute{\Phi} \setminus \Phi).\nu(\vec{n}_1{:}\vec{T}_1).(C_1' \parallel n\langle t_1\rangle) \between \acute{C}_2 \equiv \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle t\rangle)$ (or symmetrically), which yields the result. □

Lemma A.4.10 (Decomposition and $\rightsquigarrow$-step)**.** *If $C_1 \between C_2 \equiv C$ and $C \rightsquigarrow \acute{C}$, then there exists a trace s such that $\Psi, \Xi \vdash C_1 \stackrel{s}{\Longrightarrow} \Psi, \acute{\Xi} \vdash \acute{C}_1$ and $\Phi, \bar{\Xi} \vdash C_2 \stackrel{\bar{s}}{\Longrightarrow} \Psi, \bar{\acute{\Xi}} \vdash \acute{C}_2$, where $\nu(\acute{\Phi} \setminus \Phi)$. $\acute{C}_1 \between \acute{C}_2 \equiv \acute{C}$ and where $\Phi = \Delta, \Sigma, \Theta$ and $\acute{\Phi} = \acute{\Delta}, \acute{\Sigma}, \acute{\Theta}$. Pictorially:*

$$\begin{array}{ccc} C_1 \between C_2 & = & C \\ \bar{s} \vdots\, s & & \wr \\ \nu(\acute{\Phi} \setminus \Phi).\ (\acute{C}_1 \between \acute{C}_2) & \cdots & \acute{C}\ . \end{array}$$

Proof. We start by looking at the form of the component C. It is able to do an immediate $\rightsquigarrow$-step, which means, some thread in C executes a top-most let-command:[10]

$$C \equiv \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle \mathit{let}\, x{:}T = e\, \mathit{in}\, t\rangle) \rightsquigarrow \nu(\vec{n}{:}\vec{T}, \vec{n}'{:}\vec{T}').\ (C' \parallel C'' \parallel n\langle \acute{t}\rangle) \equiv \acute{C}\ .$$

By Lemma A.4.9 we know that

$$\Psi, \Xi \vdash C_1 \stackrel{s}{\Longrightarrow} \Psi, \acute{\Xi} \vdash \tilde{C}_1$$

with $\tilde{C}_1 = \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle \mathit{let}\, x{:}T = e\, \mathit{in}\, t_1\rangle$ and

$$\Psi, \bar{\Xi} \vdash C_2 \stackrel{\bar{s}}{\Longrightarrow} \Psi, \bar{\acute{\Xi}} \vdash \acute{C}_2$$

(or symmetrically) and where furthermore $\nu(\acute{\Phi} \setminus \Phi).(\nu(\vec{n}_1{:}\vec{T}_1)(C_1' \parallel n\langle t_1\rangle)) \between \acute{C}_2 \equiv \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle t\rangle)$. Thus by Lemma A.4.8,

$$\begin{array}{c} \nu(\acute{\Phi} \setminus \Phi).(\nu(\vec{n}_1{:}\vec{T}_1).(C_1' \parallel n\langle \mathit{let}\, x{:}T = e\ \mathit{in}\ t_1\rangle)) \between \acute{C}_2 \equiv \\ \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle \mathit{let}\, x{:}T = e\ \mathit{in}\ t\rangle) = C\ . \end{array}$$

Now define $\acute{C}_1$ as the component after performing the redex of the thread which corresponds to the step $C \rightsquigarrow C'$, i.e.,

$$\tilde{C}_1 = \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle \mathit{let}\, x{:}T = e\, \mathit{in}\, t_1\rangle \rightsquigarrow \nu(\vec{n}_1{:}\vec{T}_1, \vec{n}'{:}\vec{T}').\ C_1' \parallel C'' \parallel n\langle \acute{t}_1\rangle \triangleq \acute{C}_1\ ,$$

and furthermore $\acute{C}_2 \triangleq C_2'$. Thus with the help of Lemma A.4.4, $\acute{C}_1 \between \acute{C}_2 = \acute{C}$, as required. □

Lemma A.4.11 (Decomposition and τ-step)**.** *If $C_1 \between C_2 \equiv C$ and $C \xrightarrow{\tau} \acute{C}$, then there exists a trace s such that $\Psi, \Xi \vdash C_1 \stackrel{s}{\Longrightarrow} \Psi, \acute{\Xi} \vdash \acute{C}_1$ and $\Psi, \bar{\Xi} \vdash C_2 \stackrel{\bar{s}}{\Longrightarrow} \Psi, \bar{\acute{\Xi}} \vdash \acute{C}_2$, where $\nu(\acute{\Phi} \setminus \Phi)$. $\acute{C}_1 \between \acute{C}_2 \equiv \acute{C}$. Pictorially:*

$$\begin{array}{ccc} C_1 \between C_2 & = & C \\ \bar{s} \vdots\, s & & \downarrow \tau \\ \nu(\acute{\Phi} \setminus \Phi).\ (\acute{C}_1 \between \acute{C}_2) & \cdots & \acute{C}\ . \end{array}$$

[10] Note that it is an invariant of the semantics that e is block/return-free. The internal semantics is formulated without block and return statements, which have been introduced as augmentational syntax to formulate the external semantics.

Proof. The component C is able to do an immediate τ-step, which is either a method call or a field update; in both cases, the τ-step looks as follows (in case of a method call, $F' = F$):

$$\begin{array}{lcl} C & \equiv & \nu(\vec{n}{:}\vec{T}).(C' \parallel o[c,F] \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t\rangle) \\ & \xrightarrow{\tau} & \nu(\vec{n}{:}\vec{T}).(C' \parallel o[c,F'] \parallel n\langle \mathit{let}\ x{:}T = e'\ \mathit{in}\ t\rangle) \equiv \acute{C}\ . \end{array}$$

As in the corresponding proof of Lemma A.4.10 for $\rightsquigarrow$-steps, Lemma A.4.9 yields

$$\Psi, \Xi \vdash C_1 \stackrel{s}{\Longrightarrow} \Psi, \acute{\Xi} \vdash C_1'$$

with $C_1' = \nu(\vec{n}_1{:}\vec{T}_1).\ (C_1'' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ \ t_1\rangle)$ and

$$\Psi, \bar{\Xi} \vdash C_2 \stackrel{\bar{s}}{\Longrightarrow} \Psi, \bar{\acute{\Xi}} \vdash C_2'$$

and where $\nu(\acute{\Phi} \setminus \Phi).(\nu(\vec{n}_1{:}\vec{T}_1)(C_1'' \parallel n\langle t_1\rangle)) \between C_2' \equiv \nu(\vec{n}{:}\vec{T}).\ (C' \parallel o[c,F] \parallel n\langle t\rangle)$.

Now we distinguish, whether the object o belongs to C_1' or C_2', i.e., whether the τ-step in question is an internal step of C_1' or whether it is a synchronization step of both C_1' and C_2'.

Case: $o \in \mathit{dom}(C_1')$
Thus, $C_1'' \equiv \nu(\vec{n}_1''{:}\vec{T}_3'').(C_1''' \parallel o[c,F])$, and hence

$$C_1' \equiv \nu(\vec{n}_1{:}\vec{T}_1, \vec{n}_1''{:}\vec{T}_3'').(C_1''' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t_1\rangle \parallel o[c,F])\ .$$

Then define $\acute{C}_1$ as the component after executing the redex of the thread, which corresponds to the step $C \xrightarrow{\tau} \acute{C}$, i.e.,

$$\begin{array}{lcl} C_1' & \equiv & \nu(\vec{n}_1{:}\vec{T}_1).\ (C_1'' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t_1\rangle) \\ & \equiv & \nu(\vec{n}_1{:}\vec{T}_1, \vec{n}_1''{:}\vec{T}_1'').\ (C_1''' \parallel o[c,F] \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t_1\rangle) \\ & \xrightarrow{\tau} & \nu(\vec{n}_1{:}\vec{T}_1, \vec{n}_1''{:}\vec{T}_1'').\ (C_1''' \parallel o[c,F'] \parallel n\langle \mathit{let}\ x{:}T = e'\ \mathit{in}\ t_1\rangle) \\ & \triangleq & \acute{C}_1\ . \end{array}$$

With the help of the composition Lemma A.4.4, $\acute{C}_1 \between \acute{C}_2 \equiv \acute{C}$, as required.

Case: $o \notin \mathit{dom}(C_1')$
In this case, a communication step takes place between C_1' and C_2'. Since we do not allow direct field update across object boundaries, a method update cannot cross component boundaries and the τ-step must be a method call. Note also that the common step cannot be a return communication: Originally performed by the global component C, values are not returned by τ-steps. Therefore, the τ-step of C looks in particular as follows:

$$\begin{array}{lcl} C & \equiv & \nu(\vec{n}{:}\vec{T}).(C'' \parallel o[c,F] \parallel c[\![F'', M]\!] \parallel n\langle \mathit{let}\ x{:}T = o'.l(\vec{v})\ \mathit{in}\ t\rangle) \\ & \xrightarrow{\tau} & \nu(\vec{n}{:}\vec{T}).(C'' \parallel o[c,F] \parallel c[\![F'', M]\!] \parallel n\langle \mathit{let}\ x{:}T = M.l(o)(\vec{v})\ \mathit{in}\ t\rangle) \equiv \acute{C}\ . \end{array}$$

We further distinguish whether the thread enters C_2' for the first time by the method call or not.

Subcase: $n \in dom(C_2')$
In this case, corresponding to a CALLI$_1$-step, resp., CALLI$_2$ for C_2', the thread is already contained blocked, resp., stopped within C_2', i.e., $C_2' \equiv$

$$\nu(\vec{n}_2{:}\vec{T}_2).\ (C_2''' \parallel o[c,F] \parallel c[\![F'', M]\!] \parallel n\langle let\ x_2{:}T_2 = o_2\ blocks\ for\ o_3\ in\ t_2\rangle)\ , \tag{A.32}$$

which corresponds to the case for CALLI$_1$. The one for CALLI$_2$, when the thread is stopped within C_2' is analogous. Now, the component C_1 can perform the following trace:

$$\begin{array}{ll} \Psi, \Xi \vdash C_1 & \stackrel{s}{\Longrightarrow} \\ \Psi, \acute{\Xi} \vdash C_1' & \stackrel{\nu(\Phi').n\langle call\ o.l(\vec{v})\rangle!}{\longrightarrow} \\ \Psi, \acute{\Xi}' \vdash \acute{C}_1 & \end{array}$$

with $\Phi' \not\vdash n$, and

$$\acute{C}_1 \equiv \nu(\vec{n}_1''{:}\vec{T}_1'').\ (C_1'' \parallel n\langle let\ x{:}T = o'\ blocks\ for\ o\ in\ t_1\rangle)$$

where $(\vec{n}_1''{:}\vec{T}_1'') = (\vec{n}_1'{:}\vec{T}_1) \setminus \Phi'$, and where the contexts of $\acute{\Xi}'$ are determined by the context update according to CALLO from Table 2.11. For the communication partner C_2, using CALLI$_2$ in the last step, we furthermore obtain the trace:

$$\begin{array}{ll} \Psi, \bar{\Xi} \vdash C_2 & \stackrel{\bar{s}}{\Longrightarrow} \\ \Psi, \bar{\acute{\Xi}} \vdash C_2' & \stackrel{\nu(\Phi').n\langle call\ o.l(\vec{v})\rangle?}{\longrightarrow} \stackrel{\tau}{\longrightarrow} \\ \Psi, \bar{\acute{\Xi}}' \vdash \acute{C}_2\ , & \end{array}$$

where

$$\acute{C}_2 \equiv \nu(\vec{n}_2{:}\vec{T}_2).\ (C_2''' \parallel o[c,F] \parallel c[\![F'', M]\!] \parallel n\langle \acute{t}_2\rangle$$

and with

$$\begin{array}{rcl} \acute{t}_2 & = & let\ x_2'{:}T = M.l(o)(\vec{v})\ in\ o\ returns\ x_2'\ to\ o_3; \\ & & let\ x_2{:}T_2 = o_2\ blocks\ for\ o_3\ in\ t_2\ . \end{array} \tag{A.33}$$

Note that the input rule CALLI$_1$ is dual to CALLO as far as the update of the assumption and commitment contexts is concerned. As for the role of the calling object, in this case o': Being not transmitted from the caller to the callee as part of the (augmented) label, the code augmentation do *not* agree on the exact identity of the caller (o' vs. o_3), the corresponding premises of the call-rules assure that o' and o_3 belong to the same clique. Thus, the above step is possible where the assumption and commitment contexts of $\acute{C}_2$ match the ones for $\acute{C}_1$. As in the previous case, one can show $\acute{C}_1 \between \acute{C}_2 \equiv \acute{C}$, as required.

Subcase: $n \notin dom(C_2')$
Analogous. This corresponds to a combination of CALLO (as in the previous subcase) and CALLI$_0$. The call label then is *augmented*

$$\nu(\Phi').n\langle[o'']call\ o.l(\vec{v})\rangle!$$

resp. the dual input label. The component C_2' does not contain the thread n yet (cf. equation (A.32), and $\acute{t}_2$ looks as follows (cf. equation (A.33)):

$$\acute{t}_2 \quad = \quad let\ x_2'{:}T = M.l(o)(\vec{v})\ in\ o\ returns\ x_2'\ to\ \odot_n; stop\ . \tag{A.34}$$

Note that o'' in the call label augmentation corresponds neither to the actual sender o' nor the sender $\odot_n$ as stored in the code of the receiver. However, both L-CALLI$_0$ and L-CALLO can guess the same o'', as representative of the clique of o' and $\odot_n$. □

Lemma A.4.12 (Trace decomposition). *Assume* $\Psi, \Xi \vdash C_1$ *and* $\Psi, \bar{\Xi} \vdash C_2$ *with* $C_1 \between C_2 \equiv C$. *If* $C \Longrightarrow C'$, *then*

$$\Psi, \Xi \vdash C_1 \stackrel{t}{\Longrightarrow} \Psi, \Xi' \vdash C_1' \quad \textit{and} \quad \Psi, \bar{\Xi} \vdash C_2 \stackrel{\bar{t}}{\Longrightarrow} \Phi, \bar{\Xi}' \vdash C_2' \,,$$

for some augmented trace s *where* $C' \equiv \nu(\Phi' \setminus \Phi).\ C_1' \between C_2'$.

Proof. The property follows directly by induction on the number of internal steps from Lemma A.4.10 and A.4.11. □

A.4.4 Soundness

In the proof, as well as the one for completeness, a component is interacting with a surrounding program context, i.e., both do complementary actions. See Section 3.1 for the definition of $\bar{t}$, the complementary trace of t.

Complementary traces describe the situation where component and environment can act together and where the complementary communication steps cancel out into internal behavior. When putting together two components, their respective domains are disjoint wrt. named objects and classes. This disjointness does not hold, however, for the code of the thread, since each half of the program contains its share of the thread, with all the blocked method bodies (except one) "stacked" one upon the other with the let construct.

To compose two components into a common one, the two "halves" of each stack must be merged ("zipped") to form one combined stack. Given two components, we write $C_1 \between C_2$ for the result of the merging. Informally, $C_1 \between C_2$ can be understood as $C_1 \parallel C_2$, with the exception of the thread, where the parallel composition $n\langle t_1 \rangle$ and $n\langle t_2 \rangle$ is resolved into a single stack $n\langle t_1 \between t_2 \rangle$. The definition is basically equivalent to the one in [82]

Proof of Soundness (Lemma 3.2.1). We have to show that if $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$, then $\Xi_0 \vdash C_1 \sqsubseteq_{obs} C_2$.

Assume $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$ and $\bar{\Xi}_0, c_b{:}\mathsf{barb} \vdash C_0$ as observer for C_1, resp., for C_2, where $\bar{\Xi}_0$ denotes Ξ_0 with the roles of assumption and commitment exchanged. We further assume $(C_1 \parallel C_0) \Downarrow_{c_b}$, i.e., $(C_1 \parallel C_0) \Longrightarrow C' \downarrow_{c_b}$ for some component C'. The parallel composition $C_1 \parallel C_0$ is well-typed (justified by T-PAR; in particular, the initial configuration contains exactly one mentioning of the thread n, and not two). Hence the merging Lemma A.4.2 gives that $C_1 \parallel C_0 \equiv C_1 \between C_0$. Further by decomposition (Lemma A.4.12), C_0 and C_1 can perform complementary traces, i.e.,

$$\Xi_0, c_b{:}\mathsf{barb} \vdash C_1 \stackrel{t_1}{\Longrightarrow} \Xi', c_b{:}\mathsf{barb} \vdash C_1' \tag{A.35}$$

and

$$\bar{\Xi}_0, c_b{:}\mathsf{barb} \vdash C_0 \stackrel{\bar{t}_1}{\Longrightarrow} \bar{\Xi}', c_b{:}\mathsf{barb} \vdash C_0' \tag{A.36}$$

where $C' \equiv \nu(\Phi' \setminus \Phi).C'_0 \between C'_1$. By Lemma A.4.3, $C' \downarrow_{c_b}$ implies that either C'_1 or C'_0 strongly barbs on c_b. Assume that $\Xi_0 \vdash t_1 : det_\Delta$, otherwise there is nothing to show.[11]

Case: $C'_1 \downarrow_{c_b}$
The case, where the component itself reports success, cannot occur: C_1 is well-typed in a context without c_b, i.e., $\Xi_0 \vdash C_1$. This means, C_1 cannot itself instantiate an object of class c_b. Neither is it possible that its partner C_0 transmits to C_1 a reference to such an object in the trace t_1, which would allow C_1 it invoke the success-method and report success thereby.[12] To transmit the reference from C_0 to C_1 would require that C_1 contained a method whose type would mention c_b (either as argument or as return type), which cannot be the case.

Case: $C'_0 \downarrow_{c_b}$, i.e., $\bar{\Xi}', c_b{:}\mathsf{barb} \vdash C'_0 \xrightarrow{succ}$
where (in abuse of notation) the success-reporting external label $succ$ is of the form $\nu(b{:}c_b).n\langle call\ b.succ()\rangle!$. Since C_1 is well-typed in Ξ_0, the reduction of (A.35) can be carried out also in the tightened context Ξ_0, i.e., $\Xi_0 \vdash C_1 \stackrel{t_1}{\Longrightarrow} \Xi' \vdash C'_1$. Hence the definition of $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$ (cf. Definition 3.1.11) gives

$$\Xi_0 \vdash C_2 \stackrel{t_2}{\Longrightarrow} \Xi'' \vdash C''_2.$$

for some trace t_2 with $\Xi_0 \vdash t_2 \asymp_\Delta t_1$ and with $\vdash t_2 : det_\Delta$. Dualizing with Lemma A.2.56 yields $\bar{\Xi}_0 \vdash \bar{t}_2 \asymp_\Theta \bar{t}_1$ with $\vdash \bar{t}_2 \vdash: det_\Theta$ with $\bar{\Xi}_0 \vdash \bar{t}_2 : det_\Theta$. Since neither t_1 nor t_2 can mention c_b, we obtain $\bar{\Xi}_0, c_b{:}\mathsf{barb} \vdash \bar{t}_2 \asymp_\Theta \bar{t}_1$ with $\bar{\Xi}_0, c_b{:}\mathsf{barb} \vdash \bar{t}_2 : det_\Theta$. Therefore, the reduction (A.36) implies with the closure Lemma C.2.2 $\bar{\Xi}_0, c_b : \mathsf{barb} \vdash C_0 \stackrel{\bar{t}_2}{\Longrightarrow}$, and further by composition (Lemma A.4.6)

$$C_2 \parallel C_0 \Longrightarrow C'',$$

where $C'' \equiv \nu(\Phi'' \setminus \Phi).C'_1 \between C'_2$. Since additionally, $C'' \downarrow_{c_b}$, the result follows. □

A.5 Completeness

For completeness, we start by proving a number of properties about the legal trace system, before proving definability.

A.5.1 Legal traces

For the representation of the legal traces, cf. Section 3.3.2. The lemmas of this section add up to show that the legal trace system is sound wrt. the operational semantics (cf. Lemma A.5.9). The term "soundness" as used here does not compare the observational preorder $\sqsubseteq_{obs}$ (or $\sqsubseteq_{may}$) and the trace-induced order $\sqsubseteq_{trace}$, but states that each interface behavior in the form of a trace produced by a *concrete* component is legal. In this sense, the legal trace system

[11] We could argue here, that $\Xi_0 \not\vdash t_1 : det_\Delta$ cannot be true, t_1 must be deterministic from the perspective of the observer, and furthermore we know $\Xi_0 \vdash t_1 : det_\Theta$. We don't need these facts for soundness, however.

[12] Note that transmission does not imply instantiation. Objects of type c_b are lazily instantiated *external* to $C_1 \parallel C_0$.

provides a sound approximation or abstraction of the interface behavior of a component. As stressed throughout, an important part is the sound overapproximation of the heap in the form of the connectivity contexts.

An important side result of this section is that components are input-enabled (Lemma A.5.5), i.e., a component accepts an incoming communication unconditionally (provided that according to the prior interface trace, the communication is possible).

The following lemma states a simple "invariant" about the form of the graphs encoded by E_Δ and E_Θ for legal traces. The lemma is the analog to Lemma A.1.3 for the operational semantics.

Lemma A.5.1 (Invariants)**.** *Derivations for legal traces, with $\Xi_0 \vdash \epsilon \rhd s : trace$ at the conclusion, preserve the following invariants for all subgoals $\Xi' \vdash r' \rhd s' : trace$:*

1. *$E'_\Delta \subseteq \Delta' \times (\Delta' + \Theta')$ and $E'_\Theta \subseteq \Theta' \times (\Theta' + \Delta')$.*
2. *$dom(\Delta') \cap dom(\Theta') = \emptyset$.*

Proof. By straightforward induction on the rules from Table 3.5, using the definitions of context update (Definition 2.6.8 and 2.6.9). □

Lemma A.5.2 (Soundness of balance)**.** *If $\Xi_0 \vdash C$ and $\Xi_0 \vdash C \stackrel{t}{\Longrightarrow}$, then $\vdash t : wbalanced$.*

Proof. By straightforward induction on the length of t, using the characterization of the number of calls and returns in a weakly balanced trace from Lemma A.2.17.

Let the natural numbers k_Θ and k_Δ be defined as in Lemma A.2.7. First it is clear from looking at the operational rules, that t is alternating, covering one requirement of Lemma A.2.17. Thus is suffices to show by induction that $k_\Theta \geq 0$ and $k_\Delta \geq 0$. As additional induction hypothesis we use: The number $k_\natural$ of stack-frames in $\natural$ equals k_Θ. To pick one concrete case, assume that $\Delta_0 \vdash \odot$; the argument for $\Theta_0 \vdash \odot$ is dual. For the base case, the claim holds trivially. In the induction case, we are given $\Xi_0 \vdash C_0 \stackrel{t'}{\Longrightarrow} \Xi \vdash C \stackrel{a}{\Longrightarrow}$, and we distinguish according to the nature of a.

Case: Incoming call: $t = t'\ \gamma_c?$
By induction, t' is weakly balanced, i.e., $\vdash t' : wbalanced^-$, and the length of t' is even (since t' is alternating and by $\Delta_0 \vdash \odot$). By Lemma A.2.7, $k_\Delta \geq 0$ and $k_\Theta \geq 0$. The incoming call increases k_Θ by one, preserving the invariant.

Case: Outgoing return: $t = t'\ \gamma_r!$
Rule RETO requires that $k_\natural > 0$. Hence, executing the return and thus decreasing k_Θ and $k_\natural$ both by 1 preserves the invariants.

Case: Outgoing call: $t = t'\ \gamma_c!$
Straightforward. Note that both $k_\natural$ and k_Θ remain unchanged.

Case: Incoming return: $t = t'\ \gamma_r?$
Since t' is weakly balanced we know stronger that $\vdash t' : wbalanced^-$, since t is alternating and (in the considered case of $\Delta \vdash \odot$) that the length of t' is even. Furthermore, to have rule RETI applicable, $k_\natural \geq 1$. By Lemma A.2.7(4), $k_\Theta = k_\Delta$, i.e., also $k_\Delta \geq 1$, using $k_\Theta = k_\natural$ as induction hypothesis. Hence, the incoming return, decreasing k_Δ by 1 and leaving k_Θ and $k_\natural$ unchanged, preserves the invariants.

The case for $\Theta_0 \vdash \odot$ is similar. The only critical case is the one for incoming returns. Now, the length of t' is odd instead of even. Hence, part 1b of Lemma A.2.7 applies, yielding that after t', $k_\Delta = k_\Theta + 1$, as the last label of t' must be outgoing. Therefore, also in this situation, the invariants hold after the step. □

To capture the possible reorderings of traces, the replays, etc., which all depend on the connectivity after executing a trace, we use Definition 3.1.1 analogously for legal traces:

Definition A.5.3 (Acquaintance after a trace)**.** *Assume a legal trace $\Xi_0 \vdash rs : trace$. We write $\Xi_0 \overset{r}{\Longrightarrow} \Xi$, if the derivation for $\Xi_0 \vdash r\ s : trace$ uses $\Xi \vdash r \rhd s : trace$ as intermediate goal. Furthermore we write $\Xi_0 \vdash r \rhd o_1 \leftrightharpoons o_2$ for $\Xi_0 \overset{r}{\Longrightarrow} \Xi$ and $\Xi \vdash o_1 \leftrightharpoons o_2$. The notation is used analogously for $\leftrightharpoons\hookrightarrow$.*

Note that in the definition of $\Xi_0 \overset{r}{\Longrightarrow} \Xi$, the post-configuration Ξ is *determined* by Ξ_0 and the trace r; there is only one legal trace derivation of $\Xi_0 \vdash rs : trace$. Furthermore, the "future" s does not influence Ξ. The next lemma formalizes the observation that the transformation of context by the external semantics coincides with the transformation by the rules of the legal traces. Note that we cannot prove the reverse direction of LemmaA.5.4(1) at this stage, where by reverse direction we mean: Given $\Xi_0 \overset{t}{\Longrightarrow} \Xi$, then there exits a component $\Xi_0 \vdash C_0$ such that $\Xi_0 \vdash C_0 \overset{t}{\Longrightarrow} \Xi \vdash C$. This property is a crucial ingredient for definability, i.e., also of completeness, and is shown later.

Lemma A.5.4.

1. *If $\Xi_0 \vdash C_0 \overset{t}{\Longrightarrow} \Xi \vdash C$, then $\Xi_0 \overset{t}{\Longrightarrow} \Xi$.*
2. *If $\Xi_0 \overset{t}{\Longrightarrow} \Xi_1$ and $\Xi_0 \vdash C \overset{t}{\Longrightarrow} \Xi_2 \vdash C$, then $\Xi_1 = \Xi_2$.*

In the concurrent setting, the lemma holds analogously for augmented traces.

Proof. By inspection of the rules from Table 2.11 and 3.5: Both sets of rules use the same premises to check and update the contexts.

□

Next we characterize the configuration of an input-enabled component. An enabled incoming return is a special case of that situation. Cf. Definition 3.3.3 respectively Definition 3.3.4 and equation (3.15) for the definition of $\Xi_0 \vdash t \rhd o_s \overset{a}{\rightarrow} o_r$ (enabledness of a as next interaction after trace t with sender o_s and receiver o_r). Note that we do not need (nor have) an analogous characterization for output-enabledness. One difference between a input enabled and an output enabled component, relevant in this context, is that if the component is input enabled, it is itself inactive and thus, the thread is at some definite point, as characterized by the lemma. An output enabled component is itself active, i.e., it is performing (if not deadlocked) internal actions before doing a next outgoing communication. Hence we cannot expect a characterization as precise as in the case of input-enabledness.

Lemma A.5.5 (Input enabled components)**.** *Assume $\Xi_0 \vdash C \overset{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$, starting from an initial configuration.*

1. *If* $\Xi_0 \vdash t \rhd o_r \overset{\gamma?}{\leftarrow} o_s$*, the component* $\acute{C}$ *is of one of the following three forms:*
   (a) *If* $t = \epsilon$*, then the thread* $\natural$ *is not contained in* $\acute{C}$*.*
   (b) *If* t *is not balanced, then* $\acute{C} = \nu(\Phi)(C' \parallel \natural\langle let\ x{:}T = o'_r\ blocks\ for\ o_s\ in\ t\rangle)$*.*
   (c) *If* $t \neq \epsilon$ *is balanced, then* $\acute{C} = C' \parallel \natural\langle stop\rangle$*.*
2. *If* γ *is a return, only part (1b) is possible.*

Proof. The property follows straightforwardly from the previous lemmas. Note that the three subcases of part 1 are mutually exclusive, as the empty trace is balanced. Note further that, since t is weakly balanced (Lemma A.5.2), $\Xi_0 \vdash t \rhd o_r \overset{\gamma?}{\leftarrow} o_s$ is well-defined (Definition 3.3.4 insists on a weakly balanced trace to assure that sender o_s and receiver o_r are defined). First it follows by straightforward induction on the length of the trace t that the number k_Θ (as defined in Lemma A.2.7) equals $k_\natural$, the number of stack-frames in $\natural$. Furthermore, by Lemma A.2.19, $t\ \gamma?$ is weakly balanced. Since by Lemma A.2.1, $t\ \gamma?$ is alternating, either t is empty, or the last label of t is outgoing. If $t = \epsilon$, $\Xi_0 \vdash \epsilon \rhd\colon o_r \overset{\gamma?}{\leftarrow} o_s$ implies $\Delta_0 \vdash \odot$ (and $o_s = \odot$), i.e., the threads starts in the environment and case 1a applies. Otherwise, as said, the last label of t must be outgoing. For outgoing calls, i.e., $t = t'\ \gamma'_c! = t'\ \nu(\Phi')\langle call\ o'_r.l(\vec{v})!$, the sender of $\gamma?$ equals the receiver of $\gamma'_c!$ (cf. Definition 3.3.4), i.e., $o'_r = o_s$. Note that in this case, t is not balanced. The rules from Table 2.11 directly give that the component is of the form as required by part 1b. If the last action of t was an outgoing return, i.e., $t = t'\ \gamma_r!$, we argue as follows. If t is balanced, $k_\Theta = k_\natural = 0$ (Lemma A.2.7(2)). Hence the thread is of the form $\natural\langle stop\rangle$, as required by part 1c. If t is not balanced, $k_\Theta = k_\natural > 0$ (Lemma A.2.7(3)), i.e., the call stack in $\natural$ is not empty and the thread is of the form as required by 1b. We need to check still, that the identity of o_s matches with the identity mentioned in the block-syntax, which follows from the definition of sender and receiver and the *pop*-function (Definition 3.3.4).

Remains the special case for returns, i.e., $\gamma? = \gamma_r?$. In this case, t is not balanced. Hence, with the same argument as in the corresponding situation above, the thread is of the form as required by part 1b. That the sender o_s matches the identity as mentioned in the code is again a consequence of Definition 3.3.4. □

The following lemma expresses that whether or not a component does an input step is determined *only* by the environment in the following sense: After having performed a trace t, an incoming communication step is possible, if the action is enabled after the history t, and checking input enabledness consults the environment contexts, only, plus the history of interaction (cf. the notations from equation (3.15) and from (2.17). In particular, the form of the thread *inside* in the component is such that the input step is possible (note that the different rules for input from Table 2.11 impose different restrictions on the form of the thread). This means that the restriction imposed on the possibility of taking an input step by the form of the component thread, which is an internal representation detail, is adequately represented by checking input enabledness after a trace at the interface (cf. Lemma A.5.5) plus the context checks in the premises of the rule.

Lemma A.5.6 (Input enabledness)**.** *Assume* $\Xi_0 \vdash C \stackrel{t}{\Longrightarrow} \Xi \vdash C$*. If* $\Xi_0 \vdash t\ \gamma? : trace$*, then* $\Xi \vdash C \stackrel{\gamma?}{\longrightarrow}$*.*

Proof. By Lemma A.5.5 on page 208, after trace t, thread $\natural$ of the component is of one of the forms (absent, blocked, or stopped) as stipulated by the respective cases of that lemma. In case, t is empty, case 1a applies, i.e., the thread is not contained in the component. Thus, $\Xi_0 \vdash C_0$ can do the input by CALLI_0, where the premise for context check is covered by the premise of L-CALLI_0. The premise $\Delta_0 \vdash \odot$ follows from the assumption that the thread is input-enabled after the empty trace (cf. Definition 3.3.3 for the definition of enabledness, using also Lemma A.2.13 on page 159 and the fact that the empty trace is balanced; note that for $t = \epsilon$, $o_s = \odot$).

If t is non-empty, only parts 1b or 1c of Lemma A.5.5 apply, depending on whether t is balanced or not. The assumption $\Xi_0 \vdash t\ \gamma? : trace$ implies with Lemma A.5.4(1) that $\Xi_0 \stackrel{t}{\Longrightarrow} \Xi \stackrel{\gamma?}{\Longrightarrow}$. By the premise of rule L-CALL_1, L-CALL_2, or L-RETI, the thread is input-enabled after t, i.e., $\Xi_0 \vdash r \rhd o_r \stackrel{a}{\leftarrow} o_s$.

Assume first that a is a call. If t is not balanced (but weakly balanced), the thread is blocked, with Lemma A.5.5(1b), waiting for o_s. This means, rule CALL_1 applies, allowing $\Xi \vdash C$ to perform the input. If otherwise t is balanced, the thread is stopped in C, according to part 1c of the lemma. By Lemma A.2.13, $pop\ t$ is undefined. Hence, the definition of call enabledness, especially equation (3.12), gives $\Delta_0 \vdash \odot$. Hence, CALLI_2 applies, allowing the required input step.

If, alternatively, the label a is a return, the thread can only be blocked, i.e., of the form as stated by part 1b of the lemma. This means, RETI applies, which concludes the case. □

The following easy lemma characterizes the change of enabledness when a trace is extended by a communication label.

Lemma A.5.7 (Enabledness: forward)**.** *Assume a legal trace* $\Xi \vdash r : trace$ *according to Table 3.5.*

1. *If* $\gamma_c?$ *is an input-call label and* r *is input-call enabled, then* $r\ \gamma_c?$ *is output-return enabled. Analogously, with sender and receiver mentioned: If* $\Xi \vdash r \rhd o_r \stackrel{\gamma_c?}{\leftarrow} o_s$*, then* $\Xi \vdash r\ \gamma_c? \rhd o_r \stackrel{\gamma_r!}{\rightarrow} o_s$*, for some return label* γ_r*.*
2. *If* γ_r *is a return label and* r *is input-return enabled, then* $r\ \gamma_r?$ *is output enabled. Analogously, with sender and receiver mentioned: If* $\Xi \vdash r \rhd o_r \stackrel{\gamma_r?}{\leftarrow} o_s$*, then* $\Xi \vdash r\ \gamma_r? \rhd o_r \stackrel{\gamma!}{\rightarrow} o'$*.*

The situation in both cases is dual for output labels.

Proof. Cf. the definition of the pop-operation and of enabledness (Definition 3.3.1 and Definition 3.3.3). Both parts of the lemma follow directly from the definition of enabledness. Especially, $pop\ (r\ a) = a$ (when a is a return label) is a direct consequence of the definition of pop and balance. □

Lemma A.5.8 (Legal trace: forward)**.** *Assume* $\Xi_0 \vdash t : trace$*. If*

1. $\Xi_0 \vdash t \rhd o_r \stackrel{\gamma?}{\leftarrow} o_s$ *and*

2. $\Xi_0 \stackrel{t}{\Longrightarrow} \Xi$ *and additionally* $\acute{\Xi} = \Xi + o_r \stackrel{\gamma?}{\leftarrow} o_s$ *with* $\acute{\Xi} \vdash o_r \stackrel{\lfloor a \rfloor}{\leftarrow} o_s : T$,

then $\Xi_0 \vdash t\ \gamma? : trace$. *If additionally* $\Xi_0 \vdash t : det_\Delta$ *and* $\Xi_0 \vdash t \rhd \gamma? : det_\Delta$, *then* $\Xi_0 \vdash t\ \gamma? : det_\Delta$. *The latter implication holds for* det_Θ, *as well. The lemma holds dually for* $\gamma!$.

Proof. Straightforward. □

The following lemma states that the type system for legal traces yields a sound over-approximation of the actual behavior of the transition relation.

Lemma A.5.9 (Soundness of legal traces)**.** *If* $\Xi_0 \vdash C$ *and* $\Xi_0 \vdash C \stackrel{t}{\Longrightarrow}$, *then* $\Xi_0 \vdash t : trace$.

Proof. By straightforward induction on the length of t, with additional help of subject reduction (cf. Lemma A.1.1) and Lemma A.5.5. □

Lemma A.5.10 (Trace duality)**.** *If* $\Xi_0 \vdash t : trace$, *then* $\bar{\Xi}_0 \vdash \bar{t} : trace$.

Proof. Obvious: Each rule of Table 3.5 has a dual counterpart. □

A.5.2 Definability

The next lemmas show some properties of the replay relation $\preccurlyeq_\Theta$ from Definition 3.1.8. The lemmas hold in dual form for $\preccurlyeq_\Delta$, as well. Clearly, $\preccurlyeq_\Theta$ is reflexive and transitive, and $\asymp_\Theta$ symmetric. Furthermore, by definition, we have the duality $\Xi_0 \vdash s \preccurlyeq_\Theta t$ iff $\bar{\Xi}_0 \vdash \bar{s} \preccurlyeq_\Delta \bar{t}$ (cf. Lemma A.2.56).

Lemma A.5.11 (Swapping)**.** *Assume two legal traces* $\Xi_0 \vdash t\ u\ v : trace$ *and* $\Xi_0 \vdash t\ v : trace$. *Assume further* $\Xi_0 \stackrel{tuv}{\Longrightarrow} \acute{\Xi}$, *and that* u *and* v *belong to different cliques. Then* $\Xi_0 \vdash t\ v\ u \asymp_\Theta t\ u\ v$.

Proof. For the reduction relation $\Xi_0 \stackrel{tuv}{\Longrightarrow} \acute{\Xi}$, cf. Definition A.5.3. For all component objects o from the clique of v, we have ${}_o\!\downarrow tuv = {}_o\!\downarrow tvu = {}_o\!\downarrow tv$. Analogously for all component objects o' from the clique of u, ${}_{o'}\!\downarrow tuv = {}_{o'}\!\downarrow tvu = {}_{o'}\!\downarrow tu$. Hence $\Xi_0 \vdash tuv \asymp_\Theta tvu$, as required. □

Note that the two trace segments being swapped occur at the *end* of the global trace. Indeed, the equality $t_1 v\ u\ t_2 \asymp_\Theta t_1\ u\ v\ t_2$ does in general *not hold* (for unaugmented traces).

Lemma A.5.12. *Assume two legal traces* $\Xi_0 \vdash t\ u\ v : trace$ *and* $\Xi_0 \vdash t\ v : trace$. *Assume further* $\Xi_0 \stackrel{tuv}{\Longrightarrow} \acute{\Xi}$, *and that* u *and* v *belong to different cliques. Then* $\Xi_0 \vdash t\ v \preccurlyeq_\Theta t\ u\ v$.

Proof. A straightforward consequence of the swapping Lemma A.5.11. □

Proof of Lemma 3.3.23 on page 75 (total correctness). We show $\Xi_0 \vdash C_t \stackrel{r}{\Longrightarrow} \Xi \vdash C$ for all prefixes $r \preccurlyeq t$. Let $t = r\ s$. As usual, $\Xi = \Delta; E_\Delta \vdash \Theta; E_\Theta$. The proof proceeds by induction on r, using the following induction hypotheses:

1. $\Xi \vdash C :: s$ (cf. Definition 3.3.22).

2. Depending on the enabledness condition after trace r, the thread in C_t is of the form as shown in Table A.6. This is meant as follows: if, after r, the thread is

   - input-return enabled, clause t_{ire} applies.
   - input-call enabled but not input-return enabled, clause t_{ie} applies.
   - output-return enabled, clause t_{ore} applies.
   - output-call enabled, but not output return-enabled, t_{oe} applies.

   The clause ϵ for t_{ie} denotes the absence of the thread in the component.

$$
\begin{array}{rcl}
t_{ie} & ::= & \epsilon \mid \mathit{stop} \mid t_{ire} \\
t_{ire} & ::= & t^i_{body}; t_{ie} \mid t^i_{blocked} \\
t^i_{body} & ::= & \mathit{let} \quad y{:}T' = o_r \ \mathit{blocks\ for}\ o_s \\
 & & \mathit{in} \quad (\ \mathit{let} \quad x{:}T = t^i_{sync}; t^o_{sync}\ \mathit{in}\ o_r\ \mathit{returns}\ x\ \mathit{to}\ o') \\
t^i_{blocked} & ::= & \mathit{let}\ y{:}T' = o_r\ \mathit{blocks\ for}\ o_s\ \mathit{in}\ \ t^i_{sync}; t^o_{sync} \\
t_{ore} & ::= & t^o_{body}; t_{ie} \\
t^o_{body} & ::= & \mathit{let}\ x{:}T = t^o_{sync}\ \mathit{in}\ o_s\ \mathit{returns}\ x\ \mathit{to}\ o_r \\
t_{oe} & ::= & t^o_{sync}
\end{array}
$$

Table A.6: Input and output enabled threads

Case: $r = \epsilon$
In this case $\Xi_0 \vdash C_t \stackrel{\epsilon}{\Longrightarrow} \Xi_0 \vdash C_t$, and part 1 is trivially satisfied, since there are no component objects (cf. equation (3.49)).

First it is clear that the component cannot be *return* enabled after the empty trace: The empty trace is balanced (cf. rule B-EMPTY$^+$ resp. B-EMPTY$^-$ of Table 3.3). Hence, $\mathit{pop}\ \epsilon = \bot$ (cf. Lemma A.2.13 and Definition 3.3.1 for pop). Hence the condition for return-enabledness does not apply (cf. Definition 3.3.3).

Assume then that the component is initially input enabled after ϵ (but not input-return enabled, as just argued). As just mentioned, $\mathit{pop}\ \epsilon = \bot$, and $\Delta_0 \vdash \odot$ (by equation (3.12) for input-call enabledness). Therefore, by construction, C_t does not contain the thread (Definition 3.3.20) and hence, condition of input-enabled threads in part 2 is met (with $t_{ie} = \epsilon$).

If otherwise the component is output-enabled (but not output-return enabled), we have analogously $\Theta_0 \vdash \odot$, which means according to equation (3.48), the initial thread code is of the form $\natural\langle t_0 \rangle = \natural\langle \mathit{let}\ x{:}c_i = \mathit{new}\ c_i\ \mathit{in}\ x.\mathit{start}() \rangle$ for some component class c_i. The initial configuration thus starts as follows (cf. the operational rules from Table 2.5, in particular NEWO$_i$ for instantiation and CALLI$_i$ for the internal call to start. For the definition of the start-method, see

Definition B.2.17.):[13]

$$\begin{array}{ll}
\Xi_0 \vdash C_0' \parallel c_i[\![F_i, M_i]\!] \parallel \natural\langle t_0\rangle & \Longrightarrow \\
\Xi_0 \vdash C_0' \parallel c_i[\![F_i, M_i]\!] \parallel \nu(o_\odot{:}c_i).(o_\odot[F_i, c_i] \parallel \natural\langle o_\odot.start()\rangle) & \Longrightarrow \\
\Xi_0 \vdash C_0' \parallel c_i[\![F_i, M_i]\!] \parallel \nu(o_\odot{:}c_i).(o_\odot[F_i, c_i] \parallel \natural\langle M_i.start(o_\odot)()\rangle) & \Longrightarrow \\
\Xi_0 \vdash C_0' \parallel c_i[\![F_i, M_i]\!] \parallel \nu(o_\odot{:}c_i).(o_\odot[F_i, c_i] \parallel \natural\langle t^o_{sync}()\rangle)\,. &
\end{array}$$

Hence, the thread is of the form as required by part 2.

Case: $r = r'a$

By induction, $\Xi_0 \vdash C_t \stackrel{r'}{\Longrightarrow} \Xi \vdash C$. We distinguish according to the nature of the next label a.

Subcase: Incoming call: $a = \nu(\Delta', \Theta').\langle call\ o_r.l(\vec{v})\rangle$?

As prefix of the legal trace t, also $r'a$ is legal. Hence, by the premise of one of the L-CALLI-rules (depending on the situation, only one of the two L-CALLI-rules apply) , r' is input-enabled, i.e., $\Xi \vdash r' \rhd a$. By induction, the thread is of the corresponding form t_{ie} from Table A.6 (in case that a is stronger input-return enabled, the thread is of the from t_{ire}, which is subsumed under t_{ie} in the grammar of Table A.6). For the reduction, we obtain:

$$\begin{array}{ll}
\Xi \vdash C & = \\
\Xi \vdash \natural\langle t_{ie}\rangle \parallel C' & \xrightarrow{a} \\
\acute{\Xi} \vdash \natural\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C' \parallel C(\Theta') & \xrightarrow{\tau} \\
\acute{\Xi} \vdash \natural\langle let\ x{:}T = M_r.l(o_r)(\vec{v})\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C' \parallel C(\Theta') & = \\
\acute{\Xi} \vdash \natural\langle let\ x{:}T = t_{body}[o_r/s][\vec{v}/\vec{x}]\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C' \parallel C(\Theta') & = \\
\acute{\Xi} \vdash \natural\langle let\ x{:}T = t^i_{sync}(\vec{v}); t^o_{sync}\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C' \parallel C(\Theta')\,. &
\end{array}$$

The reduction is again justified by the rules from Table 2.5 and 2.11, where t_{body} is the body of the invoked method labeled l. The external step a is is justified by CALLI_1 or CALLI_2, depending on whether the thread is input-return enabled after r' (CALLI_1) or not input-return enabled but only input-call enabled (CALLI_2). Remember: $C(\Theta')$ are the lazily instantiated objects created in the input step. The rule CALLI_0 does not apply. The τ-step is justified by the rule CALL_i for internal calls, where $C' = C'' \parallel c_r[\![F_r, M_r]\!]$ and where M_r in the step refers to the methods of the class of the receiver o_r. The premise $\Xi_0 \vdash r' \rhd o_r \stackrel{a}{\leftarrow} o_s : \vec{T} \rightarrow \_$ (determining sender and receiver plus the expected argument types) of the respective L-CALLI rule assures together with the premise $\acute{\Xi} \vdash o_r \stackrel{\lfloor a \rfloor}{\leftarrow} o_s : \vec{T} \rightarrow \_$ asserting well-typedness, that o_r is a component object (cf. equation (3.15) and Definition 3.3.4 for the definition of sender and receiver of a label and Definition 2.6.11 for well-typedness of labels, in particular LT-CALLI). See further equation (3.47) from Definition 3.3.20 for the definition of the method body of l, which is a public method.

The code $t^i_{sync}(\vec{x})$ for input synchronization is given in equation (B.3) in Definition B.2.2, where t^i_{sync} contains as free variables the formal parameter of the method it resides in, here $\vec{x}$. In the above reduction, the formal parameters $\vec{x}$ are replaced by $\vec{v}$, the actual parameters. Note that $t^i_{sync}(\vec{x})$ is the only part of t_{body} containing free occurrences of $\vec{x}$, in particular, t^o_{sync} does not contain $\vec{x}$, and hence the substitution $t^o_{sync}[\vec{v}/\vec{x}]$ is without effect.

[13] The object $o_\odot$ created initially is *not* identical to the symbol $\odot$ used in the contexts; of course it is meant to represent the initial clique in the code. Note, however, that $o_\odot$ will never be exported to the environment. Hence, it is also not important, which class/interface it possesses; any class of the component can be used.

The preconditions of the lemma for input synchronization apply at this point —the lazily instantiated component objects $C(\Theta')$ are yet undefined, the data structures of the pre-existing objects are yet unchanged; cf. the input synchronization Lemma B.3.1— hence, the reduction continues, executing $t^i_{sync}(\vec{v})$:

$$\begin{array}{ll}
\Xi \vdash \natural\langle let\ x{:}T = t^i_{sync}(\vec{v}); t^o_{sync}\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C' & \Longrightarrow \\
\acute{\Xi} \vdash \natural\langle let\ x{:}T = t^o_{sync}\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C'' & = \\
\acute{\Xi} \vdash \acute{C}\ , &
\end{array}$$

with $\acute{\Xi} \vdash \acute{C} :: s$, as required by part 1.

As for enabledness in part 2: As stated at the beginning of this subcase, the thread is input-enabled after r'. By Lemma A.5.7(1), $r'a$ is output-return enabled. Thus, at the end of the reduction, the thread at is of the required form (cf. the clause for t_{ore} for output-return enabledness).

Subcase: Incoming return: $a = \nu(\Delta', \Theta').\langle return(v)\rangle$?
By the analogous arguments as in the previous subcase, the thread is input-*return* enabled, i.e., $\Xi_0 \vdash r' \rhd a$. By induction, the thread is of the corresponding form t_{ire}; in particular the thread is blocked. In one of the cases, t_{ire} is of the form $t^i_{body}; t_{ie}$ (the alternative $t^i_{blocked}$ for t_{ire} works similarly). The reduction then looks as follows:

$$\begin{array}{ll}
\Xi \vdash C = \Xi \vdash C' \parallel \natural\langle t_{ire}\rangle & = \\
\Xi \vdash C' \parallel \natural\langle let\ y{:}T' = o_r\ blocks\ for\ o_s\ in\ (let\ x{:}T = t^i_{sync}(y); t^o_{sync}\ in\ o_r\ returns\ x\ to\ o_s); t_{ie}\rangle & \stackrel{a}{\longrightarrow} \\
\acute{\Xi} \vdash C' \parallel C(\Theta') \parallel \natural\langle (let\ x{:}T = t^i_{sync}[v/y]; t^o_{sync}\ in\ o_r\ returns\ x\ to\ o_s); t_{ie}\rangle & \Longrightarrow \\
\acute{\Xi} \vdash C'' \parallel C(\Theta') \parallel \natural\langle (let\ x{:}T = t^o_{sync}\ in\ o_r\ returns\ x\ to\ o_s); t_{ie}\rangle & = \\
\acute{\Xi} \vdash \acute{C}\ . &
\end{array}$$

The first, external step is justified by RETI from Table 2.11. Note that the let-bound variable y to receive the return value occurs free only in the code t^i_{sync} for input synchronization (cf. Definition B.2.2, where t^i_{sync} for returns, there written $t^i_{sync}(return, y)$ as meta-mathematical notation, mentions the return-variable y).

As in the previous subcase, $\acute{\Xi} \vdash \acute{C} :: s$ follows by Lemma B.3.1 for input synchronization. For enabledness: The thread in C is input-return enabled. Hence by Lemma A.5.7(2), the thread is output enabled after $r'a$, i.e., output-call enabled or stronger output-return enabled. Thus, $\acute{\Xi} \vdash \acute{C}$ is of the required form t_{ore}. In the mentioned alternative case where $t_{ire} = t^i_{blocked}$, the reduction yields t_{oe}, which conforms to the requirements from Table A.6, as well.

Subcase: Outgoing call: $a = \nu(\Delta', \Theta').\langle call\ o_r.l(\vec{v})\rangle$!
By analogous arguments as in the previous subcases, the thread is output enabled or stronger output-return enabled after r', and thus of the form t_{oe} or t_{ore}. In either case, the code starts with output synchronization, where the code for t^o_{sync} is given in equation (B.11) in Definition B.2.11. In case of output-return enabledness, the reduction sequence looks as follows:

$$\begin{array}{ll}
\Xi \vdash C & = \\
\Xi \vdash C' \parallel \natural\langle t_{ore}\rangle & = \\
\Xi \vdash C' \parallel \natural\langle t^o_{body}; t_{ie}\rangle & = \\
\Xi \vdash C' \parallel \natural\langle let\ x{:}T = t^o_{sync}\ in\ o_s\ returns\ x\ to\ o'; t_{ie}\rangle & \Longrightarrow \\
\Xi \vdash C'' \parallel \natural\langle let\ y{:}T' = o_r.l(\vec{v})\ in\ (let\ x = t^i_{sync}(y); t^o_{sync})\ in\ o_s\ returns\ x\ to\ o'; t_{ie}\rangle & \stackrel{a}{\longrightarrow} \\
\acute{\Xi} \vdash C'' \parallel \natural\langle let\ y{:}T' = o_s\ blocks\ for\ o_r\ in\ (let\ x{:}T = t^i_{sync}(y); t^o_{sync})\ in\ o_s\ returns\ x\ to\ o'; t_{ie}\rangle\ . &
\end{array}$$

The reduction sequence, executing t^o_{sync}, and part 1 of the lemma follows by Lemma B.3.2 for output synchronization. Furthermore, the thread code of in the post-configuration complies to requirements of part 2, being input-return enabled.

Subcase: Outgoing return: $a = \nu(\Theta', \Delta').\langle return(v)\rangle$!
The case for outgoing return works similarly, using again the Lemma B.3.2 for output synchronization. □

The next lemma says, the the known objects and names are stored appropriately in the scripts data structure. Remember also Notation 3.3.18.

Lemma A.5.13. *Let t be a legal trace and $\Xi_0 \vdash C_t$ given by Definition 3.3.20. Assume further $\Xi_0 \vdash C_t \stackrel{s}{\Longrightarrow} \Xi \vdash C$, and the following two sets of object identities: $O_1 = \{o' \mid \Xi \vdash_\Theta o \leftrightharpoons o'\}$ and $O_2 = \{o' \mid \Xi_0 \vdash_\Theta o \leftrightharpoons;\hookrightarrow o'\}$. Note that O_1 contains only component objects and denotes the clique of o, i.e., $O_1 = [o]_{/\Xi}$ (or $[o]$ for short), and O_2 contains only environment objects. Let furthermore $(\sigma, \check{s})$ an arbitrary script from $[o].script$. Then $O_1 = ran_\Theta(\sigma)$ and $O_2 = ran_\Delta(\sigma)$.*

Proof. By induction on the length of trace s. For the base case $s = \epsilon$, the property holds vacuously: O_1 and O_2 are empty, and furthermore, there does not exist any component object yet. The induction step for output steps is covered by the code of $step^o$ from Definition B.2.10, for input steps by the code of $step^i$ from Definition B.2.8. □

Proof of Lemma 3.3.24 on page 75 (Exactness/partial correctness). So assume $\Xi_0 \vdash C_t \stackrel{r}{\Longrightarrow}$. We need to show that $\Xi_0 \vdash r \preccurlyeq_\Theta t$.

According to Definition 3.1.8 of $\preccurlyeq_\Theta$, we need to show that for all component cliques $[o]_{/\Xi}$ (or $[o]$ for short) after r, there exists a renaming t' of t such that ${}_{[o]}\!\downarrow r \preccurlyeq {}_{[o]}\!\downarrow t'$, i.e., that for all component object names $o \in [o]_{/\Xi}$, ${}_o\!\downarrow r \preccurlyeq {}_o\!\downarrow t'$. Note that r is legal, i.e., $\Xi_0 \vdash r : trace$, using soundness of legal traces from Lemma A.5.9. I.e., r and t are legal in the same context Ξ_0. In the following, we abbreviate $[o]_{/\Xi}$ by $[o]$, analogously for $[o']$, ...

Assume $\Xi_0 \vdash C_t \stackrel{r}{\Longrightarrow} \Xi \vdash C$. The invariant of Lemma B.4.12, equation (B.43) gives for the reduction of C_t, that for all component cliques $[o']$ according to Ξ, for all scripts $(\sigma, \check{s})$ from $[o'].scripts$ and for all component objects $o \in ran_\Theta(\sigma)$:

$$\check{r}_x\sigma = {}_o\!\downarrow r \quad \text{and} \quad \check{s} = \check{s}_x \quad \text{and} \quad \check{t}_x = \check{r}_x\check{s}_x \quad \text{where } x = \sigma^{-1}(o)\ . \qquad \text{(A.37)}$$

The $\check{t}_x$ corresponds to the projection of $\check{t}$ to the role x, i.e., the "static" variant of the projection ${}_o\!\downarrow t$ of t to a component object o, when x is the role for o (cf. Definition 3.1.3 for the definition of projection). The $\check{r}_x$ is a prefix of $\check{t}_x$, and $\check{s}_x$ the remaining postfix. For the abbreviations $\check{t}_x$, $\check{r}_x$, and $\check{s}_x$, see also the mentioned Lemma B.4.12. Informally, (A.37) means that the *actual* past ${}_o\!\downarrow r$ of an object o corresponds to the some "static" past $\check{r}_x$ of $\check{t}_x$, where o is interpreted to play the role x, stipulated by $x = \sigma^{-1}(o)$. Furthermore, the still open future in $scripts$ corresponds to the rest of $\check{t}_x$.

Independent from the informal interpretation: For the (arbitrary) clique $[o']$ we have $[o'] = ran_\Theta(\sigma)$ (Lemma A.5.13). I.e., for all objects o from the component clique $[o']$, the above equation (A.37) holds. Now, $\check{r}_x\sigma = {}_o\!\downarrow r$ and $\check{r}_x \preccurlyeq \check{t}_x$ implies that for all objects o from $[o']$, that ${}_o\!\downarrow r \preccurlyeq t$, as required. □

A.5.3 Completeness argument

Proof of Lemma 3.3.26 on page 76. See Definition 3.3.25 for the relations $\preccurlyeq^{\bullet}_{\Delta}$, resp., $\preccurlyeq^{\bullet}_{\Theta}$, and $\sqsubseteq^{nondet}_{trace}$. Assume $\Xi_0 \vdash C_1 \stackrel{t_1}{\Longrightarrow}$, and distinguish according to the form of trace t_1.

Case: $t_1 = \epsilon$

Choosing $t_2 = \epsilon$ gives immediately $\Xi_0 \vdash C_2 \stackrel{t_2}{\Longrightarrow}$, as required, and furthermore, the two parts of Definition 3.3.25 for $\preccurlyeq^{\bullet}_{\Delta}$ are satisfied, relating t_1 and t_2.

Assume then a non-empty trace t_1 with $\Xi_0 \vdash C_1 \stackrel{t_1}{\Longrightarrow}$, and let $[o_1]$ be an arbitrary observer clique after t_1. Note that the replay relation is considered from the perspective of the environment: The observer cannot distinguish certain orders or whether one behavior is done once or more than once.

Case: $t_1 = r_1\ \gamma!$

We start with part 2 and dealing with the case where the last interaction of the clique $[o_1]$ is an output (from the perspective of C_1. So for the observer, it is an input). Consider the dual trace $\bar{t}_1$, i.e., the trace from the perspective of the receiver and observer. As t_1 is legal (using soundness of the legal trace system from Lemma A.5.9), the complement is legal, too (by trace duality from Lemma A.5.10), i.e.,

$$\bar{\Xi}_0 \vdash \bar{r}_1\gamma? : \mathit{trace}\,.$$

It is easy to see —there are no arguments to the $succ$-call and hence there is no connectivity information involved; furthermore, extending a weakly balanced trace by the *call* does not break the balance conditions— that also the trace extended by one outgoing success-reporting action is legal, i.e.,

$$\bar{\Xi}'_0 \vdash \bar{r}_1\ \gamma?\ \mathit{succ}! : \mathit{trace},$$

where $succ$ abbreviates $(\nu b{:}c_b).\langle \mathit{call}\ b.\mathit{succ}()\rangle!$, and where the context $\bar{\Xi}'_0$ is given by extending the environment $\bar{\Delta}_0$ to $\bar{\Delta}_0, c_b{:}\mathsf{barb}$. Note that the sender clique of the call $succ!$ is the receiver of $\gamma?$ (Lemma A.2.14(2)).

Consider the component $\bar{\Xi}'_0 \vdash C_{\bar{t}_1\,succ!}$ and let us abbreviate the observer $C_{\bar{t}_1\,succ!}$ as C_O, and furthermore let Ξ_b stand for the context $c_b{:}\mathsf{barb}$. Since initially, C_1 and C_O are static, $C_1 \between C_O = C_1 \parallel C_O$. By total correctness of C_O (Lemma 5.2.8) and composition (Lemma A.4.6), $\Xi_b \vdash C_1 \parallel C_O \Longrightarrow \Xi_b \vdash \acute{C}_{1,O} \downarrow_{c_b}$, or more explicitly:

$$\Xi_b \vdash C_1 \parallel C_O \stackrel{t_1}{\underset{\bar{t}_1}{\Longrightarrow}} \Xi_b \vdash \acute{C}_{1,O} \downarrow_{c_b}\ ,$$

where the internal reduction $\Longrightarrow$ is decorated by the two complementary traces and where furthermore $\acute{C}_{1,O} = \nu(\acute{\Phi} \setminus \Phi).\acute{C}_1 \between \acute{C}_O$ (= $\nu(\acute{\Phi}).\acute{C}_1 \between \acute{C}_O$ since Φ contains no bindings for object names). As $\Xi_0 \vdash C_1 \sqsubseteq_{may} C_2$, we can replace C_1 by C_2 and still observe success (Definition 2.5.1), i.e., $\Xi_b \vdash C_2 \parallel C_O \Longrightarrow\downarrow_{c_b}$. By trace decomposition (Lemma A.4.12),

$$\Xi_b \vdash C_2 \parallel C_O \stackrel{t_2}{\underset{\bar{t}_2}{\Longrightarrow}} \Xi_b \vdash \acute{C}_{2,O} \downarrow_{c_b} \tag{A.38}$$

for some trace t_2, more precisely:

$$\Xi_b, \Xi_0 \vdash C_2 \stackrel{t_2}{\Longrightarrow} \Xi_b, \acute{\Xi}_2 \vdash \acute{C}_2 \quad \text{and} \quad \Xi_b, \bar{\Xi}_0 \vdash C_O \stackrel{\bar{t}_2}{\Longrightarrow} \Xi_b, \bar{\acute{\Xi}}_2 \vdash \acute{C}_O\ , \tag{A.39}$$

with $C_{2,O} = \nu(\acute{\Phi}_2)\acute{C}_2 \between \acute{C}_O$. For the observer, this means

$$\Xi_b, \bar{\Xi}_0 \vdash C_O \stackrel{\bar{t}_2 succ'!}{\Longrightarrow} \tag{A.40}$$

Note that $succ'!$ may be an α-variant of $succ!$. By partial correctness from Lemma 3.3.24,

$$\Xi_b, \bar{\Xi}_0 \vdash \bar{t}_2 succ'! \preccurlyeq_\Theta \bar{t}_1 succ! \tag{A.41}$$

Since $succ$, resp., $succ'$ is unique, i.e., no α-variant occurs in $\bar{t}_2$ or in $\bar{t}_1$, by the shortening Lemma C.2.1

$$\Xi_b, \bar{\Xi}_0 \vdash \bar{t}_2 \preccurlyeq_\Theta \bar{t}_1 \ . \tag{A.42}$$

Without the trailing label $succ$, we can strengthen that statement to

$$\bar{\Xi}_0 \vdash \bar{t}_2 \preccurlyeq_\Theta \bar{t}_1. \tag{A.43}$$

By Lemma A.2.56, this is equivalent to the dual judgment $\Xi_0 \vdash t_2 \preccurlyeq_\Delta t_1$, covering part 2 from Definition 3.3.25.

For part 1, we argue as follows. Still, $[o_1]$ is the clique of the last action of t_1, i.e., a clique of the observer, which is also the sender clique of $succ!$ after $\bar{t}_1$. Equation (A.41) from above means by Definition 3.1.8 of $\preccurlyeq_\Theta$, that for all component cliques[14] $[o_2']_{/\bar{\Xi}_2'}$ after $\bar{t}_2 succ'!$, there exists an α-renaming $\bar{v}_1 succ'!$ of $\bar{t}_1 succ!$ such that

$$\Xi_b, \bar{\Xi}_0 \vdash {}_{o'}\!\downarrow \bar{t}_2 \ succ'! \preccurlyeq {}_{o'}\!\downarrow \bar{v}_1 \ succ'! \ , \tag{A.44}$$

for all objects o' from $[o_2']_{/\bar{\Xi}_2'}$ (after $\bar{t}_2 \ succ'!$). Considering specifically the success-reporting clique $[o_1]$, we have ${}_{o}\!\downarrow \bar{t}_2 \ succ'! \preccurlyeq {}_{o}\!\downarrow \bar{s}_1 \ succ'!$ for some renaming $\bar{s}_1 \ succ'!$ of $\bar{t}_1 \ succ'!$, and for all objects of that clique. Since the label $succ'$ is *unique*, $\Xi_b, \bar{\Xi}_0 \vdash {}_{o}\!\downarrow \bar{t}_2 = {}_{o}\!\downarrow \bar{s}_1$ for all o of $[o_1]$, which can be strengthened to $\bar{\Xi}_0 \vdash {}_{o}\!\downarrow \bar{t}_2 = {}_{o}\!\downarrow \bar{s}_1$ since the type/class c_b is neither mentioned in $\bar{s}_1$ nor in $\bar{t}_2$. Dualizing the projection (Lemma A.2.47) gives

$${}_{o}\!\downarrow t_2 = {}_{o}\!\downarrow s_1 \ , \tag{A.45}$$

from which the result follows.

Case: $t_1 = r_1 \ \gamma?$

This case follows from the previous one for a trace ending with an output by the following argument, basically exploiting the fact that a component is input enabled and cannot *refuse* to take an input, where the crucial preservation property is provided by Lemma A.2.49.

By soundness of the legal trace system from Lemma A.5.9, the behavior t_1 of C_1 is legal, i.e., $\Xi_0 \vdash t_1 : trace$, in particular, t_1 is weakly balanced (Lemma A.2.21). Hence by Lemma A.2.14(2), $sender(r_1 \ \gamma?) = receiver(r_1)$, if r_1 is non-empty, i.e., the sender environment clique of label $\gamma?$ is the receiver of the last (outgoing) label in r_1; in case, r_1 is empty, i.e., where $receiver(r_1)$ is undefined, $sender(r_1 \ \gamma?) = sender(\gamma?)$ is $\odot$ (by the same Lemma A.2.14(2)). Clearly, $\Xi_0 \vdash C_1 \stackrel{r_1}{\Longrightarrow}$, and the shorter r_1 is legal, as well. Hence, by the previous subcase,

$$\Xi_0 \vdash C_2 \stackrel{u_1}{\Longrightarrow} \quad \text{and} \quad \Xi_0 \vdash u_1 \preccurlyeq^\bullet_\Delta r_1 \ , \tag{A.46}$$

[14] Component cliques from the dual perspective of $\Xi_b, \bar{\Xi}_0$, i.e., the cliques of the observer C_O.

for some trace u_1 (cf. condition 1 and 2 from Definition 3.3.25). By once again soundness of the legal trace system (Lemma A.5.9), also $\Xi_0 \vdash u_1 : \mathit{trace}$. The fact that $r_1\ \gamma?$ is legal yields with Lemma A.2.35 that

$$\Xi_0 \vdash r_1 \rhd \gamma? : \mathit{ok}\ . \tag{A.47}$$

Combining this with the right-hand side of (A.46) gives with Lemma A.2.49

$$\Xi_0 \vdash u_1 \rhd \gamma? : \mathit{ok}\ . \tag{A.48}$$

By Lemma A.5.8, $\Xi_0 \vdash u_1\ \gamma? : \mathit{trace}$, and finally, by using input enabledness from Lemma A.5.6 and setting $t_2 = u_1\ \gamma?$, we get $\Xi_0 \vdash C_2 \xrightarrow{\gamma?}$, i.e., $\Xi_0 \vdash C_2 \stackrel{t_2}{\Longrightarrow}$, as required. □

Proof of Lemma 3.3.28 on page 77 (individual determinism). We show by induction on the length of u_1, that $\Xi_0 \vdash u_2 \asymp_\Delta u_1$ for some trace u_2.

Case: Base case: $u_1 = \epsilon$
Immediately, for $u_2 = \epsilon$ and reflexivity: $\Xi_0 \vdash \epsilon \asymp_\Delta \epsilon$.

Case: Induction case: $u_1 = u_1'\ a$
We distinguish according to the nature of action a.

Subcase: $u_1 = u_1'\ \gamma?$ (input)
This case is covered by input enabledness: Extending the trace from the induction hypothesis by an input gives $\Xi_0 \vdash C_2 \stackrel{u_2'\gamma?}{\Longrightarrow}$. Furthermore we are given $\Xi_0 \vdash u_2' \asymp_\Delta u_1'$ and assume wlog. that the sender clique of $\gamma?$, an environment clique, is not affected by the renaming possible when using the relation $\asymp_\Delta$. Thus $\Xi_0 \vdash u_1'\ \gamma? \asymp_\Delta u_2'\ \gamma?$, as required. Note that (from the perspective of Δ) $u_2'\ \gamma?$ is legal (in particular deterministic) since $u_1'\ \gamma?$ is, and since $\Xi_0 \vdash u_2' \asymp_\Delta u_1'$.

Subcase: $u_1 = u_1'\ \gamma!$ (output)
Let $[o]$ be the receiver environment clique of $\gamma!$. There are two cases to distinguish. If $\gamma!$ is a replay-action from the perspective of the observer, i.e., if $\Xi_0 \vdash u_1'\ \gamma! \asymp_\Delta u_1'$, the case is immediate by induction and by transitivity of $\asymp_\Delta$.

So assume $\Xi_0 \vdash u_1'\ \gamma! \not\asymp_\Delta u_1'$. This is the only case where something interesting happens, namely when we need the observer to enforce progress. Using induction on the shorter u_1', we get

$$\Xi_0 \vdash v_2'' \asymp_\Delta u_1'\ . \tag{A.49}$$

for some trace v_2''. Rename v_2'' to v_2', such that the names from from the clique $[o]$ after u_1 are *identical* with the names after v_2', i.e., $\Xi_0 \vdash v_2' \downarrow_{[o]} \asymp_\Delta u_1' \downarrow_{[o]}$ and $v_2' =_\alpha v_2''$, where $[o]$ is the receiver clique of $\gamma!$. Since $\asymp_\Delta$ is closed under renaming, we have also

$$\Xi_0 \vdash v_2' \asymp_\Delta u_1'\ . \tag{A.50}$$

Note that the projection $u_1' \downarrow_{[o]}$ might be empty, namely in the situation, were the environment clique $[o]$ projected onto is created in the last step of $u_1'\ \gamma!$.

Additionally, by assumption, there exists a legal trace and thus a Δ-deterministic trace u_2 such that the two conditions of Definition 3.3.25 for $\Xi_0 \vdash u_2 \preccurlyeq^\bullet_\ominus u_1'\ \gamma!$ hold, i.e.,

1. ${}_o\!\downarrow u_2 = {}_{o'}\!\downarrow u_1' \, \gamma!$ for all Δ-objects $o' \in [o]$, where $[o]$ is the receiver clique of $\gamma!$, and

2. $\Xi_0 \vdash u_2 \preccurlyeq_\Delta u_1' \, \gamma!$ and

Using this information, the induction hypothesis, and the assumption of determinism, the goal now is

$$\Xi_0 \vdash u_2 \succcurlyeq_\Delta u_1' \, \gamma! \, ,$$

since, together with condition 2, this gives $\Xi_0 \vdash u_2 \asymp_\Delta u_1' \, \gamma!$, as required.

Condition 1 gives that u_2 is of the form $u_2' \, \gamma!$, since we project on the clique of the receiver of label $\gamma!$. Note that u_2 indeed *ends* with $\gamma!$, since it is an outgoing communication and we project onto the environment clique of its receiver.

Furthermore, the assumption that both the trace v_2' from the induction hypothesis and trace $u_2' \, \gamma!$ are in the set $T = \{u' \mid u' \preccurlyeq_\alpha u \quad \text{or} \quad u' \succcurlyeq_\alpha u\}$ of traces (for some u) implies that v_2' is a (not necessarily proper) prefix of $u_2' \, \gamma!$, or vice versa. Thus we distinguish:

Subsubcase: $u_2' \, \gamma! \succ v_2'$, i.e., $u_2' \, \gamma! = v_2' \, w \, \gamma!$ for some $w \succcurlyeq \epsilon$
We first argue that condition 1 together with the fact that we use v_2' from equation (A.50) as an appropriate renaming of v_2'' from equation (A.49) from above gives, that w does not concern the clique $[o]$. Assume for a contradiction, that a non-empty subsequence w_2 of $w = w_1 \, w_2 \, w_3$ concerns the component clique $[o]$. This implies that the non-empty w_2 occurs (without renaming) both in $v_2' \, w \, \gamma!$ (as part of w) and in v_2', both times interacting with $[o]$, which is impossible.

By (the dual variant of) Lemma A.5.12,

$$\Xi_0 \vdash u_2' \, \gamma! \succcurlyeq_\Delta v_2' \, \gamma! \, . \tag{A.51}$$

Furthermore we know that $\mathit{receiver}(v_2' \, \gamma!) = \mathit{receiver}(u_1' \, \gamma!)(= [o])$. So from the induction hypothesis $\Xi_0 \vdash v_2' \asymp_\Delta u_1'$ in equation (A.50), we get

$$\Xi_0 \vdash v_2' \, \gamma! \asymp_\Delta u_1' \gamma! \, , \tag{A.52}$$

and hence by transitivity of the $\preccurlyeq_\Delta$-relation, $\Xi_0 \vdash u_2' \, \gamma! \succcurlyeq_\Delta u_1' \, \gamma!$, as required.

Subsubcase: $u_2' \, \gamma! \prec v_2'$, i.e., $v_2' = u_2' \, \gamma! \, w$, for some w.
Again, by condition 1, w does not concern the clique $[o]$, which together with the induction hypothesis implies that already $\Xi_0 \vdash v_2' \succcurlyeq_\Delta u_2' \, \gamma!$. □

Proof of completeness (Theorem 3.3.29 on page 77). Assume $\Xi_0 \vdash C_1 \stackrel{t_1}{\Longrightarrow}$. Since the set of traces of C_1 is prefix closed, clearly $\Xi_0 \vdash C_1 \stackrel{u_1}{\Longrightarrow}$ for all prefixes $u_1 \preccurlyeq t_1$. Lemma 3.3.26 therefore gives that for all $u_1 \preccurlyeq t_1$, there exists a u_2 with $\Xi_0 \vdash C_2 \stackrel{u_2}{\Longrightarrow}$ and $\Xi_0 \vdash u_2 \preccurlyeq^\bullet u_1$.

We have not yet used the fact that closed programs are deterministic. It is easy to see that the series of u_2 are all contained in the set $T = \{u_2' \mid u_2' \preccurlyeq_\alpha u \quad \text{or} \quad u_2' \succcurlyeq_\alpha u\}$ for some u. Hence by 3.3.28 $\Xi_0 \vdash u_2 \asymp_\Delta t_2$, as required by Definition 3.1.11 of $\sqsubseteq_{trace}$. □

APPENDIX **B**

Coding

This chapter is concerned with the realization of the observer. Thus it contains the missing pieces of the code and the corresponding properties used in the proofs for completeness, more precisely, in the construction and the corresponding proofs for Propositions 3.3.23 and 3.3.24 in the sequential case and Corollary 5.2.11 in the multithreaded case. The core of the construction and the corresponding proofs are identical, and thus most of the definitions and properties apply to both the sequential and the concurrent setting. We start in Section B.1 with an overview.

B.1 Overview

The construction of C_t from a legal trace contained, what we called, synchronization code. We start with an abstract description of what it is good for.

The pieces of synchronization code in the construction of the component C_t from Definition 3.3.20 (resp. Definition 5.2.7) come in two flavors, *input* and *output* synchronization code, and flank the corresponding external transition steps at the interface. Output synchronization code *precedes* the corresponding output, and dually, input synchronization *trails* the input action.

The *commitment* contexts of the judgments $\Xi \vdash C$ are nothing else than an interface specification of the component wrt. the existence of objects (and threads) plus their connectivity. Thus the implementation requirements can be understood by looking at the change of the $\Xi \vdash C$- judgments in external steps (cf. Tables 2.11 and 3.5, resp., Tables 4.8 and 5.1 for the concurrent case.) The changes are always additive, i.e., the contexts only grow larger. To implement the extension of the typing context Θ in an output step, the component must *create* corresponding objects, whose references are then published. Likewise the component must cater for lazily instantiated objects of the environment, which lead to an extension of E_Δ in an output step, and in the multithreaded setting for new threads exported to the outside by an outgoing call. On the other hand, the component is not responsible for extensions of Θ by incoming lazy instantiation.

As a manner of speaking, the commitment context $\Sigma, \Theta; E_\Theta$ for a judgment $\Xi \vdash C$ specifies the *static* (in the sense of "current") requirements to be implemented in C, whereas the (remaining part of) the given legal trace specifies the dynamic or *behavioral* part of the coding requirement (cf. Definition 3.3.22). The programming task for $\Xi_0 \vdash C_t$ amounts to implement an *interpreter* that works off the given trace t step by step.

For connectivity as specified by E_Θ, we adopt a "distributed" implementation, where the information must be distributed or broadcast to all members of the clique, when the connectivity context E_Θ is enlarged.

We split the synchronization task into the following sub-problems:

1. *create* new objects to be made known or exported to the outside (cf. Definition B.2.13),

2. *broadcast* connectivity information to keep the component fully connected and in sync wrt. the future behavior (cf. Definition B.4.8), and

3. *serialize* the component's actions to exhibit *exactly* the behavior prescribed by the trace, at least up to the closure conditions on the set of traces.[1]

4. In the multithreaded setting, provide mutual exclusion to avoid concurrent access to the common data structures, and furthermore,

5. *new threads* are spawned, before their name is exported to the environment.

[1] Of course, the component cannot completely enforce the given behavior, for various reasons. Especially, separate cliques cannot enforce a particular order of their respective events.

The first point is straightforward: The synchronization code for output contains appropriate new-statements. The second one will be done by traversing the clique, updating the connectivity knowledge of all of its members.

The serialization task mentioned in point 3 is not implied by the previous discussion about how the commitment contexts and their change specify the implementation task. It is mandated by the completeness proof in general. Anyway, the task is to ensure that the actions and reactions of the component follow the prescribed order.

For instance, consider (in the multithreaded setting) that the trace contains the following sequence of two actions $\gamma?\ \gamma'!$

$$n_1\langle call\ o_1.l_1()\rangle?\ n_2\langle call\ o_2.l_2()\rangle!\ , \tag{B.1}$$

where $n_1 \neq n_2$. In this situation, the implementation must *enforce* the given order, i.e., it is necessary to assure that thread n_2 does not issue the second call *before* the first incoming call has been accepted.[2] Note that if the component had to realize the opposite order

$$n_2\langle call\ o_2.l_2()\rangle!\ n_1\langle call\ o_1.l_1()\rangle?\ , \tag{B.2}$$

this order cannot be enforced, the order of equation (B.1) is unavoidable, as well. Cf. also the switching rules, in particular rule O-OI, for $\sqsubseteq_\Theta$ of Table 5.2.

To achieve "serialization", each object of the clique must be aware and kept up-to date of the current status wrt. the sequence of interactions at the clique's interface. In the situation of equation (B.1), for instance, the caller object of the second, outgoing call, must be aware whether or not the first call $\gamma?$ has already occurred.

In the concretely constructed component, the objects do not keep a *history* of past interaction. Rather the current state is characterized by the *future interaction* the component still has to realize. We call such a linear description of the future of an object (plus an abstraction of the already witnessed past) a *script* (cf. the interface type in Definition B.2.1 and also the informal discussion in Section 3.3.3 and equation (3.45)).

Whereas the scripts are kept in instance variables of each object, conceptually they describe the future of the whole clique. The values of the scripts for each object will be kept in sync, i.e., we maintain the *invariant* that all members of a clique agree upon their potential futures. Note further that whereas traces of a component can be thought of as are tree-structured, the futures are *linear*; the trees branch into the past, since cliques only merge, but never split. The constructed component equips each object with the possible future behavior. Since the instances of the class may have to behave differently according to the given legal trace, the class contains a *set* of possible linear futures.

Concretely, the future behavior is implemented by an instance variable $script$ containing one sequence of *actions*, while the class collects all possible futures in the $scripts$ (plural) instance variable. We refer to $c.scripts$ to the (static) code of $scripts$ in c (cf. also Definition B.2.1). We maintain as invariant that all objects of a clique agree on the common future.

[2] Assuming, the two actions concern the same clique, otherwise the order cannot be enforced.

To avoid data corruption due to concurrent access, all the described bookkeeping is done under mutual exclusion, at least per clique.[3] We use the syntax $(\!|t|\!)$ to indicate that the code t is executed without interference from other threads. Intuitively, the opening parenthesis $(\!|$ takes a lock (if available) which ensures undisturbed access to the whole clique.[4] The dual $|\!)$ releases the lock again.

B.2 Abstract sync code

This section describes at an abstract level the "synchronization code" which has been used in the proofs of definability (partial and total correctness). The code works with the data structures mentioned above and illustrated in the examples of Section 3.3.3.

Definition 3.3.16 sketched the interface of each component class of the constructed C_t, concentrating on the two main methods taking care of external output steps and of external input steps and the core scripts data structure. Those two methods $step^o$ and $step^i$ are accompanied by a number of auxiliary method definitions, dealt with in the following. Definition B.2.1 shows them in overview, i.e., presenting a more detailed view on Definition 3.3.16. Apart from the fact that the "type" of labels and thus scripts are a bit more complex (containing additionally the thread name, for instance) and the methods $(\!|$ and $|\!)$, Definition B.2.1 is identical for the concurrent and the deterministic setting.

Definition B.2.1 (Data structures (2))**.** *Each class contains fields $init$ and $scripts$ containing the future. In overview and ignoring "overloading", the interface type for each component class is of the form:*

$$
\begin{array}{ll}
[\!(& scripts, init : \mathsf{set\ of\ script} \\
& \Theta : \mathsf{set\ of\ object} \\
& step^i : \mathsf{label} \times (\mathsf{set\ of\ object}) \to \mathsf{Unit} \\
& step^o : \mathsf{Unit} \to \mathsf{Unit} \\
& initialize : \mathsf{label} \to \mathsf{Unit}, \mathsf{Unit} \to \mathsf{Unit} \\
& create : \mathsf{label} \to \mathsf{assoc} \\
& pickrepresentative : \mathsf{label} \to \mathsf{set\ of\ object} \\
& collectroles : \mathsf{assoc} \times (\mathsf{set\ of\ object}) \to \mathsf{set\ of\ assoc} \\
& broadcast : \mathsf{scripts} \to \mathsf{Unit} \\
& interpret : \mathsf{label} \to \mathsf{Unit} \\
& start, spawn : \mathsf{Unit} \to \mathsf{Unit} \\
& (\!|, |\!) : \mathsf{Unit} \to \mathsf{Unit} \\
\hline
& l : \vec{T} \to T \\
& \vdots \\
)\!] .
\end{array}
$$

The methods above the horizontal line are private, i.e., hidden from the environment via

[3] Two different cliques cannot be coordinated, of course, as they are unconnected by definition and enforcing mutex would require at least some bit of shared information.

[4] Locking the whole, distributed network of the clique objects looks harder than it is. As we are after may-testing, only, we need not worry about deadlock, let alone loosing liveness or fairness. It suffices that any failed attempt to obtain the lock simply blocks or diverges, thus foiling success.

subtyping. The publicly available methods, i.e., those mentioned in the type interface of the component, are below that line: l,. . . .

B.2.1 Input and output synchronization

The code operates on the scripts (later implemented as a statically determined number of instance variables) containing the current future(s) of an object, resp., clique, relying on further auxiliary operations performing initialization, broadcasting of information, updating the set of known references, and shortening the still open futures, etc. The behavior of the corresponding methods, shown below, should be clear at an intuitive level, looking at the code; their properties and implementations are presented later. In this section, we concentrate on the two kinds of code, responsible for synchronization at the top-level, namely t^i_{sync} and t^o_{sync} for input and output. Top-level in the sense that this is the code, which appeared in the Definition 3.3.20 resp. 5.2.7 of C_t.

Input

We start with the code for *input*, executed immediately after each incoming communication. At an abstract level, given the current futures of a clique, it shortens the available future in accordance with the (input) action a just occurred, potentially merging a number of cliques. If new component instances are created, they are properly initialized. If the current incoming communication is not consistent with any possible future, the thread blocks.

Definition B.2.2 (Synchronization: Input (cf. Lemma B.3.1))**.** *The code for synchronization at the beginning of a method l with formal parameters $\vec{x}$ and inside a component class c of type $[\![\ldots, l{:}\vec{T} \to T, \ldots]\!]$ is given as:*

$$
\begin{array}{lll}
t^i_{sync}(l,\vec{x}) \triangleq & (\!| & a := label_l(\vec{x}); \\
& & initialize(a); \\
& & \quad let \quad \vec{o} : \mathsf{set\ of\ object} = pickrepresentative(a) \\
& & \quad in \quad self.step^i(a,\vec{o}) \\
& & \qquad\quad self.spawn(); \\
& & \qquad\quad \vec{o}.broadcast(self.scripts) |\!) \ .
\end{array}
\tag{B.3}
$$

In the single-threaded setting, the $self.spawn()$ is absent. We use $t^i_{sync}(l,\vec{x})$ as metamathematical notation and not to indicate that l and $\vec{x}$ receive values by argument passing in a reduction step.[5] *For incoming returns, the definition is used analogously. However, the method label is unimportant, and we write $t^i_{sync}(return, x)$ in that case, where x is the let-bound local variable used to receive the return value (cf. the operational rule* RETI*).*

The $a := label_l(\vec{x})$ remembers the label in the local instance state, using the formal parameters of the method plus the method name l in case of a call; for returns, the label is determined by $a := label_{return}(x)$. The label, with the references filled in, is then referred to as $self.a$ or a for short in the method body. Note that the code of $label_l(\vec{x})$ (resp., of $label_{return}(x)$) is the *only* part

[5] As $\vec{x}$ represents the formal parameters of the method l and as the variables $\vec{x}$ occur free in t^i_{sync}, there is some form of "parameter passing", however, by the substitution which is part of the parameter passing of method calls.

of the code of t^i_{sync} mentioning the formal parameters $\vec{x}$ (resp., x). After been stored in the instance state, the passed values are looked up from the fields of the resp. object.

After assigning the value to the instance variable a, the $initialize$-method, invoked on an (uninitialized) component object, initializes the $scripts$ data structure, filling in the identity of the newly created objects into all roles. Afterwards, the futures of all objects in the clique are shortened (if possible) according to current incoming action a. In more detail, the target object of the communication —the "self" object— is responsible for shortening. However, it needs to consult the partner cliques being merged in the current step. The consultation concerns for instance the local "view" on the future and on the involved object identities. For each of the partner cliques, the target object needs the information only once; hence it chooses one representative for each clique, via the $pickrepresentative$-method. Once the future has been shortened and the data structures have been locally updated (i.e., after having executed the $step^i$-method), the new state is broadcast to all object of the (now merged) clique. At that point, in the *multithreaded* setting, an appropriate number new threads is spawned, which start to run asynchronously.

In the multithreaded setting, all data manipulation is done under mutual exclusion (at component clique level), enforced by $(\!|$ and $|\!)$. In the sequential setting, the bracketing $(\!|\_|\!)$ is absent (or implemented via some skip-statement). Indeed, it would do no harm if $(\!|$ and $|\!)$ were functionally present in the single-threaded setting, as well.[6]

Remark B.2.3 (Initialization)**.** *The initialization in equation (B.3) is invoked on* all *incoming arguments referring to component objects, not just on those yet uninitialized. Invoked on an already initialized object, the method is without effect. The reason for this strategy is that the receiver of an identity has no means to detect, that the object is* globally *new and thus not yet initialized. In case of merging cliques, the received identity is* locally *new to the receiving clique; however, the object had a prior, independent existence and history, and is already initialized. Cf. the handling of binders in the definition of projection in Definition 3.1.3, where similar considerations are relevant, albeit at the more abstract level of traces, not at the level of code. The parallel is that the code of $initialize$ implements ν-binders occurring in the (local* projection *of a) trace.* □

Definition B.2.4 (Initialization)**.** *Each class c is equipped with a method $initialize$ of type* Unit $\to$ Unit*. Assume that $y_1, \ldots, y_n$ are the roles of type c. Then*

$$
\begin{array}{lcll}
initialize & \triangleq & \mathsf{if} & self.scripts \neq \bot \\
 & & \mathsf{then} & () \\
 & & \mathsf{else} & \sigma_1 := \sigma_1[y_1 \mapsto self]; \\
 & & & \vdots \\
 & & & \sigma_n := \sigma_n[y_n \mapsto self]\ .
\end{array}
\tag{B.4}
$$

The $initialize$-method of type label $\to$ Unit*, as used in t^i_{sync} (equation (B.3)), invokes $o.initialize()$ from equation (B.4) on all component references of a.*

[6] The reason why it would do no harm is: The implementation of the synchronization code avoids *recursion* and thus we do not need re-entrant calls to the monitors.

Definition B.2.5 (Spawn)**.** *The method* $spawn$ *of type* Unit $\to$ Unit *is given as*

$$spawn() \triangleq new\langle self.start(); \ldots; self.start()\rangle \ , \tag{B.5}$$

where the number of new*-expressions if given by the number of component-generated threads in the given trace* t*.*

The next method $collectroles$ is part of input synchronization, i.e., part of the code of $step^i$ (cf. Definition B.2.8).[7] From the perspective of one particular role association (the σ of the argument) of one object as representative of its clique, it collects from a number of (still) separate cliques the matching combinations of associations. As precondition, we assume that the argument objects are members of pairwise disjoint component cliques, which in addition are disjoint from the clique invoking the method which is the object to which the argument association belongs to.

Since the domains of the associations σ correspond to (the static representation of) the cliques, the $collectroles$-method correspond to finding matching (disjoint) cliques to merge. It combines the current role association σ with combinations of associations of other cliques.

Definition B.2.6 (Collect roles)**.** *The method* $collectroles$ *is of type:*

$$\mathsf{assoc} \times (\mathsf{set\ of\ object}) \to \mathsf{set\ of\ assoc} \ . \tag{B.6}$$

Its code is given as

$$collectroles(\sigma, \vec{o}) \triangleq \{\sigma' \mid \sigma' = \sigma \oplus \bigoplus_{o \in \vec{o}} \sigma'' \in o.\Sigma\} \ . \tag{B.7}$$

The operation $\oplus$ *on associations is defined as follows:*

$$\sigma_1 \oplus \sigma_2 = \begin{cases} \sigma_1 \cup \sigma_2 & \textit{if } dom(\sigma_1) \cap dom(\sigma_2) = \emptyset \\ \bot & \textit{else} \ . \end{cases} \tag{B.8}$$

Most of the work for input synchronization is done in the method $step^i$. Abstractly and for one object, namely the one on which it is invoked —the target object of the communication— $step^i$ checks whether the next action a is possible, i.e., whether it appears as next step in the still open futures (there might be more than one matching continuation). If so, the respective futures are shortened by one; if there is no continuation, the code blocks.

More concretely, the implementation is a bit more complex due to the fact that the step a under consideration may be a *merging* step. This makes it more complex in that the object which executes the code (as representative of the target clique of the communication) may not have the full information wrt. already taken roles. The taken roles, the static analogs to object identities, are kept in the *domain* of the associations σ_k. Before the merge, the still separate cliques are guaranteed to have disjoint *ranges* in all their associations as far as identities of *component* objects are concerned, since the ranges correspond to the dynamic set of identities in the actual trace. The roles in the *domains* of the association, however, are in general not disjoint, since the so far separate

[7] Newly created component objects during output synchronization need not be initialized; they simply adopt the data structures of the creating clique.

cliques may have (and in general will have) associated the same roles with their so far encountered component object identities. Of course, a role cannot be associated with two different identities. The code of $collectroles$ thus combines the role associations from the futures of the partner cliques, weeding out impossible combinations.

After combining the associations, for each remaining open future, the next action is checked against the current one, and the respective future is shortened accordingly, or invalidated.

Remark B.2.7 (Association domains)**.** *As the roles in the domain of the associations correspond to the evolving clique structure of the original trace, the domains of two associations σ_1 and σ_2 of two separate cliques are either disjoint, or else in subset relation (cf. Corollary A.3.4). If disjoint, they can be combined via $collectroles$, using $\oplus$, if not, no common future is possible.* □

Definition B.2.8 (Input step)**.** *Each component class is equipped with a method $step^i$ of type* label $\times$ set of object $\rightarrow$ Unit*. Its code is given as*

$$
\begin{array}{l}
step^i(a{:}\mathsf{label}, \vec{x}{:}\mathsf{set\ of\ object}) \triangleq \qquad \text{(B.9)}\\
\quad \forall \quad (\sigma_i, \breve{a}_i\breve{s}_i) \in self.scripts.\\
\qquad \mathit{let} \quad \{\sigma_i^1, \ldots \sigma_i^m\} = collectroles(\sigma_i, (\vec{x} \setminus self))\\
\qquad \mathit{in} \quad \mathsf{if} \quad \exists \sigma_i^j.\ \breve{a}_i\sigma_i^j = a\\
\qquad\qquad \mathsf{then} \quad self.scripts_i := (\sigma_i^j, \breve{s}_i)\\
\qquad\qquad \mathsf{else} \quad stop \quad .
\end{array}
$$

Output

Next the code of output synchronization, executed immediately *before* each output. The task here is to use the $scripts$ to determine an output reaction, i.e., whether to respond with a return or to fire another call, and in each case determine the run-time values. If there is no future left, the thread must terminate.

The code is shown in Definition B.2.10 and Definition B.2.11 below. The function *picks* from all the still active futures in $self.scripts$ one representative; in the code, the representative is denoted by σ for the role association and $\breve{a}$ for the first action of chosen future (the rest of the future is not needed for performing the next step. Thus, the value is represented by the _-wildcard in the definition).

Remark B.2.9 (Deterministic/non-deterministic setting)**.** *Note that the next (output) actions $\breve{a}_i$ and $\breve{a}_j$ of two different scripts may indeed be literally* different *even in the deterministic setting.*

This fact should come as no surprise in the non-deterministic, multithreaded setting: After a given history, an object may show different reactions. In the deterministic *setting, where the scripts are derived from a deterministic trace, this indeterminacy wrt. the next action has a different reason, which additionally is present in the multithreaded case, of course. For a deterministic program, the reaction of a clique of objects is* determined *by the past interaction, but only up-to* renaming*. The "surviving" scripts in the state of the (dynamic) clique reflect the fact, that the actual and unique history of the current clique corresponds to the simultaneously executed static sequences of actions encoded from the original trace.*[8] *So if still two different scripts*

[8]The static pasts (and the actual dynamic past) are not remembered; what is stored is the role association of actual identities with the static roles.

are open at a current state, it means that in the original trace there exist, at the corresponding point in the trace, two component cliques in the same state up-to renaming. Cf. Definition 3.1.10 for the definition of deterministic trace.

Due to determinism, their output reaction must be equivalent, implying that the static analogs $\check{a}_i$ and $\check{a}_j$ are equivalent up-to different uses of roles. Even stricter, $\check{a}_i\sigma_i$ and $\check{a}_j\sigma_j$ must be equivalent up-to different roles mentioned "boundedly".

To sum up: Determinism of the original trace allows to just "pick" one representative $\check{a}$ and a corresponding association σ, if there is more than one available. The chosen representative $\check{a}$ is used to create new component and environment instances and storing the association temporarily in σ'. This association reflects the choice of roles as in $\check{a}$, i.e., it assumed that the newly generated references will from now on take the roles according the ν-bound roles mentioned in $\check{a}$. An alternative script, starting with $\check{a}'$ instead of $\check{a}$, will store the same *freshly created references in different roles. For instance, if $\check{a}$ and $\check{a}'$ are of the forms*

$$\nu(x_0,x_1).\langle call\ x_0.l(x_1,x_2)\rangle! \quad and \quad \nu(x_0',x_1').\langle call\ x_0'.l(x_1',x_2')\rangle!$$

respectively, then the newly created association $\sigma' = create(\check{a})$ is if the form $[x_o \mapsto o_0, x_1 \mapsto o_1]$, where o_1 and o_2 are freshly created references.[9] *For the alternative $\check{a}'$, however, the created references o_0 and o_1 are the same —even if the implementation executes all possible scripts simultaneously, the dynamic* references *need to be shared between the scripts— but take the roles x_0' and x_1' instead, i.e., the corresponding additional role association is $[x_0' \mapsto o_0, x_1' \mapsto o_1]$. This "renaming" of the roles is done by iterating over all still active scripts, where $\sigma' \circ \pi(\check{a},\check{a}_i)$ calculates the permutation of the roles.* □

After the post-state for each script has been determined in that way for the communication's source object, the information of the corresponding clique is brought up-to date via broadcast. With the script data structures shortened, finally the corresponding output must be actually performed, as well. This is done by the interpret-function (cf. Definition B.2.12).

Definition B.2.10 (Output step)**.** *Each component class is equipped with a method:*

$$\begin{array}{lll} step^o() & \triangleq & let \quad (\sigma,\check{a}\_) \in self.scripts \\ & & in\ let \quad \sigma' = create(\check{a}) \\ & & in \quad \forall(\sigma_i,\check{a}_i\check{s}_i) \in self.scripts \\ & & \qquad let \quad \acute{\sigma}_i = \sigma_i \oplus \sigma' \circ \pi(\check{a},\check{a}_i); \\ & & \qquad in \quad scripts_i := (\acute{\sigma}_i,\check{s}_i); \\ & & self.broadcast(self.\Sigma); \\ & & \quad let \quad a = \check{a}(\sigma \oplus \sigma') \\ & & \quad in \quad self.interpret(a) \quad . \end{array} \tag{B.10}$$

The code for output synchronization invokes $step^o$, after locking the clique using (|. Note that lock-release (|) is not mentioned directly in (B.11), but is part of the interpret-method (via $step^o$ and executed at the end of t^o_{sync}).

Definition B.2.11 (Synchronization: Output (cf. Lemma B.3.2))**.** *The code for output synchronization of type* Unit → Unit *is defined as follows:*

$$t^o_{sync}() \triangleq \ (|\ self.step^o()\ . \tag{B.11}$$

[9] For environment objects, e.g. for o_0, only the references are generated; the corresponding instance will be created later lazily when the name extrudes to the environment. For o_0, this will be the case, when the actual call is issued.

The interpret method is used for output synchronization, in particular in $step^o$ of equation (B.10). It can be seen as the dual to $label_l(\vec{x})$ and $label_{return}(x)$. When handed over a return label, it simply extracts the value and gives it back which then will be returned to the environment. In case of an outgoing call label, it similarly extracts the relevant information and fires that call, with trailing synchronization code appropriately added so to handle a possible return from the environment and a possible next output action afterwards. If no label is handed over, the method terminates the thread.

Definition B.2.12 (Interpret)**.** *Each component class is equipped with an private method* $interpret$ *given as follows:*

$$\begin{array}{lllll} interpret(a:\mathsf{label}) \triangleq & case\ a & & & \\ & \quad \bot & \mathsf{then} & stop & \\ & \quad n\langle return(v)\rangle! & \mathsf{then} & \rangle\!\rangle; v & \\ & \quad n\langle call\ o_r.l(\vec{v})\rangle! & \mathsf{then} & \rangle\!\rangle; & \\ & & & let & y = o_r.l(\vec{v}) \\ & & & in & t^i_{sync}(return, y); t^o_{sync}() \\ & esac\ . & & & \end{array} \tag{B.12}$$

Part of performing output is the creation of the entities, which are *new* to the environment. This is done quite at the beginning of $step^o$ in Definition B.2.10. The creation concerns environment objects exported by lazy instantiation, component objects exported by scope extrusion, and new threads. The $create$-operation is responsible for the creation only, but not for initialization. Newly created environment objects are not initialized, of course; internal objects are initialized, i.e., equipped with the appropriate values for the scripts variable only after creation, as part of $step^o$.

Definition B.2.13 (Object creation)**.** *Each component class is equipped with a private method* $create$ *of type* $\mathsf{label} \to \mathsf{assoc}$*, given as follows:*

$$create(\nu(\check{\Phi}).\lfloor\check{a}\rfloor) \triangleq \ \sigma_\bot[\check{\vec{o}} \mapsto new\,\vec{c}]; \qquad where \quad \check{\Phi} = \check{\vec{o}}{:}\vec{c}, \check{\Sigma} \tag{B.13}$$

i.e., $\check{\vec{o}}{:}\vec{c}$ *contains all the (static representation of) new objects in the label. The actual creation of* new *is left implicit in the* $\mapsto$*-expression.*

Remark B.2.14 (Reflection)**.** *Note* $create$ *from equation (B.13) uses "reflection" in some sense. In interpreting the binding part of the label, it interprets a "binding"* $\check{o}{:}c$ *listed in* $\check{\Phi}$ *as instruction to execute* $new\,c$ *(and store the result appropriately). This means that in the static encoding of labels* $\check{a}$*, traces* $\check{t}$*, etc., we need an internal representation for each class* c *in the system (which we denoted here by* c *itself).* □

Spawning new threads

The $create$ of Definition B.2.13 creates no new threads. Its code might be of course executed by a newly created thread as part of its first external activity, i.e., as part of its output synchronization in preparation of the first outgoing call. Thread creation is different from object creation: The input and output synchronization is executed *by* an already existing thread which interprets the scripts such that it creates new objects at the current point in the script. When

a new thread crosses the interface, it means a new thread must have been spawned. For incoming threads, this is not a problem; it is the responsibility of the environment to generate them. When a new thread is exported to the environment, the component must create that thread and let it run. The code for creation must be executed by *another* thread, and the corresponding $new\langle t\rangle$ statement must ultimately be contained in some method.

In general, the implementation deals with thread creation as follows: It simply creates *enough* threads *as soon as possible.* Upon creation, the new thread remains hidden; it only becomes visible when it starts executing and starts interpreting the script data structure. In principle, the implementation could let loose all the threads at the very beginning, if it were not for the connectivity and the heap structure. As expressed in the corresponding rules L-CALLO$_0$ and L-CALLI$_0$ of the legal trace system, the new thread is connected to one already existing clique and is acquainted with the references of that clique; if there is more than one candidate clique as originator, the rules non-deterministically guess one.

Thus, C_t cannot start all threads at the beginning and irrespective of the clique structure, but must start enough threads per component clique. Since new cliques can be created only by incoming communication, the corresponding spawning of new threads is part for input synchronization, only.

After a new thread is created, the spawning thread continues asynchronously; both threads execute their respective code independently, apart from potential shared access to (parts of) the heap. The new thread has connection to the clique in which the spawning thread is executing the new-expression. In particular the new thread can access the corresponding clique via $self$.[10] However, the guessing of the sender at a given point in the trace does not determine *when* the thread has been as been created nor *which thread* has spawned the new one. Indeed, the creation can have taken place at any point in time from the creation of the (first object in the) guessed clique till the new thread appears at the interface.

Example B.2.15 (Thread creation)**.** *Consider the following trace*

$$\begin{array}{l}\nu(o_r^1).n\langle call\ o_r^1.l_1()\rangle?\ n\langle return()\rangle!\ \nu(o_r^2).n\langle call\ o_r^2.l_2()\rangle?\ n\langle return()\rangle! \\ \nu(n_1'{:}thread).n_1'\langle call\ o_r.l()\rangle!\ \nu(n_2'{:}thread).n_2'\langle call\ o_r.l()\rangle!\ .\end{array} \tag{B.14}$$

The environment, using a thread n, creates two component cliques, represented by o_r^1 and o_r^2. After the interaction with the two cliques, the component reacts with 2 outgoing calls, where without further information, for both the sender is unknown, it can be either of both cliques. The corresponding rules for legal traces, L-CALLO$_0$ guesses *the origin of the first outgoing call by either $\Theta \vdash o_r^1$ or $\Theta \vdash o_r^2$ in the premise, and analogously when guessing the sender of the second outgoing call, when Θ is the corresponding commitment context.*

If the senders are o_r^1 and o_r^2 for the two outgoing calls respectively, the static scripts look as follows:

$$\begin{array}{l}\nu(\check{o}_r^1).\check{n}\langle call\ \check{o}_r^1.l_1()\rangle?\ \check{n}\langle return()\rangle!\ \nu(\check{n}_1'.{:}thread).\check{n}_1'\langle call\ \check{o}_r.l()\rangle! \\ \nu(\check{o}_r^2).n\langle call\ \check{o}_r^2.l_2()\rangle?\ \check{n}\langle return()\rangle!\ \nu(\check{n}_2'{:}thread).\check{n}_2'\langle call\ \check{o}_r.l()\rangle!\ .\end{array} \tag{B.15}$$

[10] In a setting with thread classes, one needs *constructors* to pass on arguments to the new thread. Otherwise it would execute independently of already existing part of the heap. See e.g. [12].

In that case, the execution of the method body of l_1 spawns two *new thread, i.e., it executes $new\langle self.start()\rangle$ twice as part of the spawn-method. (cf. Definition B.2.5). Only one of the two will be able to actually perform the outgoing call later.* □

Remark B.2.16 (Thread creation and mutual exclusion)**.** *The spawning of new threads is part of the input synchronization code, which is executed under mutual exclusion, i.e., protected by $(\!|$ and $|\!)$. Being spawned inside the protected region does* not *grant the new thread access to the data structures of the clique of the spawner. The new thread starts executing* outside *the monitor and needs to acquire the lock before accessing the data structures. Cf. Definition B.2.17 for the start-method.* □

Remains the code to actually start a thread. This is already needed in the sequential setting where it is invoked exactly once, namely at the very beginning in case the initial threads starts executing in the component. In the multithreaded setting, the start method is invoked additionally for every thread created by the component. The implementation is simple: It just triggers the code for output synchronization, which then starts interpreting the script(s).

Definition B.2.17 (Start)**.** *The start-method of type* Unit $\to$ Unit *is given by*

$$start() \triangleq t^o_{sync}() \ . \tag{B.16}$$

B.2.2 Mutual exclusion

In the multithreaded setting we must assure that the data-handling is done under mutual exclusion to avoid data corruption. In the may-testing setting, we do not need to solve the general mutual exclusion problem, i.e., we do not need to worry about the more complex requirements [48][49][89][119] like liveness, fairness, non-starvation, etc. The concentration on the core safety requirement, namely absence of interference, simplifies the implementation. Basically we have to implement a rudimentary monitor- or lock-mechanism, which assures mutual exclusion *per clique.* It suffices to detect, when mutual exclusion is violated and then stop executing.

On the other hand, the implementation task is complicated by the fact that our language is rather restricted. In particular, the calculus does not offer built-in synchronization capabilities such as *Java*'s `synchronized` methods or synchronized blocks, which at least can assure mutual exclusion on a per-object basis. Worse still, the calculus offers nothing but object references as native data. On the level of references, the calculus allows *atomic update* (via rule FUPDATE of Table 2.5), atomic read, and to a limited extent atomic comparison (via the two COND$_i$-rules). The comparison, however, is atomic only if the entities being compared are already evaluated to references, i.e., a redex of the form if $o_1 = o_2$ then t_1 else t_2 reduces in an atomic step, as justified by one of the two COND$_i$-rules.

More realistically, one would be interested in executing

$$\text{if } self.x_1 = self.x_2 \text{ then } t_1 \text{ else } t_2 \ ,$$

i.e., comparing the values of instance variables, and on this level, atomicity is *not assured.* In particular, we do not have an atomic test-and-set operation (or similar luxury) on instance variables. Moreover, all more high-level data (e.g., booleans) is to be encoded by groups of instance variables, in particular the

lock-mechanism needs to be implemented by instance variable(s), and so the question is:

> How to use object references to implement the safety aspects of mutual exclusion on a per-clique basis?

Remark B.2.18 (Reentrant monitor locks)**.** *The lock mechanism must assure mutual exclusion between concurrent threads per clique. The calculus allows recursion, so in principle we need to consider situations where a thread owning the (to be implemented) lock re-enters the monitor via a recursive call. A mechanism allowing such a behavior is called a* reentrant *monitor [73][31] and needs a more complex data structure than a simple binary flag to realize the locking mechanism. Basically, it needs to remember the thread that owns the lock and how many recursive calls deep this thread resides in the monitor. This is needed to detect when to release the lock again, namely when the recursion depth of the lock-owner has reached zero again. If the maximal recursion depth cannot be statically determined —and in general it can of course not— this calls for an* unbounded *data structure. This means, our standard data encoding trick, putting everything in a statically predefined ensemble of instance variables, would fail.*

We need, however, the lock mechanism only *for the observer we construct during the completeness proof. In that chosen implementation, we do not need locks counting the recursion depth, since we only protect the protect the book-keeping associated with each individual label (after incoming communication, resp., before outgoing communication), but not protect whole method calls (from the call till the matching return). In this sense, we do not need to implement reentrant monitors: A thread that has entered a component clique leaves the clique* only *by leaving into the environment, and at that point it* releases *the lock with all data in consistent condition. If the thread reenters the same clique later, it needs to re-acquire the lock.*

However, for convenience, we implement slightly more complex locks than simple binary flags. Besides the fact whether the lock is taken or not, we remember the name of the thread which owns the lock (without counting how deep the recursion depth of the thread inside the monitor is, as just explained). □

Before presenting the implementation of the lock in Definition B.2.20 and B.2.21, we show how to implement its core, namely a boolean flag. Apart from testing for definedness of an instance variable, conceptually and implicitly, the *only* built-in boolean expression in our language is the equality-test on identities. Thus it suggests itself to represent the "boolean" value on the comparison of references.[11]

Definition B.2.19 (Boolean flag)**.** *Given a class c, a boolean flag for instances of that class is a pair of instance variables $x_1 : c$ and $x_2 : c$, both initially carrying the code $\bot$, i.e., as all instance variables, being "undefined" initially.*

The "value" false corresponds to $x_1 = o$ and $x_2 = o'$ for two object references with $o \neq o'$; correspondingly $true$ when $o' = o$, i.e., checking for the flag being true in a conditional is encoded as follows, where o_{self} is the object the code is executed in:

$$\mathsf{if}\ x_{flag}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \quad \triangleq \quad \varsigma(s{:}c).\lambda().\ \begin{array}{ll} let & y_1{:}c = s.x_1\ in\ let\ y_2 = s.x_2 \\ in & \mathsf{if}\ y_1 = y_2\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ . \end{array} \tag{B.17}$$

[11] The comparison $o_1 = o_2$ is a boolean expression only implicitly, since it is by itself not an expression, but occurs only as part of the conditional expression. Note also testing for (un)definedness using the "expression" $undef(v.l)$ does not implement a boolean flag, because one cannot reset a defined value to the native $\bot_c$.

Setting the flag to false is defined as

$$x_{flag} := \mathit{false} \quad \triangleq \quad \varsigma(s{:}c).\lambda().\ \begin{array}{ll} \mathit{let} & y_1 = \mathit{new}\, c\ \mathit{in}\ \mathit{let}\, y_2 = \mathit{new}\, c \\ \mathit{in} & s.x_1 := y_1; s.x_2 := y_2\ . \end{array} \tag{B.18}$$

Definition B.2.20 (Lock (cf. Lemma B.2.22))**.** *Each class c contains as* lock *the following triple of instance variables: $x_1 : c$, $x_2 : c$, and $owner : thread$.*

As mentioned, the reading and writing of the boolean flag is *not atomic.* We need to be careful, therefore, when acquiring the lock. We use the *uniqueness* of freshly generated names for our mutex protocol. When successful, the instance variable $owner$ is set to the identity of the thread then holding the lock. Note that, unlike many of the constructions so far, the code of Definition B.2.21 is native calculus code.[12] Assuring mutual exclusion is a detail, however, a crucial one, and thus we show the implementation only using the bare means of the language, in particular, using only object references as data, and without further layer of abstraction. Once, mutual exclusion is guaranteed, we are dealing "only" with sequential and finite data structures and operations thereon.

Definition B.2.21 (Lock handling (object level))**.** *The lock is manipulated by two operations, acquiring and releasing the lock, written $(\!|$ and $|\!)$.*[13] *The operations are coded as follows:*

$$\begin{array}{lcl} (\!| & \triangleq & \varsigma(s{:}c).\lambda().\ \mathit{let}\quad x_1^{local} : c = s.x_1\ \mathit{in} \\ & & \quad\ \mathit{let}\quad x_1^{new} : c = \mathit{new}\, c\ \mathit{in}\ \underline{s.x_1 := x_1^{new}}; \\ & & \qquad \mathit{let}\quad x_2^{local} : c = s.x_2\ \mathit{in} \\ & & \qquad\ \mathsf{if}\ x_1^{local} \neq x_2^{local}\ \mathsf{then}\ \mathit{stop}; \\ & & \qquad\ \mathsf{else}\ \ \mathit{let}\quad y_1{:}c = s.x_1\ \mathit{in} \\ & & \qquad\qquad\ \mathsf{if}\quad y_1 = x_1^{new} \\ & & \qquad\qquad\ \mathsf{then}\quad \mathit{owner} := \mathit{currentthread} \\ & & \qquad\qquad\ \mathsf{else}\quad \mathit{stop}\ . \end{array} \tag{B.19}$$

$$\begin{array}{lcl} |\!) & \triangleq & \varsigma(s{:}c).\lambda().\ \mathit{let}\quad x{:}c = \mathit{new}\, c \\ & & \mathit{in}\quad s.\mathit{owner} := \bot; s.x_2 := x; s.x_1 := x\ . \end{array} \tag{B.20}$$

Lock handling is illustrated in Figure B.1. The pair of instance variables encoding the flag are shown in the middle (as circles) of the picture. The two values are initially equal, indicating that the lock is free. The thread on the left succeeds in finishing the protocol and thus acquires the lock, in that it replaces the value of x_1 by a freshly generated identity (the diamond shape). After the assignment $\mathit{self}.x_1 := x_1^{new}$, the values of $\mathit{self}.x_1$ and $\mathit{self}.x_2$ are unequal, which means, a second thread, which reads $\mathit{self}.x_1$ into its local store after this assignment, will not succeed in entering the critical section. The convention interpreting $x_1 = x_2$ as free lock and $x_1 \neq x_2$ as lock taken is thus not arbitrary in this protocol, the interpretation is not symmetric.

[12] Well, the only deviations are the following two: We use the (1) sequencing operator ; and the negated comparison (2) $x \neq y$ in the comparison. Both are instances of trivial syntactic sugar.

[13] Using standard jargon, we call $(\!|t|\!)$ also an *atomic region* or a *bracketed section.* When using this notation we assume that t neither contains further interaction with the lock of the concerned object, in particular not nested calls of $(\!|$ and $|\!)$, nor interaction with the environment. Note the new threads can be spawned in t (in the multithreaded setting) but they start running "outside" the monitor.

A thread enters the "trying section" of the protocol, i.e., it expresses its wish to acquire the lock, by setting x_1 atomically to a fresh reference, which renders $x_1 \neq x_2$. This setting itself is atomic; if we used $x_1 \neq x_2$ to represent a free lock, an atomic entering of the trying section would not be possible, as it would involve updating both x_1 and x_2.[14] As there is an unbounded number of fresh references available, the protocol works for an arbitrary number of threads.

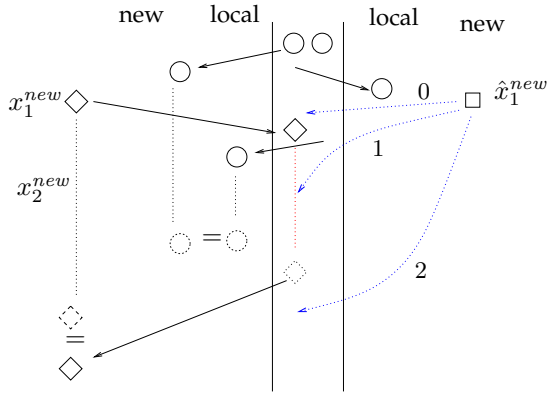


Figure B.1: Acquiring a lock

Of course, the *reading* and subsequent comparison of x_1 and x_2 is not atomic. Thus, obviously, $x_1^{local} = x_2^{local}$ in the code of equation (B.19) can *not* be taken as sign to enter the critical section; the opposite $x_1^{local} \neq x_2^{local}$, however, is taken as sign to give up and stop. Thus, x_1, the variable of the two, which is updated first in the trying section, is read for a second time. If still unchanged, the thread can safely enter the critical section.

Considering the assignment $self.x_1 := \hat{x}_1^{new}$ of a second thread, it cannot successfully happen in position (1) of the figure, as this would prevent the success of the thread on the left-hand side. If the assignment happens at position (2), the corresponding thread on the right will not be able to finish the code of $(\!|$, since independent of which value of x_1 it has copied into its local memory, x_2 will contain a different reference, which terminates the thread. The same happens, if the thread on the right executes the assignment at point (0). Again, the unbounded reservoir of fresh references plays a crucial role.

A remark about the *initialization* of object locks for new objects. Upon creation, the value of the lock is "undefined". However, the creation of a new component object is done *within* a critical section for some clique. When finished with the corresponding synchronization code, $|\!)$ is executed for all objects of that clique, *including* the newly created ones, setting the lock to the status "free". Thus, locks yet uninitialized are never accessed by $(\!|$.

Lemma B.2.22 (Mutex)**.** *The implementation of $(\!|t|\!)$ guarantees mutual exclusion at object level.*

Proof. It goes without saying that we assume that t does not contain further instances of $(\!|$ and $|\!)$ (concerning the same object) or other fiddling with the instance variables x_1 and x_2 implementing the lock. Note that in the constructed

[14] This does not mean that an implementation using $x_1 \neq x_2$ to denote the free lock is impossible.

observer, t contains code that spawns new threads. The new threads, however, apply for the lock before they can access any shared data.

Assume an arbitrary number of parallel threads inside an object o, with the lock free, i.e., with $o.x_1 = o.x_2$. If the lock is not free from the beginning, obviously no thread can enter the critical section. So assume for a contradiction that two threads succeed in entering the critical section, i.e., a situation

$$n_1\langle t_1 \rangle\!\rangle \quad \text{and} \quad n_2\langle t_2 \rangle\!\rangle$$

after some reduction. As, by assumption, both threads reach the end of $\langle\!|$, the sequence $n_1\langle o.x_1 := x_1^{new} \ldots\ldots \mathit{let}\ y_1{:}c = o.x_1\rangle$ of equation (B.19) occurs completely before the analogous sequence $n_2\langle o.x_1 := x_1^{new} \ldots\ldots \mathit{let}\ y_1{:}c = o.x_1\rangle$ in thread n_2 (or vice versa); otherwise, for (at least) one of the two threads, the respective local variable y_1 contains afterwards a different reference than x_1^{new}, which prevents the completion of $\langle\!|$.

Assume then wlog. that the mentioned sequence of n_1 precedes the one of n_2. After $n_1\langle o.x_1 := x_1^{new} \ldots\ldots \mathit{let}\ y_1{:}c = o.x_1\rangle$ of n_1, $o.x_1 \neq o.x_2$ is guaranteed; hence the comparison $x_1^{local} = x_2^{local}$ in thread n_2 fails, which prevents n_2 to complete $\langle\!|$ and to enter its critical section t_2, which contradicts our assumption. □

The next lemma is a variant of the above mutex lemma. If differs in that now we assume that the critical regions of threads accessing the same lock are executed *successfully.* Lemma B.2.22 showed the basic safety property of mutual exclusion, namely that never the critical sections are executed at the same time, where obviously one possibility of assuring this is to stop within $\langle\!|$ (in the code of equation (B.19), there are two points where this may happen). Now we explore the consequences for the reductions assuming that especially the $\langle\!|$-code does *not* fail. This additional knowledge gives a finer view on which code is executed under mutual exclusion. To formulate the lemma, we introduce the following abbreviations. The two relevant atomic, elementary steps —the points of no return— in the code of $\langle\!|$ (cf. equation (B.19)) are

- the first copying of $s.x_1$ into the local store by a $\xrightarrow{\tau_r}$-step, i.e., the $x_1^{local}{:}c = s.x_1$ in the first line, and
- the replacement of $s.x_1$ by a fresh identity by a $\xrightarrow{\tau_w}$-step, i.e., the $s.x_1 := x_1^{new}$ in the second line.

We denote by $\langle\!|_r$ the code of $\langle\!|$ starting in front of the read-step, and $\langle\!|_w$ the code starting in front of the write-step. Concerning the end $|\!\rangle$ of a critical region, there is no such uncertainty. The very last action of $|\!\rangle$, i.e., the update of $s.x_1$ to the newly generated value which coincides afterwards with $s.x_2$ marks the exact end of the code executed under mutual exclusion. Note that in $|\!\rangle$, first $s.x_2$ is assigned the new reference, and afterwards $s.x_1$, which is the reverse order in which the variables $s.x_1$ and $s.x_2$ are read in $\langle\!|$. The finer knowledge about mutual exclusion is needed for "disentangling" the atomic regions.

Lemma B.2.23 (Mutex). *Assume $\Xi_0 \vdash C_t \Longrightarrow \Xi \vdash C \Longrightarrow \acute{\Xi} \vdash \acute{C}$, where in the reduction sequence, two threads n_1 and n_2 both execute $\langle\!|$ successfully on the lock of the same object o. Assume that initially the lock is free (i.e., initially, x_1 and x_2 contain the same value, and the owning thread is undefined). The implementation of*

$(\!|\ \_\ |\!)$ guarantees that the sequences $(\!|^1_w\ t^1\ |\!)^1$ and $(\!|^2_r\ t^2\ |\!)^2$ are executed under mutual exclusion.

Proof. The code for $(\!|$ and $|\!)$ is given in Definition B.2.21, equations (B.19) and (B.20).

Assume for a contradiction, that mutual exclusion in the form as stated in the lemma is violated, i.e., there is an overlap in the execution of

$$t^1_w = (\!|^1_w\ ; t^1\ |\!)^1 \quad \text{and} \quad t^2_r = (\!|^2_r\ t^2\ |\!)^2\ .$$

There are two cases to consider, namely whether in the overlapping execution, t^1_w does the first step, or t^2_r.

If t^1_w is first, then from its first $\xrightarrow{\tau_w}$-step until the very last step of $|\!)^1$, a $\xrightarrow{\tau_w}$-step, as well, the value of $s.x_1$ equals o^1_1, a value generated freshly by n_1, the first thread. By the assumption, that t^2_r comes after t^1_w and overlaps, n_2 necessarily reads o^1_1 into its local variable x^{local}_1. Since thread n_1 successfully completes its atomic region, its comparison of x^{local}_1 and y_1 must evaluate to the same reference such that $y_1 = x^{new}_1$ (in line 6). Therefore, the competitor n_2 performs its $s.x_1 := x^{new}_1$ *after* n_1 reads $s.x_1$ for a second time into its local variable y_1 (line 5). I.e., n_2's mentioned update occurs at point 2 or later in Figure B.1. This further implies for thread n_2 that $x^{local}_2 \neq x^{local}_1$: Either, x^{local}_2 reads the value of $s.x_2$ as it was at very beginning, i.e., before t^1_w started —the value of $s.x_2$ is read but not changed by the code of $(\!|$— or it already reads a value *after* the completion of $|\!)$, when $s.x_1$ and $s.x_2$ are overwritten by the same, freshly generated reference (cf. equation (B.20)). Because of the freshness of the generated references, in both situations, $x^{local}_2 \neq x^{local}_1$ for n_2, i.e., the thread fails to enter the critical region, contrary to our assumption.

Alternatively, t^2_r is first and again there is an overlap of t^1_w and t^2_r. Now, the reading $x^{local}_1 = s.x_1$ of n_2 copies the *original* value of $s.x_1$ (say o_0) to n_2's local space (line 1). If then n_2's first write action $s.x_1 := x^{new}_1$ (line 2) is so early that it precedes n_1's first read action $x^{local}_1 = s.x_1$, the situation is symmetric to the one just discussed, i.e., it leads to a contradiction. In order that n_2 succeeds in entering its critical section, its comparison $y_1 = x^{new}_1$ in line 6 must evaluate to true. This implies that n_1 performs its update to $s.x_1$ *after* the point where n_2 re-reads the value of $s.x_1$ into its local space (using y_1), i.e., n_1's mentioned update occurs at point 2 or later in Figure B.1. But at that point, the value of $s.x_1$ is already different from the original value o_0 and will never be o_0 again even after n_2 has completed its critical section. Therefore, n_1's first comparison $x^{local}_1 = x^{local}_2$ necessarily yields false, and hence n_1 cannot enter its critical section, contradicting our assumptions. □

Using this knowledge, we can disentangle the critical sections of two threads.

Lemma B.2.24 (Mutex disentangling)**.** *Let $\Xi_0 \vdash C_t$ be given by Definition 5.2.7 and assume $\Xi_0 \vdash C_t \Longrightarrow \Xi \vdash C \Longrightarrow \acute{\Xi} \vdash \acute{C}$. Assume that the threads n_1 and n_2 perform their critical section in total, i.e., n_1 performs the sequence $(\!|^1\ t^1_{critsec}\ |\!)^1$ and analogously for n_2, where both lock-handling codes refer to the same lock and where neither $t^1_{critsec}$ nor $t^2_{critsec}$ (of course) access the lock. Then also $\Xi \vdash C \Longrightarrow \acute{\Xi} \vdash \acute{C}$ such that*

$$\Xi \vdash C \underset{(\!|^1\ t^1_{critsec}\ |\!)^1}{\Longrightarrow\Longrightarrow\Longrightarrow} \underset{(\!|^2\ t^2_{critsec}\ |\!)^2}{\Longrightarrow\Longrightarrow\Longrightarrow}, \tag{B.21}$$

or the other way around. In the reduction sequence, we indicate the executed code below the arrow.

Proof. The code $\langle\!|$ for acquiring a lock and $|\!\rangle$ for release is given in Definition B.2.21, equation (B.19) and (B.20).

In $\Xi \vdash C \underset{n_1,n_2}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$, both n_1 and n_2 perform their critical section from the beginning of $\langle\!|$ till the end of $|\!\rangle$. In detail, the reduction for $\langle\!|; t_{critsec}$ looks as follows, where o is the target object whose lock is concerned and c its class. We show only the steps of the thread itself, not the whole component C.

$$\begin{array}{lll}
\langle\!|; t_{critsec} = & \varsigma(s{:}c).\lambda().\langle\!|_{body}; t_{critsec} & \xrightarrow{\tau_r^m} \\
& \langle\!|_{body}[o/s]; t_{critsec} & = \\
& let\ x^{local}{:}c = o.x_1\ in\ t'; t_{critsec} & \xrightarrow{\tau_r} \\
& let\ x^{local}{:}c = o_1\ in\ t'; t_{critsec} & \rightsquigarrow^* \\
& (let\ x_1^{new} = o^{new}\ in\ \underline{o.x_1 := x_1^{new}}; t''); t_{critsec} & \xrightarrow{\tau_w} \\
& t''; t_{critsec} & \longrightarrow^* \\
& t_{critsec}\ . &
\end{array} \tag{B.22}$$

A lock-release performs the following steps:

$$\begin{array}{lll}
|\!\rangle; t = & \varsigma(s{:}c).\lambda().|\!\rangle_{body}; t & \xrightarrow{\tau_r^m} \\
& |\!\rangle_{body}[o/s]; t & \longrightarrow^* \\
& let\ x{:}c = o\ in\ o.x_2 := x; t & \xrightarrow{\tau_w} \\
& let\ x{:}c = o\ in\ o.x_1 := x; t & \xrightarrow{\tau_w} \\
& t\ . &
\end{array} \tag{B.23}$$

Note that a lock release $|\!\rangle$ can never deadlock and that the very last step of $|\!\rangle$, the $\xrightarrow{\tau_w}$, is the first point where another thread has the chance to enter.

By the mutex Lemma B.2.23, the code of $\langle\!|_w^1\ t_{critsec}^1\ |\!\rangle^1$ (with $\xrightarrow{\tau_w}$ as the first step) and $\langle\!|_r^2\ t_{critsec}^2\ |\!\rangle^2$ are executed under mutual exclusion, i.e., without overlap (and vice versa). In particular the shorter $t_w^1 = \langle\!|_w^1\ t_{critsec}^1\ |\!\rangle^1$ and $t_w^2 = \langle\!|_w^2\ t_{critsec}^2\ |\!\rangle^2$ are executed under mutual exclusion. Assume wlog. that t_w^1 precedes t_w^2. This implies (Lemma B.2.23) that also t_w^1 precedes t_r^2, i.e.,

$$\Longrightarrow \xrightarrow{\langle\!|_w^1} \xrightarrow{t^1} \xrightarrow{|\!\rangle^1} \Longrightarrow \xrightarrow{\langle\!|_r^2} \xrightarrow{t^2} \xrightarrow{|\!\rangle^2} \ . \tag{B.24}$$

We now argue that all internal steps of n_2 can be completely ordered after n_1 has left its critical region, i.e., after $|\!\rangle^1$. According to equation (B.22), the reduction from the beginning of $\langle\!|^2$ to $\langle\!|_r^2$ consists of a single τ_r^m-step, replacing the self-parameter by the the actual identity of the object by rule CALL$_i$. By the non-interference Lemma C.2.4, the τ_r^m-step for the method call can be postponed, yielding

$$\Longrightarrow \xrightarrow{\langle\!|^1} \xrightarrow{t^1} \xrightarrow{|\!\rangle^1} \Longrightarrow \xrightarrow{\langle\!|^2} \xrightarrow{t^2} \xrightarrow{|\!\rangle^2} , \tag{B.25}$$

as required. □

Next we generalize the lemma to deal with external labels, as well. At the same time, we generalize the lemma also to deal with mutual exclusion for whole cliques; Lemma B.2.24 dealt with the critical section for one single object, only.

Lemma B.2.25 (Disentangling). *Let $\Xi_0 \vdash C_t$ be given by Definition 5.2.7 and assume $\Xi_0 \vdash C_t \stackrel{r}{\Longrightarrow} \Xi \vdash C \stackrel{a_1 a_2}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$, where the labels a_1 and a_2 are performed by the threads n_1 and n_2. Assume further that the critical sections $(\!|^1\ t^1_{sync}\ |\!)^1$ and $(\!|^2\ t^2_{sync}\ |\!)^2$ belonging a_1 resp. a_2 are performed completely. Then $\Xi \vdash C \stackrel{b_1 b_2}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ by a clean reduction, where $\Xi_0 \vdash r\ a_1\ a_2 \sqsubseteq^{switch}_{\Theta} r\ b_1\ b_2$.*

Proof. First note that in case of an incoming label, the corresponding atomic region is executed after the external step; for an outgoing label, the execution of the critical section precedes the labeled step. By the definition of switching (cf. Table 5.2), either $b_1\ b_2$ equals $a_1\ a_2$ or equals the reversed order $a_2\ a_1$. See also the discussion on page 105.

We show the argument for one single object. The generalization to disentangle the steps of a whole clique is straightforward, using the non-interference Lemma C.2.4, in particular switching the order of steps belonging to different objects.

We show the case where $a_1 = \gamma_1?$ and $a_2 = \gamma_2!$, which corresponds to the switching rule O-OI (which is the rule where the reverse direction is not covered). All other combinations of inputs and outputs work similarly. Projected to thread n_1, the interactions for a_1, resp., for a_2, projected to n_2, look as follows:

$$\stackrel{\gamma_1?}{\Longrightarrow} \underset{(\!|^1 t^1_{sync} |\!)^1}{\Longrightarrow} \quad \text{and} \quad \underset{(\!|^2 t^2_{sync} |\!)^2}{\Longrightarrow} \stackrel{\gamma_2!}{\Longrightarrow}\ , \tag{B.26}$$

where t^1_{sync} is part of the input synchronization code t^i_{sync} and t^2_{sync} of output synchronization t^o_{sync} (cf. Definition B.2.2 and B.2.11). By Lemma B.2.24, the reduction from C to $\acute{C}$ can be reordered such that there is no overlap between the critical sections, i.e., that

$$\underset{(\!|^1 t^1_{sync} |\!)^1\ (\!|^2 t^2_{sync} |\!)^2}{\overset{\gamma_1? \qquad\qquad \gamma_2!}{\Longrightarrow}} \quad \text{or} \quad \underset{(\!|^2 t^2_{sync} |\!)^2\ (\!|^1 t^1_{sync} |\!)^1}{\overset{\gamma_1? \qquad\qquad \gamma_2!}{\Longrightarrow}}\ . \tag{B.27}$$

The second reduction, where the order of the critical sections is opposite of the order of the labels, is not clean. By the non-interference Lemma C.2.4, the atomic, external $\gamma_1?$-step does not interfere with the steps of $(\!|^2\ t^2_{sync}\ |\!)^2$ of thread n_2, and neither $\gamma_2!$ with the steps of $(\!|^1\ t^1_{sync}\ |\!)^1$. Since furthermore by the same lemma, $\gamma_2?$ and $\gamma_2!$ do not interfere with each other, the reduction on the right-hand of (B.27) can be reordered (reversing the switching rule O-OI) into

$$\text{or} \quad \underset{(\!|^2 t^2_{sync} |\!)^2 \qquad (\!|^1 t^1_{sync} |\!)^1}{\overset{\gamma_2! \gamma_1?}{\Longrightarrow}}\ , \tag{B.28}$$

as required. □

The lock grabbing $(\!|$ and the lock release $|\!)$ from Definition B.2.21 assure mutual exclusion on the level of single objects. The implementation, however, needs interference free execution *per clique*, since the data structures of all members of a clique are updated in the synchronization code. With $(\!|\_|\!)$ on object level, the implementation is fairly simple, since again we can ignore liveness properties; basically when failing to get hold of all locks of a clique —another thread might try to collect the locks of the same clique starting the traversal from a different entry point— the algorithm is free to give up.

We need to be careful in one respect: The implementation of Definition B.2.20 and Definition B.2.21 realizes a simple binary lock mechanism but no reentrant locks (cf. also Remark B.2.18). In the implementation we need to refrain therefore from *recursive* traversal schemes of acquiring locks on the object level; otherwise the traversal will block.

Definition B.2.26 (Lock handling (clique level))**.** *Given an object with known objects $self.\Theta$, then the lock handling on clique level is simply defined as*

$$(\!| \triangleq \Theta.(\!| \,, \tag{B.29}$$

i.e., as loop[15] *over all (defined) objects from Θ. Analogously for $|\!)$. In abuse of notation, we will write for the lock-handling on the clique level simply $(\!|$ and $|\!)$.*

Remark B.2.27 (Mutual exclusion and merging)**.** *As the clique structure is dynamic, in particular, component cliques may merge, the code for input synchronization must obtain the lock not just for the clique of the target object of the communication as given by the connectivity* before *the step, but for* all *cliques which are in the process of being merged.* □

To formulate the properties of the lock-handling code, we use the following assertions.

Notation B.2.28 (Lock ownership)**.** *Given a well-typed, fully-connected component as constructed in Definition 5.2.7 and equipped with locks as just described. By writing $\Xi \vdash C : o \hookleftarrow n$ ("in component C, thread n owns the lock of object o") we mean that*

$$C \equiv \nu(\Phi).(C' \parallel o\langle lock = n, \ldots\rangle) \,, \tag{B.30}$$

where x_{lock} is the triple of instance variables as given in Definition B.2.20. When writing $\Xi \vdash C : [o] \hookleftarrow n$ ("in component C, thread n owns the lock of clique *$[o]$") we mean that the assertion of equation (B.30) holds for all objects o' with $\Xi \vdash o \leftrightharpoons o'$.*

B.3 Properties of the synchronization code

In Section B.2 we allowed ourselves a number of "higher-level" data structures and operations to concentrate on the core of the construction. As they are not supported by the core calculus, we describe in the following how to implement them. In most cases, the implementation is straightforward, if a bit tedious. We follow a *top-down* approach, i.e., first we state the relevant lemmas about the synchronization code t^i_{sync} and t^o_{sync} from the definitions from Section B.2.

For synchronization for incoming communication, see Definition B.2.2. The lemma below basically states, that the code preserves (or rather re-establishes) the invariants as expressed by the commitments. Remember that the synchronization code for inputs comes *after* the actual external input step. In particular, the commitments are temporarily violated. The execution shortens the future of the affected clique by the corresponding input label (in case it is an expected one), where the clique may be merged from previously separate ones during the execution of the code, and where new objects may be created in the corresponding communication. In effect, being executed directly after an incoming

[15] An iteration, not a recursion. As always, an upper bound on the number of iterations is statically determined.

communication, the new objects themselves are already instantiated, which is done "automatically" by the semantics. In case a new thread name enters the component in the input —this can happen only in an incoming call— also the thread itself is already present in the component in the pre-condition of the lemma.

We can handle the concurrent framework in the same way as the single-threaded one. "Cleaning up" a multithreaded reduction (cf. Definition C.4.1 on page 259 for clean reduction) allows to treat the synchronization for each interaction without interference of other threads.

Lemma B.3.1 (Synchronization: Input (cf. Definition B.2.2))**.** *Let $a = \nu(\Phi').\gamma?$ be an incoming label with $\Phi' = \Delta', \Sigma', \Theta'$, and where Δ' contains the environment objects transmitted by scope extrusion, Θ' the lazily instantiated component objects, and Σ' potentially a new thread name (in the multithreaded setting). Assume $\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C : \Theta, \Theta', \acute{\Sigma}; E_\Theta = \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C' \parallel n\langle t^i_{sync}(); t\rangle : \Theta, \Theta', \acute{\Sigma}; E_\Theta$. Furthermore*

$$\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C : \acute{\Theta}, \acute{\Sigma}; E_\Theta :: [o] \rhd {}_{[o]}{\downarrow} as' \tag{B.31}$$

for all object cliques $[o]$ according to E_Θ and where $\Theta \vdash o$, and furthermore

$$\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C : \acute{\Theta}, \acute{\Sigma}; E_\Theta :: [o'] \rhd \bot \tag{B.32}$$

for all object cliques $[o']$ with $\Theta' \vdash o'$ (see Definition 3.3.22). Then

$$\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta \Longrightarrow \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C'' \parallel n\langle t\rangle : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta = \acute{\Xi} \vdash \acute{C} \tag{B.33}$$

and

$$\acute{\Xi} \vdash \acute{C} :: s' \,. \tag{B.34}$$

In the multithreaded setting, we assume further that the lock of the clique is free.

Proof. Note that in the specification of the pre-condition, the newly created component instances are already present, as asserted by $\acute{\Theta} = \Theta, \Theta'$. They are, however, not yet appropriately connected and also they do not yet have the future behavior initialized appropriately. This is asserted by the connectivity context E_Θ (as opposed to $\acute{E}_\Theta$) and by equation (B.32) for the future of the lazily instantiated, new objects.

The code of t^i_{sync} is given in equation (B.3) and does the following steps: Storing the *label*, *initialization*, *stepping forward*, and *broadcasting* (plus lock-handling in the multithreaded setting). Let us abbreviate $t^i_{sync}; t$ as t_0, the thread at the control point after initialization t_1, and after returning from the invocation of $step^i$ as t_2. Finally, t_3 corresponds to t, the remaining code after synchronization.

After obtaining the lock (in the multithreaded setting) and storing the label a in the instance state, $initialize$ is invoked on *all* component objects (including potentially self)[16] which mentioned in a. Iterated application of the initialization Lemma B.4.6) yields that for all freshly instantiated objects $\Theta' \vdash o'$ we are given $\acute{\Xi} \vdash C_1 \parallel n\langle t_1\rangle :: [o'] \rhd {}_{o'}{\downarrow} t$ (part (2) of the Lemma), where t is the given trace. Since by assumption, it is the *first* appearance of o' in

[16]Note that it is possible that the target of the communication, which corresponds to the object that executes the synchronization code, is lazily instantiated itself in the communication and hence is uninitialized at that point immediately after the communication.

the trace, this implies $\acute{\Xi} \vdash C_1 \parallel n\langle t_1\rangle :: o' \rhd {}_{[o']}\!\downarrow as'$ and furthermore $\acute{\Xi} \vdash C_1 \parallel n\langle t_1\rangle :: [o'] \rhd {}_{[o']}\!\downarrow as'$. By part (1) of the same lemma, the initialization method leaves all previously existing cliques $[o]$ with $\Theta \vdash o$ unchanged, i.e., $\acute{\Xi} \vdash C_1 \parallel n\langle t_1\rangle :: [o] \rhd {}_{[o]}\!\downarrow as'$ as in the pre-configuration.

So, considering both new component objects and old ones, we have for all objects o from $\acute{\Theta} = \Theta + \Theta'$

$$\acute{\Xi} \vdash C_1 \parallel n\langle t_1\rangle :: [o] \rhd {}_{[o]}\!\downarrow as' \ ,$$

after executing *initialize*, and where $[o]$ are the cliques according to E_Θ.[17]

This means, the pre-condition of the step Lemma B.4.7 for input is given. Thus, $\acute{\Xi} \vdash C_1 \parallel n\langle t_1\rangle \Longrightarrow \acute{\Xi} \vdash C_2 \parallel n\langle t_2\rangle$ such that $\acute{\Xi} \vdash C_2 \parallel n\langle t_2\rangle :: o_r \rhd {}_{[o]}\!\downarrow s'$, where o_r is the receiver of the input action a. Finally, by the *broadcast* Lemma B.4.9, $\acute{\Xi} \vdash C_2 \parallel n\langle t_2\rangle \Longrightarrow \acute{\Xi} \vdash C_3 \parallel n\langle t\rangle$ with $\acute{\Xi} \vdash C_3 \parallel n\langle t\rangle :: [o] \rhd {}_{[o]}\!\downarrow s'$ for all component cliques $[o]$ according to $\acute{\Xi}$, which means $\acute{\Xi} \vdash C_3 \parallel n\langle t\rangle :: s'$ (cf. Definition 3.3.22), as required. □

For output, the statement of the pre- and post-assertions is simpler than for input, since no merging of cliques is involved. So the following lemma expresses that the output synchronization code from Definition B.2.11 does the expected job, i.e., it creates the required internal objects mentioned in Θ', initiates the objects to be lazily instantiated in the external step to follow from Δ', and shortens the future behavior.

Lemma B.3.2 (Synchronization: Output (cf. Definition B.2.11))**.** *Let $a = \nu(\Phi').\gamma!$ be an outgoing label with $\Phi' = (\Delta', \Sigma', \Theta')$ where Δ' be the lazily instantiated environment objects and Δ' be the identities of component objects transmitted by scope extrusion. Assume $\Xi \vdash C$ with $C = C' \parallel n\langle t^o_{sync}(); t\rangle$. If $\Xi \vdash C :: a\ s$, then $\Xi \vdash C \stackrel{a}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ and $\acute{\Xi} \vdash \acute{C} :: s$, where $\acute{C} = C'' \parallel n\langle t'\rangle$ where t' either blocked before performing input synchronization, or stopped. In the multithreaded setting, we assume further that the lock of the clique is free.*

Proof. The code for output synchronization is given as $(|; step^o()$ in equation (B.11) in Definition B.2.11 (see also Definition B.2.10).

$$\begin{array}{lll} C' \parallel n\langle t^o_{sync}; t\rangle = & C' \parallel n\langle (|; step^o; t\rangle & \Longrightarrow \\ & C' \parallel n\langle step^o; t\rangle & = \\ & C' \parallel n\langle let\ (\sigma, \check{a}\_) \in scripts\ in\ t_1\rangle\ . & \end{array}$$

According to the assumption $\Xi \vdash C :: as$ for the pre-configuration (cf. Definition 3.3.22). Assume first that the code is executed in a clique known to the outside, say $[o]_{/\equiv}$ (or $[o]$ for short) with $\Theta \vdash o$. By equation (3.49) and (3.50), there exists at least one open future $(\sigma, \check{a}\_)$ in *scripts*. Thus, the reduction continues:

$$\begin{array}{lll} \ldots & C' \parallel n\langle let\ \sigma' = create(\check{a})\ in\ t_2\rangle & \Longrightarrow \\ & C' \parallel \nu(\Phi').(C(\Phi') \parallel n\langle t'_2\rangle) & \end{array}$$

where $C(\Theta')$ equals $o_1[c_1, F_1], \ldots, o_k[c_k, F_k]$, i.e., it contains the freshly created component objects in their initial state, and $\sigma' = [\check{o}_1 \mapsto o_1, \ldots, \check{o}_m \mapsto o_m]$, when $\check{o}_i$ are the roles mentioned bound in the label $\check{a}$.

[17] And not $\acute{E}_\Theta$, which means, still reflecting the clique structure before the merge and especially with the lazily instantiated new objects forming singleton cliques.

Let now $(\sigma_i, \check{a}_i\check{s}_i)$ be an arbitrary element of type script from the set $scripts$. Note that there exists at least one such element, namely the $(\sigma, \check{a}\check{s})$ mentioned above. If $\check{a}_i$ is a renamed variant of $\check{a}$ wrt. the *roles* occurring bound in $\check{a}$, resp., in $\check{a}_i$, then the association $\sigma' \circ \pi(\check{a}, \check{a}_i)$ is defined and of the form $[\ldots \check{o}_{\pi(i)} \mapsto o_i, \ldots]$.[18] Let abbreviate the association by σ'_i. Note that either $dom(\sigma') = dom(\sigma'_i)$ (exactly when $\check{a}_i = \check{a}$ and π corresponds to the identity) or $dom(\sigma) \cap dom(\sigma'_i)$ is empty (in all other cases). Note further that $dom(\sigma'_i) \cap dom(\sigma_i)$ is empty, since σ'_i contains the bindings exactly for those roles which are *new* wrt. the ith script, and which are mentioned as new in the binding part of $\check{a}_i$. With the domains of σ'_i and σ_i disjoint, $\sigma'_i \oplus \sigma_i$ is defined (cf. Definition B.2.6, equation (B.8)). Thus, for $\acute{\sigma}_i$, the association for the ith script after evaluating $\check{a}_i$, we have

$$o.script_i = (\acute{\sigma}_i, \check{s}_i) \quad \text{such that} \quad {}_{[o]}\!\downarrow s \lesssim \check{s}_i\acute{\sigma}_i \ , \tag{B.35}$$

where o is the object which "executes" the code (cf. also Definition 3.3.22).

If $\check{a}$ is *not* a renamed variant of $\check{a}_i$, the corresponding script $script_i$ is deleted from the set $scripts$.

This means, after executing the loop $\forall(\sigma_i, \check{a}_i, \check{s}_i) \in self.scripts$ in the code of $step^o$ (cf. equation (B.10)), for all scripts remaining in the set $self.scripts$, equation (B.35) holds. Note that there survives at least one script after the loop, satisfying (B.35), which corresponds to the script where $\check{a}_i = \check{a}$.

Note that the pick of $(\sigma, \check{a}\check{s})$ in the first line of $step^o$ represents the exact point where *non-determinism* in the reaction of C_t occurs. In the sequential, deterministic setting, we cannot non-deterministically pick one element of $self.scripts$. However, the definition of a deterministic trace (cf. Definition 3.1.10) assures that *all* $\check{a}_i$ in the loop are renamings of each other. This means, *all* scripts "survive" the loop starting in line 3 of $step^o$! In both the deterministic and the non-deterministic case, for all surviving scripts we have $\check{a}_i\acute{\sigma}_i = a$, since we assumed for the pre-configuration ${}_{[o]}\!\downarrow as \lesssim (\check{a}_i\check{s}_i)\sigma_i$ and since *all* roles mentioned ν-bound in $\check{a}$ are instantiated $create(\check{a})$.

Given (B.35) for all scripts, executing $self.broadcast(\Sigma)$ establishes $o'.script_i = (\acute{\sigma}_i, \check{s}_i)$ for all objects in o's clique, including the newly created objects, i.e., it establishes $[o]_{/\acute{\Xi}}.script_i = (\acute{\sigma}_i, \check{s}_i)$, where $\acute{\Xi}$ is the context updated by the label a. As mentioned, the label a of the trace corresponds to the one in the code of $step^o$, handed over to $self.interpret$ (cf. Definition B.2.12).

In case of a return, $interpret$ gives back the corresponding value and the reduction continues as follows (not that the value v must be an object reference o', and that Φ' either is empty or contains the binding for the object reference o'):

$$\begin{array}{lll} \ldots & \Xi \vdash \nu(\Phi'').(C' \parallel \nu(\Phi').(C(\Phi') \parallel n\langle let\ x{:}T = (\!|\,;v)\ in\ o\ returns\ x\ to\ o_r; t_3\rangle)) & \Longrightarrow \\ & \Xi \vdash \nu(\Phi'').(C' \parallel \nu(\Phi').(C(\Phi') \parallel n\langle o\ returns\ v\ to\ o_r; t_3\rangle)) & \overset{a}{\longrightarrow} \\ & \acute{\Xi} \vdash \nu(\Phi'').(C' \parallel C(\Phi') \parallel n\langle t_3\rangle)\ . & \end{array}$$

[18] The $\pi(\check{a}, \check{a}_i)$ is meant as renaming *from* the roles of $\check{a}$ *to* the roles of $\check{a}_i$.

In case of a call, *interpret* issues the call; the reduction continues as follows:

$$
\begin{array}{l}
\ldots \\
\quad \Xi \vdash \nu(\Phi'').(C' \parallel \nu(\Phi').(C(\Phi') \parallel n\langle\ let\ x{:}T \quad = (\!|;\ \ let\ y{:}T' \quad = o_r.l(v) \qquad\qquad \Longrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad in \qquad t^i_{sync}(return, y); t^o_{sync}()) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad in \quad o\ returns\ x\ to\ o_r; t_3\rangle)) \\
\quad \Xi \vdash \nu(\Phi'').(C' \parallel \nu(\Phi').(C(\Phi') \parallel n\langle\ let\ y{:}T' \quad = o_r.l(v) \qquad\qquad \xrightarrow{a} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad in \quad let\ x{:}T' \quad = t^i_{sync}(return, y); t^o_{sync}()) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad in \quad o\ returns\ x\ to\ o_r; t_3\rangle)) \\
\quad \Xi \vdash \nu(\Phi'').(C' \parallel \nu(\Phi').(C(\Phi') \parallel n\langle\ let\ y{:}T' \quad = o\ blocks\ for\ o_r \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad in \quad let\ x{:}T' \quad = t^i_{sync}(y); t^o_{sync}()) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad in \quad o\ returns\ x\ to\ o_r; t_3\rangle)\ .
\end{array}
$$

□

Example B.3.3 (Input and output synchronization)**.** *Let us illustrate the working of input and output synchronization. Consider the following (balanced) trace t,*

$$\nu(o_1).n\langle call\ o_1.l_1()\rangle?\ \nu(o_2).n\langle call\ o_2.l_2()\rangle!\ n\langle return()\rangle?\ n\langle return()\rangle!\ , \qquad \text{(B.36)}$$

abbreviated as $a_1\ a_2\ a_3\ a_4$ and where we omit typing information for simplicity. Let further C_0 for C_t according to Definition 3.3.20 resp. 5.2.7. Then the execution of the first two actions of the trace, the incoming followed by the outgoing call, works as follows. The reduction sequence is simplified in that we omit the types, and that in the reduction, we keep the $let\ x = \ldots\ in\ o_1\ returns\ x\ to \odot$ syntactically at the outermost level and reduce the thread inside*. In full detail, there are additionally* LET*-steps to move the active redex to the front.*

$$
\begin{array}{lll}
\Xi_0 \vdash C_0 & \xrightarrow{a_1} & \\
\Xi_1 \vdash C_0 \parallel o_1[c_1, F] \parallel n\langle let\ x = o_1.l_1()\ in\ o_1\ returns\ x\ to \odot\rangle & \xrightarrow{\tau} & \\
\Xi_1 \vdash C_0 \parallel o_1[c_1, F] \parallel n\langle let\ x = M_1.l_1(o_1)()\ in\ o_1\ returns\ x\ to \odot\rangle & = & \\
\Xi_1 \vdash C_1 \parallel n\langle let\ x = t^i_{sync}(l); t^o_{sync}()\ in\ o_1\ returns\ x\ to \odot\rangle & \Longrightarrow & \textit{(init., pick repres.)} \\
\Xi_1 \vdash C'_1 \parallel n\langle let\ x = o_1.step^i(o_1, a_1)\!|); t^o_{sync}()\ in\ o_1\ returns\ x\ to \odot\rangle & \Longrightarrow & \textit{(shorten future)} \\
\Xi_1 \vdash C''_1 \parallel n\langle let\ x = t^o_{sync}()\ in\ o_1\ returns\ x\ to \odot\rangle & \Longrightarrow & \\
\Xi_1 \vdash C'_1 \parallel n\langle let\ x = o_1.step^o()\ in\ o_1\ returns\ x\ to \odot\rangle & = & \\
\Xi_1 \vdash C'_1 \parallel n\langle\ let \quad x = (\ \ let \quad (\sigma, \check{a}\check{s}) \in o_1.scripts\ in \ldots) & \Longrightarrow & \textit{(det. next action)} \\
\qquad\qquad in \quad o_1\ returns\ x\ to \odot\rangle & & \\
\Xi_1 \vdash C'_1 \parallel n\langle\ let \quad x = (\ \ let \quad \sigma' = create(\check{a}_2)\ in \ldots) & \Longrightarrow & \textit{(create } o_2\textit{)} \\
\qquad\qquad in \quad o_1\ returns\ x\ to \odot\rangle & & \\
\Xi_1 \vdash C'_1 \parallel n\langle\ let \quad x = (\ \ let \quad \acute{\sigma} = \sigma \oplus \sigma'\ in \ldots) & \Longrightarrow & \textit{(extend assoc.)} \\
\qquad\qquad in \quad o_1\ returns\ x\ to \odot\rangle & & \\
\Xi_1 \vdash C'_1 \parallel n\langle\ let \quad x = (\ \ scripts := (\acute{\sigma}, \check{s}); \ldots) & \Longrightarrow & \textit{(update future)} \\
\qquad\qquad in \quad o_1\ returns\ x\ to \odot\rangle & & \\
\Xi_1 \vdash C'_1 \parallel n\langle\ let \quad x = (\ \ let \quad a = \check{a}_3(\sigma \oplus \sigma')\ in\ o_1.interpret(a)) & \Longrightarrow & \textit{(det. next action)} \\
\qquad\qquad in \quad o_1\ returns\ x\ to \odot\rangle & & \\
\Xi_1 \vdash C'_1 \parallel n\langle\ let \quad x = (\ \ o_1.interpret(a_2)) & \Longrightarrow & \textit{(exec. } a_2\textit{)} \\
\qquad\qquad in \quad o_1\ returns\ x\ to \odot\rangle & & \\
\Xi_1 \vdash C'_1 \parallel n\langle\ let \quad x = (\ \ (\!|); let\ y = o_2.l_2()\ in\ t^i_{sync}(return, y); t^o_{sync}()) & \overset{a_2}{\Longrightarrow} & \textit{(call)} \\
\qquad\qquad in \quad o_1\ returns\ x\ to \odot\rangle & & \\
\Xi_3 \vdash C_3 \parallel n\langle\ let \quad x = (\ \ let\ y = o_1\ blocks\ for\ o_2\ in\ t^i_{sync}(return, y); t^o_{sync}()) & & \\
\qquad\qquad in \quad o_1\ returns\ x\ to \odot\rangle & &
\end{array}
$$

In the example, there is only one component object involved, namely o_1. Consequently, the scripts data structure only contains one single static future which is being worked

off. After the two calls, the thread n is blocked and waiting for return and there remain still the two calls $\check{a}_3\check{a}_4$ to be executed in the script. Continuing with the incoming a_3, the input synchronization would shorten the future to $\check{a}_4$ (without creating new objects, assigning new roles, or merging), and the trailing output synchronization will take care that the return a_4 happens by passing over the return value to the let-bound variable x and shortening the future to the empty sequence. □

B.4 Data structures and operations

In this section we show in a more detail the implementation of the data structures and the code we used in Section B.2. Furthermore we prove the lemmas corresponding to the code which we used in Section B.3. In particular, we show how to encode the references occurring in the given trace in C_t.

B.4.1 Objects and connectivity

A crucial point concerns the data *dynamically* created during the run, in particular the identities. Created freshly, their values cannot be fixed at compile-time. As each legal trace is finite, the values can be represented in a statically determined number of instance variables. We furthermore assume, that the ν-bound identities in the trace are alphabetically renamed so that two different identities do not literally carry the same name.

The instance variables $\check{o}$ are needed to store the references once they are created or received from outside. Analogously $\check{n}$ to store thread names, in the multithreaded setting and at the beginning of the run, the values are not yet available. Since the language does not contain fully functional native nil-references, we need to provide them ourselves.

Remark B.4.1 (Undefined)**.** *To fill in some static "value" in the fields of a class, we used $\bot_c$ as notation for an instance variable of type c yet undefined. Note that $\bot_c$ is* not *a value of the calculus, it rather denotes the* absence *of a value. In particular, $\bot_c$ cannot be "copied" into another instance variable and it cannot be handed over as argument. Operationally, it behaves as $\varsigma(s{:}c).\lambda().\,stop$. This design choice is taken to avoid to define $\bot_c = \bot_c$, or $\bot_c = o \ldots$.*

As we decided that trying to access the undefined value leads to an error (represented as just deadlock), we need the additional operation to test for being undefined. □

Definition B.4.2 (Object identities)**.** *Given a legal trace t justified by $\Xi \vdash t : trace$. For each object reference o of type c occurring in the trace, $\check{o}$ denotes an instance variable of type c contained in all classes of Θ. Initially, the value of the instance variable representing one instance of type c is $\bot_c$.*

$$x_{flag} := false \quad \triangleq \quad \varsigma(s{:}c).\ \begin{array}{ll} let & x, y{:}c = new\, c, new\, c \\ in & self.x_1 = x; self.x_2 := y\ . \end{array} \tag{B.37}$$

We refer to the pair of instance variable and the flag as $\check{o}$ (the static analog of o), and to its type as object.

Definition B.4.3 (Operations)**.** *We denote by $\bot_c$ a "pair" of stop and boolean value $false$, which we also refer to as nil value. We write $\check{o} := o'$ for allocating a pair, i.e. for*

$$\check{o} := o' \quad \triangleq \quad x_o := o; x_o^{isallocated} := true \ .$$

Furthermore we write for the comparison of the instance variable (pair) with a nil-pointer:

$$\check{o} = \bot_c \quad \triangleq \quad \mathsf{if}\ isallocated(\check{o})\ \mathsf{then}\ false\ \mathsf{else}\ true$$

where $isallocated$ refers to the boolean allocation flag.

The variable $\check{o}$ (or rather the pair) is set to a non-nil value exactly once; afterwards it is accessed in read-only manner, only. The implementation of the above operations and their "soundness" wrt. the intended meaning is obvious.

To represent its connectivity, each object maintains the set of all identities it knows, or rather all references it as ever heard of. We refer to the set of objects as $self.\Theta$. This set corresponds to clique of objects of the instance at hand, i.e., all objects acquainted to the current one according to E_Θ.

Definition B.4.4 (Set of objects)**.** *We refer to a collection of* allocated *objects of type* object *as of type* set of object*. In particular, we refer to the ensemble of instance variables $\check{o}$ of type* object*, which are allocated, by $self.\Theta$, and to its type as* set of object*.*

Definition B.4.5 (Checking for containment)**.** *Checking whether an object reference is already known is done as follows:*

$$\begin{array}{rcl} o \in self.\Theta & \triangleq & \mathsf{if}\ o = \check{o}_1\ \mathsf{then}\ true; \\ & & \ldots \\ & & \mathsf{if}\ o = \check{o}_n\ \mathsf{then}\ true\ \mathsf{else}\ false\ . \end{array}$$

$\check{o}_1, \ldots, \check{o}_n$ are all instance variables introduced in Definition B.4.2.

B.4.2 Labels and scripts

A core data structure for the observer C_t is the list containing t (respectively projections thereof), which we called the *scripts* that need to be realized. To represent it we used the data types of lists and of sets (cf. Definition B.2.1). Indeed, the definition seems even to use set and list as type constructors in a parametric or generic way. A closer look at C_t and its behavior shows, however, that we are in a more comfortable position. First of all, we do not need *dynamic* data structures. The mentioned structures are needed to represent (and together with appropriate method code to enforce) a given trace t, as part of the completeness proof. With this trace finite and given, the encoding need not provide a general implementation of sets or lists, i.e., an implementation of the types set and list used in Definition B.2.1; it suffices to implement the particular t in the $script$-variable, and the given ensemble of various behaviors of instances of a class in the $scripts$-variable.

Next we encode *traces,* i.e., sequences or lists of labels (cf. again Definition B.2.1). The form of the labels as used in the semantics is given in Table 2.8 resp. 4.6, more precisely the version of the labels augmented with a caller identity. Again, it is the finiteness of the given trace, mentioning only a finite number of references and values and method names, which allows to encode all

required labels in a statically determined arrangement of instance variables. Given a label a, we denote by $\check{a}$ the instance variable (or rather the collection of instance variables) used to store the label a. A label a needs to be stored in a structured way, since we need to refer to the constituent parts, in particular, the object identities, for comparison. Furthermore, we assume a status of being undefined, written again $\perp$, implemented the same way as for object references.

B.4.3 Synchronization code

Next we describe the implementation of the algorithms operating on the data. On an abstract level, we have made use of the properties of the algorithms already in Section B.2 and Section B.3.

The next lemma characterizes the behavior initialization code (cf. Definition B.2.4 and also Example 3.3.14), which is part of the input (but not for output) synchronization. Basically, the initialization establishes the invariants for a freshly created component object, treating it as a (perhaps only momentarily) isolated clique of its own. We start with the lemma that deals with one single component object. The code is given in Definition B.2.4. Note that we must assume that the object o on which the initialization is performed is a component object, otherwise the call would be visible at the interface or the program would be ill-typed, since in general, the initialization method is not offered by objects at the interface.

Lemma B.4.6 (Initialization)**.** *Let $\Xi_0 \vdash C_t$ be given as usual. Assume $\acute{\Xi} \vdash C = \acute{\Xi} \vdash C' \parallel n\langle o.initialize(); t\rangle$ with $\acute{\Theta} \vdash o$. Furthermore assume that equation (B.31) (from the input synchronization lemma B.3.1 holds.*[19]*Then $\acute{\Xi} \vdash C \Longrightarrow \acute{\Xi} \vdash C'' \parallel n\langle t\rangle$ s.t.:*

1. *If $\acute{\Xi} \vdash C :: [o] \rhd w$, then $\acute{\Xi} \vdash C'' \parallel n\langle t\rangle :: o \rhd w$ (where $w \neq \perp$ and $C'' = C'$).*
2. *If $\acute{\Xi} \vdash C :: [o] \rhd \perp$, then $\acute{\Xi} \vdash C'' \parallel n\langle t\rangle :: o \rhd {}_{[o]}\!\downarrow t$.*

Proof. For the code of the $initialize$-method, see Definition B.2.4. There are two cases to distinguish. In case (1), when the object is already initialized, we are given $o.scripts \neq \perp$ (cf. equations (3.49) and (3.50)) and the claim follows directly from the code; the execution of the initialization code has no effect.

In case (2), we are given $\acute{\Xi} \vdash C :: [o] \rhd \perp$, i.e., $o.scripts = \perp$ (cf. equation (3.51)). Since the object o is new, it is not yet connected to any other object and ${}_{[o]}\!\downarrow t = {}_{o}\!\downarrow t$. Let $\check{t}_{\check{o}}$ abbreviate ${}_{\check{o}}\!\downarrow \check{t}$. By construction, the instance variable $scripts$ of class c of o contains (potentially among other futures) the pair,

$$(\sigma_\perp, \check{t}_{\check{o}})$$

which is the static representation of o's behavior in the given t. According to the code of $initialize$ in equation (B.4), $\sigma_{\check{o}}$ is set to $\sigma_{\check{o}}[\check{o} \mapsto o]$, which means that ${}_{o}\!\downarrow t \lesssim \check{t}_{\check{o}}\sigma_{\check{o}} = \check{t}_{\check{o}}[\check{o} \mapsto o]$ after executing $initialize$. Hence $\acute{\Xi} \vdash C' \parallel n\langle t\rangle :: o \rhd {}_{o}\!\downarrow t$. Therefore, for all cliques $[o]$ we have $\acute{\Xi} \vdash C' \parallel n\langle t\rangle :: [o] \rhd {}_{o}\!\downarrow t$, as required. □

The next lemma specifies the behavior of the step-method, used during input synchronization.

[19] I thought, I might need additionally that the conditions of Lemma B.3.1 hold, in particular

Lemma B.4.7 (Input step)**.** *Let* $\acute{\Xi} \vdash C$ *abbreviate* $\acute{\Xi} \vdash C' \parallel n\langle o.step^i(a, \vec{o}); t\rangle$*, where* $\vec{o}$ *is a set of object references containing one representative for each component clique. Furthermore*

$$\acute{\Xi} \vdash C :: [o] \rhd {}_{[o]}\!\downarrow as' \tag{B.38}$$

for each component clique $[o]$*. Then* $\acute{\Xi} \vdash C \Longrightarrow \acute{\Xi} \vdash C'' \parallel n\langle t\rangle$ *such that*

$$\acute{\Xi} \vdash C'' \parallel n\langle t\rangle :: o \rhd {}_{[o]}\!\downarrow s' \ .$$

Proof. The code of the $step^i$-method is shown in equation (B.9) in Definition B.2.8. Let $s = as'$. By assumption, $\acute{\Xi} \vdash \acute{C} :: o' \rhd {}_{[o']}\!\downarrow s$ for all component objects o', which means (Definition 3.3.22) there exists $(\sigma', \check{s}_{o'}) \in o'.scripts$ where

$${}_{[o']}\!\downarrow s \lesssim \check{s}_{o'}\sigma' \ . \tag{B.39}$$

If o' (resp. its clique) is involved in the communication, i.e., $[o']$ is the target of the communication and/or being merged in the current step, the current future $\check{s}_{o'}$ of o' is of the form $\check{a}_i\check{s}_i$, as ${}_{[o']}\!\downarrow as'$ starts with (the projection of) the label a (cf. rule P-IN$_2$ from Table 3.2). Let $a_{o'}$ denote the first label of the projection ${}_{[o']}\!\downarrow as'$. By definition of projection, $a_{o_1} \neq a_{o_2}$ for two different component cliques o_1 and o_2, in case a merges the two: The difference concerns the ν-binders, as the local projection adds ν-binders to names which are *locally* new.

Now consider a[20] combination $\sigma_1, \ldots \sigma_k$ of associations from the involved component cliques $[o_1], \ldots, [o_k]$, such that for each σ_i, equation (B.39) holds. In particular, we have for the next action a,

$${}_{[o_i]}\!\downarrow a \lesssim \check{a}_{o_i}\sigma_i \ . \tag{B.40}$$

We first need to argue that the combination $\bigoplus \sigma_i$ (which is contained in the result of $collectroles$, cf. equation (B.7)) is defined. Note that code of $collectroles$ in line 3 of equation (B.9) combines all currently open associations, one σ'' from one of the still open futures from each component clique. Each participating component has, as consequence of the assumption (B.38), at least one still open future; however, not all combinations $\sigma_i'' \oplus \sigma_j''$ from the open scripts of two objects o_i and o_j are defined. The mentioned assumption guarantees, that there exists such a valid combination.

As $\oplus$ is associative and commutative, we concentrate on the combination of the associations of *two* cliques, say $[o_1]$ and $[o_2]$. By the invariant of Lemma B.4.11, the domains of σ_1 and σ_2 are non-empty and furthermore that $dom(\sigma_1) \cap dom(\sigma_2) = \emptyset$ or $dom(\sigma_1) \subseteq dom(\sigma_2)$ or vice versa. Furthermore, since σ_1 and σ_2 belong to two different component cliques, i.e., are taken from $o_1.scripts$ and $o_2.scripts$ where o_1 and o_2 belong to two different cliques, $ran(\sigma_1) \cap ran(\sigma_2) = \emptyset$. Finally, since a merges the two cliques, a mentions an object from $[o_1]$ as well as from $[o_2]$, i.e., $\lfloor \check{a}_1 \rfloor = \lfloor \check{a_2} \rfloor = \lfloor \check{a} \rfloor$ contains a variable from $dom(\sigma_1)$ as well as from $dom(\sigma_2)$ (because of equation (B.40)), where $\lfloor \check{a} \rfloor$ is the core of label $\check{a}$. Hence by Lemma B.4.15, $\sigma = \sigma_1 \oplus \sigma_2$ is defined and furthermore $\lfloor \check{a} \rfloor \sigma \lesssim \lfloor \check{a} \rfloor$ and ${}_{[o]}\!\downarrow s' \lesssim \check{s}'\sigma$. Since for each all free roles x of $\lfloor \check{a} \rfloor$, $x \in dom(\sigma_1)$ or $x \in dom(\sigma_2)$, $\check{a}\sigma = \check{a}$, as required by the code of $collectroles$ from the 4th line of equation (B.7). □

[20] There may indeed be more than one in case the overall static script $\check{t}$ contains different versions of the same behavior due to replay.

The broadcast method, used in the code for performing the output synchronization as well as for input (cf. equation (B.10) and (B.3)), is rather straightforward. It simply updates the $scripts$ data structure of all component objects its current clique with the value the current objects has stored for itself. Remember that by $self.\Theta$ we refer to the component references that the object under consideration is currently aware of.

Definition B.4.8 (Broadcast)**.** *Each component class is equipped with a private method* $broadcast$ *of type* scripts $\rightarrow$ Unit *given as follows*

$$broadcast(scripts : \mathsf{scripts}) \triangleq self.(\acute{\Theta}.scripts) := scripts\ . \tag{B.41}$$

Lemma B.4.9 (Broadcast)**.** *Assume* $\Xi \vdash C = \Xi \vdash C' \parallel n\langle\vec{o}.broadcast(\Sigma); t\rangle$*, where* $\vec{o}$ *contains one representative of each component clique according to* Ξ*, and* Σ *the value of the* $scripts$ *data structure of the executing object* o*. If* $\Xi \vdash C :: o \rhd s$*, then* $\Xi \vdash C'' \Longrightarrow\!\!\!\!\!\parallel\ n\langle t\rangle$ *and* $\Xi \vdash \acute{C} :: [o] \rhd s$*.*

Proof. Straightforward: the broadcast-method simply copies $self.scripts$ to all objects to the current clique of the receiving method. □

The following lemma shows a central invariant of the implementation. The $scripts$-variable for each clique contains sets of pairs

$$(\sigma_i, \check{s}_i)\ ,$$

where $\check{s}$ represents one open future and σ the abstraction of the witnessed past, in the form of a variable-reference association. The still open future $\check{s}_i$ is part of the globally given static trace (by way of projection) $\check{t}$ and determines exactly the state up-to which the predefined script has been "played". The past interaction, in the static representation, is kept as *domain* of the corresponding association σ_i. The invariant states, that for each still open script, the past σ_i and the future s_i "fit together" in that they correspond to a state in the given behavior t to be realized. So the invariant of equation (B.42) corresponds to the property of equation (A.31) for dynamic traces.

Lemma B.4.10. *Let* t *be the given, legal trace,* $\check{t}$ *the static analog, and* $\Xi_0 \vdash C_t$ *the programmed component, as before. Assume* $\Xi_0 \vdash C_t \stackrel{r}{\Longrightarrow} \Xi \vdash C$*. Then for all component cliques* $[o]$ *according to* Ξ*, we have that for all elements* $(\sigma_i, \check{s}_i)$ *of* $[o].scripts$

$$dom(\sigma_i) = na\check{m}es(\check{t} - \check{s}_i)\ . \tag{B.42}$$

Proof. Straightforward, by induction on the number of reduction steps. The data structures are changed in the synchronization code, the core is to show, that t^i_{sync} and t^o_{sync} maintain the invariant.

In the base case, in the initial configuration $\Xi_0 \vdash C_t$, the invariant trivially holds, as there are no component cliques. In case of *output*, we must show that that t^o_{sync}, in particular $step^o$ (cf. Definition B.2.11 and B.2.10) preserve the invariant. Before the step, the scripts are of the form $(\sigma_i, \check{a}_i\check{s}'_i)$. According to line 2 of equation (B.10), $dom(\sigma')$ contains the roles occurring bound in $\check{a}$ (filled in by the $create$-operation). Let σ'_i denote the additional bindings added to σ_i in those roles mentioned bound in $\check{a}_i$ from the loop of line 3 — 5 of $step^o$. Since the permutation $\pi(\check{a}, \check{a}_i)$ is defined exactly if $\check{a}_i$ is a "renaming" of the bound

roles of $\check{a}$, $dom(\sigma'_i)$ contains exactly the roles mentioned bound in $\check{a}_i$. Thus, the extension $\acute{\sigma}_i = \sigma_i \oplus \sigma'_i$ maintains the invariant.

For input, cf. Definition B.2.2 and especially B.2.8 for $step^i$. The update of scripts (locally for one object) is done by the assignment $self.scripts_i := (\sigma_i^j, \check{s}_i$ in line 5 of equation (B.9). For the association σ_i^j, we have the equation $a = \check{a}_i\sigma_i^j$, i.e., σ_i^j contains *at least* the bindings for the roles mentioned new in $\check{a}_i$. For all cliques participating in the merged, the invariant holds before the step, i.e., for all σ'' summed up in $collectroles$ of equation (B.7), there is $dom(\sigma'') = na\check{m}es(\check{t} - \check{a}_i^j\check{s}_i^j)$, where $\check{a}_i^j$ is the local version of $\check{a}$. Therefore, $collectroles$ gives back a set of associations where for all σ_i^j in that set s.t. $dom(\sigma_i^j) = \bigcup_k dom(\sigma_k)$, where k ranges over all cliques being merged, and were for $dom(\sigma_k)$ the induction hypothesis applies, i.e., $dom(\sigma_k) = na\check{m}es(\check{t} - \check{b}\check{s}')$, where $\check{b}\check{s}'$ is the open future corresponding to σ_k. Note that $collectroles$ may give back associations σ_k, whose corresponding next step $\check{b}$ does not match with the actual label a to process. In line 4 of equation (B.9), it is checked that $\check{a}_o\sigma_k = a$ (σ_k is called σ_i^j in the actual code of $step^i$), which means $dom(\sigma_k) \supseteq na\check{m}es(\check{a}_i)$, from which the result follows. □

Lemma B.4.11. *Let t be a legal trace and $\Xi_0 \vdash C_t$ given as in Definition 3.3.20. Assume $\Xi_0 \vdash C_t \overset{r}{\Longrightarrow} \Xi \vdash C$. Then for all*

$$dom(\sigma_1) \cap dom(\sigma_2) = \emptyset \text{ or } dom(\sigma_1) \subseteq dom(\sigma_2) \text{ or } dom(\sigma_1) \supseteq dom(\sigma_2) \ .$$

Proof. By Lemma B.4.10 on the previous page, for all component cliques $[o]$ and for all pairs $(\sigma_i, \check{s}_i)$ in $\Xi \vdash C$, that $dom(\sigma_i) = na\check{m}es(\check{t} - \check{s}_i)$ where $na\check{m}es(\check{t} - \check{s}_i)$ refers to the instances variables not object identities in $\check{t} - \check{s}_i$ (cf. also Definition 3.1.5). Translated into the original traces t and s_i, $names(t - s_i)$ corresponds to a component clique, from which the result follows (cf. Lemma A.3.9). □

For the next lemma, we use the following notations. The static representation $\check{t}$ incorporated in C_t consists of a finite set of linear traces, one for each name of a component object mentioned in t (cf. the value of $init$ from Definition 3.3.20). We abbreviate the future projection ${}_o\!\downarrow t$ by t_o. Analogously we write $\check{t}_x$ for the static analog, i.e., $\check{t}_x$ corresponds to t_o with all references replaced one-to-one by their roles, and where in particular x is the role of o, i.e., $\check{o} = x$. Furthermore, during the run of C_t, we refer with r_x to the already past part of t_x, and s_x the still open future. Note that s_x is represented in the program code as part of the pairs of type $\mathsf{script} = \mathsf{assoc} \times \mathsf{future}$ in the $scripts$-variable, whereas the corresponding r_x is not remembered in the code; it is used for the induction in the proof, only. Remembered from the past in the code is only the association from roles to identities.

Lemma B.4.12. *Let t be a legal trace and $\Xi_0 \vdash C_t$ given as in Definition 3.3.20. Assume $\Xi_0 \vdash C_t \overset{r}{\Longrightarrow} \Xi \vdash C$. Let $[o]$ be an arbitrary component clique according to Ξ and $(\sigma, \check{s})$ and arbitrary elements from $[o].scripts$. Furthermore, the lock of $[o]$ is free. Then for all $o \in ran(\sigma)$:*

$$\check{r}_x\sigma = {}_o\!\downarrow r \quad \textit{and} \quad \check{r}_x\check{s}_x = \check{t}_x \quad \textit{and} \quad \check{s} = \check{s}_x \quad \textit{where } x = \sigma^{-1}(o) \ . \tag{B.43}$$

An alternative formulation (σ is one-to-one for component objects) of equation (B.43) reads: for all $x \in dom(\sigma)$,

$$\check{r}_x\sigma = {}_o\!\downarrow r \quad \textit{and} \quad \check{r}_x\check{s}_x = \check{t}_x \quad \textit{and} \quad \check{s} = \check{s}_x \quad \textit{where } o = \sigma(x) \ . \tag{B.44}$$

Proof. Straightforward. □

B.4.4 Substitutions

The implementation is centered around a static representation of the behavior of a given trace t and executes this representation, the scripts, step by step. Part of the current state of execution is an abstraction of the past, already executed script, matched against the actually happened trace. This abstraction is kept, per script, as an association from instance variables (the static "roles") to object identities. We can consider the associations also as substitutions from roles to identities. With this intuition in mind, we will assume in particular, that the substitutions are *injective* —two different roles cannot be taken by the same object— and that the domain and the ranges of the substitutions are separate: the substitutions do not rename variables, but assign values to them. Following conventional usage, we write $\varphi\sigma$ for applying the substitution σ to a formula φ.

Definition B.4.13 (Matching)**.** *We write $\varphi \lesssim \psi$ if there exists a substitution σ s.t., $\varphi = \psi\sigma$ ("ψ matches φ"). When we need to be explicit about the substitution, we also write $\varphi \lesssim_\sigma \psi$.*

The next lemma states a simple fact about substitutions. Remember that the definition of $\oplus$ requires the two substitutions being added to be of disjoint domain (cf. equation (B.8) from Definition B.2.6).

Lemma B.4.14 ($\oplus$)**.** *Assume two substitutions σ_1 and σ_2. If $\varphi\sigma_1 \lesssim \psi$ and $\varphi\sigma_2 \lesssim \psi$, then $\varphi\sigma \lesssim \psi$, where $\sigma = \sigma_1 \oplus \sigma_2$.*

Proof. Straightforward. □

Lemma B.4.15. *Assume two substitutions σ_1 and σ_2 with non-empty domain, s.t.*

$$dom(\sigma_1) \cap dom(\sigma_2) = \emptyset \text{ or } dom(\sigma_1) \subseteq dom(\sigma_2) \text{ or } dom(\sigma_1) \supseteq dom(\sigma_2)\ ,$$

and $ran(\sigma_1) \cap ran(\sigma_2) = \emptyset$. Let φ further be a formula which contains at least one variable $x_1 \in dom(\sigma_1)$ and one $x_2 \in dom(\sigma_2)$. If $\varphi\sigma_1 \lesssim \psi$ and $\varphi\sigma_2 \lesssim \psi$, then $\sigma = \sigma_1 \oplus \sigma_2$ is defined and $\varphi\sigma \lesssim \psi$. If additionally for all variables x of φ, $x \in dom(\sigma_1)$ or $x \in dom(\sigma_2)$, then $\varphi\sigma = \psi$.

Proof. If $dom(\sigma_1) \cap dom(\sigma_2)$, $\sigma = \sigma_1 \oplus \sigma_2$ is clearly defined and $\psi \lesssim \varphi\sigma$ is immediate with Lemma B.4.14. Now assume wlog. that $dom(\sigma_1) \subseteq dom(\sigma_2)$. By assumption, there exists at least one $x_1 \in dom(\sigma_1)$ that appears in the free variables of ψ. Since additionally, $x_1 \in dom(\sigma_2)$ and since the ranges of σ_1 and σ_2 are disjoint, clearly $\varphi\sigma_1 \lesssim \psi$ and $\varphi\sigma_2 \lesssim \psi$ cannot be true, yielding a contradiction to $dom(\sigma_1) \subseteq dom(\sigma_2)$.

The second point of the lemma is an immediate consequence. □

Overview over the code

We conclude by providing an overview of the pieces of code we used to program the observer C_t for a given trace t. The notations are given in Table B.1. We have not given the code of those definitions to the lowest level, since, once having assured mutual exclusion, the data manipulations are straightforward,

data type	explanation	notation	used
assoc	update of assoc	$\sigma := \sigma[y \mapsto self]$	B.2.4
assoc		$\oplus, \bigoplus, \cup$	B.2.6,B.2.10
assoc	domain	if $dom(\sigma_1) \cap dom(\sigma_2) = \emptyset$	B.2.6
assoc	set comprehension	$\{\ldots \mid \ldots\}$	B.2.6
	init	$initialize$	B.2.4
	empty assoc	$\sigma_\perp$	B.2.13
all	being undef	$x = \perp$ or $\neq$	B.2.4
	skip	()	B.2.4,B.2.8
set of script	iterator	$\forall(\sigma, \check{s}) \in$	B.2.8
assoc	subst.	$\check{a}\sigma$	B.2.8
label	equality	$a_1 = a_2$	B.2.8
set of assoc	iterator	$\exists\sigma \in$	B.2.8
script	match/shorten		B.2.8,B.2.10
set of script	$\exists$-iterator	$\exists(\sigma, \check{s}) \in$	B.2.10
assoc/label	creation	$create(\check{a})$	B.2.10
	wildcard	_	B.2.10
		$self.\Sigma$	B.2.10
		$broadcast$	B.2.10
	app. of permutation	$\_ \circ \pi(\check{a}_1, \check{a}_2)$	B.2.10
		$interpret$	B.2.10
	mutex locks	$(\!\|/\|\!)$	B.2.10/B.2.12
	output step (reaction)	$step^o$	
	input step	$step^i$	
	output sync. code	t^o_{sync}	
	input sync. code	t^i_{sync}	
	case/pattern matching	$case/esac$	B.2.12
	comprehension	"where"	B.2.13
	compound return value		B.2.13
	multiple declaration	$let\ x, y{:}c = \ldots$	B.2.19
lock	undefined	$\perp$	B.2.19
	known objects	$self.\Theta$	B.2.26
	iteration	$\Theta.(\!\|$	B.2.26

Table B.1: Overview

if tedious, and the object calculus is Turing complete. See for instance [2, Section 6.3] for an encoding of the untyped λ-calculus into the untyped object calculus. The encoding presented there uses only field update, even if the object calculus used for the encoding features also method update. Note also that Definition B.2.21 provides the code for lock-handling, i.e., to assure mutual exclusion, in the native calculus.

Ultimately, the encoding of C_t rests on the fact that the trace t is a finite sequence mentioning only a finite number of identities, which are represented in a finite number of instances variables, the roles (cf. Definition 3.3.17). See also Remark 3.3.19 on page 74 about the encoding of the associations σ. The operations shown in Table B.1 do not operate on $dom(\sigma)$ as a "set" of instance variable, rather they access, i.e., query and update, the instance variable constituting $dom(\sigma)$.

APPENDIX C

Multithreading

This chapter collects the proofs concerning the multi-threaded language. A number of proofs directly correspond to the ones in the sequential setting; in those cases we do not repeat them.

C.1 Operational semantics

The properties of the corresponding Section A.1 carry over to the multithreaded setting. We show only the generalization of the invariants of Lemma A.1.3.

Lemma C.1.1 (Invariants)**.** *Assume* $\Xi_0 \vdash C \xRightarrow{t} \acute{\Xi} \vdash \acute{C}$*. Then* $\acute{\Xi} \vdash \acute{C} = \acute{\Delta}, \acute{\Sigma} \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}$ *with*

1. $\acute{E}_\Delta \subseteq \acute{\Delta} \times (\acute{\Delta} + \acute{\Theta})$ *and* $\acute{E}_\Theta \subseteq \acute{\Theta} \times (\acute{\Theta} + \acute{\Delta})$*.*
2. $dom(\acute{\Delta}) \cap dom(\acute{\Theta}) = \emptyset$*, for all object and class references.*
3. *for thread identifiers* n *we have:* $\acute{\Sigma} \vdash n : thread$ *iff exactly one of the two assertions* $\acute{\Delta} \vdash \odot_n$ *or* $\acute{\Theta} \vdash \odot_n$ *holds.*

Proof. Analogous to the proof of Lemma A.1.3. □

C.2 Closure

The closure relation $\sqsubseteq_\Theta$ is given in Section 5.2.2 on page 100. An important part is the relation $\preccurlyeq_\Theta$ from Definition 3.1.8, embodying the tree-like structure of the merging cliques plus the replay. In the concurrent setting, it additionally contains the uncertainty of observations by concurrent threads ("switching") due to the fact that interface interaction are themselves not atomically observable since they are side-effect free. In this section, we show the "soundness" of the closure relation.

C.2.1 Traces as trees

Lemma C.2.1 (Shortening)**.** *Assume* $\Xi_0 \vdash sa \preccurlyeq_\Theta ta' : trace$*, where* a' *is a renaming of* a*, and where the labels* a *resp.* a' *are unique in the following sense: neither* a *nor a renaming occurs in* s *or in* t*. Then* $\Xi_0 \vdash s \preccurlyeq_\Theta t : trace$*.*

Proof. The property follows straightforwardly from the definition of $\preccurlyeq_\Theta$ (Definition 3.1.8) and the uniqueness of label a; in particular, a is not a renaming of any label occurring in t, for otherwise, label a in sa on the left-hand side of $\Xi_0 \vdash sa \preccurlyeq_\Theta ta : trace$ could be justified by a renaming of a variant occurring in ta on the right-hand side. □

Lemma C.2.2. *If* $\Xi \vdash C \xRightarrow{t^+}$ *and* $\Xi \vdash s^+ \preccurlyeq_\Theta, t^+$*, then* $\Xi \vdash C \xRightarrow{s^+}$*.*

Proof. Straightforward. □

C.2.2 Switching

Switching is defined in Section 5.2.2. We start with a simple fact about the switching relation: when changing the perspective from the component side to the environment side, the direction of the switching relation reverses.

Lemma C.2.3 (Switching & duality)**.** $\Xi \vdash s \sqsubseteq_\Theta^{switch} t$ *iff.* $\Xi \vdash s \sqsupseteq_\Delta^{switch} t$ *(resp.* $\bar{\Xi} \vdash \bar{s} \sqsupseteq_\Theta^{switch} \bar{t}$*).*

Proof. Straightforward from the switching rules of Table 5.2, and using the fact that dualization preserves legality (Lemma A.5.10 on page 211). In particular, the direction of the inequation O-OI *reverses* when dualizing. □

Next we justify the switching rules from Table 5.2 as part of the trace closure. We start by the basic commutativity properties of the basic internal and external reduction (resp. congruence) steps. The only two actions which access the state, i.e., the values of the field variables of objects, are method update and internal method call/field lookup (cf. rules CALL_i, FLOOKUP, and FUPDATE of Table 4.5). All other steps are satisfying a diamond property, respectively a commuting diamond property. We formulate the independence of those steps as a commutation property for steps of two different threads.

For the properties in relation with lock handling (in particular for analyzing the code of ⦇ and ⦈ from Definition B.2.21) we need to be more fine-grained: we distinguish read-access and write-access to the instance state by distinguishing between τ_r- and τ_w-steps: τ_r-steps are justified by rule CALL_i and τ_w-steps by FUPDATE.[1] Furthermore we write τ_r^m when CALL_i is used for a method call, not for a field access. When writing τ, we mean either τ_r, τ_w, or τ_r^m.

Lemma C.2.4 (Non-interference)**.** *Assume $\Xi \vdash C \to_1 \to_2 \acute{\Xi} \vdash \acute{C}$, where $\to_1$ and $\to_2$ are each an instance of one of the relations $\xrightarrow{\tau}$, $\rightsquigarrow$, $\xrightarrow{a}$, and $\equiv$. If not both of the relations $\to_1$ and $\to_2$ are $\xrightarrow{\tau}$-steps and if they are not performed by the same thread (where a $\equiv$-step is not performed by any thread), then $\Xi \vdash C \to_2 \to_1 \acute{\Xi} \vdash \acute{C}$.*

Furthermore, τ_1 and τ_2 can be switched, if both are executed by two different threads, and if both are τ_r-steps or if one of them is a τ_r^m-step, or of both are executed in different instances.

Proof. Straightforward. □

Lemma C.2.5 (Switching)**.** *If $\Xi_0 \vdash C_0 \overset{r}{\Longrightarrow} \Xi \vdash C \xrightarrow{a_1} \Longrightarrow \xrightarrow{a_2} \acute{\Xi} \vdash \acute{C}$, then also the following reductions are possible:*

1. *$\Xi_0 \vdash C_0 \overset{r}{\Longrightarrow} \Xi \vdash C \xrightarrow{a_2} \xrightarrow{a_1} \Longrightarrow \acute{\Xi} \vdash \acute{C}$, where $a_1 = \gamma_1?$ and $a_2 = \gamma_2?$.*
2. *$\Xi_0 \vdash C_0 \overset{r}{\Longrightarrow} \Xi \vdash C \Longrightarrow \xrightarrow{a_2} \xrightarrow{a_1} \acute{\Xi} \vdash \acute{C}$, where $a_1 = \gamma_1!$ and $a_2 = \gamma_2!$.*
3. *$\Xi_0 \vdash C_0 \overset{r}{\Longrightarrow} \Xi \vdash C \xrightarrow{a_2} \xrightarrow{a_1} \Longrightarrow \acute{\Xi} \vdash \acute{C}$ and also $\Xi_0 \vdash C_0 \overset{r}{\Longrightarrow} \Xi \vdash C \Longrightarrow \xrightarrow{a_2} \xrightarrow{a_1} \acute{\Xi} \vdash \acute{C}$, where $a_1 = \gamma_1!$ and $a_2 = \gamma_2?$.*

Proof. First note that in all three cases, $\Xi_0 \vdash C_0 \overset{r}{\Longrightarrow} \Xi \vdash C \xrightarrow{a_1} \Longrightarrow \xrightarrow{a_2}$ implies that a_1 and a_2 are labels concerning two different threads. Let us denote the thread of a_1 as n_1 and the one of a_2 as n_2. The underlying reason for the property of the lemma is that the steps $\xrightarrow{a_1}$ and $\xrightarrow{a_2}$ themselves are side-effect free, and that a thread is inactive once it has left the component or before it (re-)enters a component (cf. also Lemma C.2.4).

Case: $\Xi \vdash C \xrightarrow{\gamma_1?} \Longrightarrow \Xi' \vdash C' \xrightarrow{\gamma_2?} \acute{\Xi} \vdash \acute{C}$

None of the steps in $\Xi \vdash C \xrightarrow{\gamma_1?} \Longrightarrow \Xi' \vdash C'$ is done by thread n_2, and thus the result follows by iterated application of Lemma C.2.4.

[1] The rule NEWO_i for instantiation from Table 4.5 accesses the state as kept in the classes, as well. Since classes are "*read-only*", those steps do not interfere with any others, and we used a $\rightsquigarrow$-step in the internal semantics.

Case: $\Xi \vdash C \xrightarrow{\gamma_1!} \Xi' \vdash C' \Longrightarrow \xrightarrow{\gamma_2!} \acute{\Xi} \vdash \acute{C}$
Analogous to the previous case with Lemma C.2.4, except that here we use that none of the steps in $\Xi' \vdash C' \Longrightarrow \xrightarrow{\gamma_2!} \acute{\Xi} \vdash \acute{C}$ is executed by n_1.

Case: $\Xi \vdash C \xrightarrow{\gamma_1!} \Longrightarrow \xrightarrow{\gamma_2?} \acute{\Xi} \vdash \acute{C}$
Analogous to the previous cases. □

Note that switching of the fourth combination ($a_1 = \gamma_1?$ and $a_2 = \gamma_2!$) is not possible. The reason is that in the reduction $\Xi \vdash C \xrightarrow{\gamma_1?} \Longrightarrow$ there might be side-effects that make the second interaction impossible, in other words, the second $\gamma_2!$ might be causally dependent on $\gamma_1?$.

C.2.3 Closure

The following lemma for information order closure justifies the definition of the $\sqsubseteq_\Theta$-relation: If a component realizes a trace s, all traces in the closure, i.e., all traces $\sqsubseteq_\Theta s$, are also possible.

Lemma C.2.6 (Closure). *If* $\Xi_0 \vdash C \overset{t}{\Longrightarrow}$ *and* $\Xi_0 \vdash s \sqsubseteq_\Theta t : \mathit{trace}$, *then* $\Xi_0 \vdash C \overset{s}{\Longrightarrow}$.

Proof. By induction on the length of the derivation for $\Xi_0 \vdash s \sqsubseteq_\Theta t : \mathit{trace}$ (cf. Table 5.2 on page 100). The cases for reflexivity and transitivity are trivial, resp., by straightforward induction.

Case: O-INPUT: $\Xi_0 \vdash t\gamma? \sqsubseteq_\Theta t : \mathit{trace}$
We are given $\Xi_0 \vdash C \overset{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$. By assumption, $t\gamma?$ is legal, i.e., $\Xi_0 \vdash t\gamma? : \mathit{trace}$, and we distinguish two subcases:

Subcase: $\gamma? = \nu(\Phi').n\langle \mathit{call}\ o_r.l(\vec{v})\rangle?$
In this case, legality is justified by one of the L-CALLI-rules from Table 5.1 on page 99 in the last step. Inverting any of these rules yields that the component is input enabled, i.e., $\Xi_0 \vdash t \rhd o_r \overset{\lfloor\gamma?\rfloor}{\leftarrow} o_s$, and furthermore that the incoming values meet the typing assumptions. By definition of enabledness (Definition 3.3.3 on page 57) applied to thread n, either $\Theta \vdash \odot$ and $\mathit{pop}\ n\ t = \bot$, or $\mathit{pop}\ n\ t = \gamma'!$, which implies using the subcases of Lemma A.5.5(1) that either n is not contained in C, or $\acute{\Xi} \vdash \acute{C}$ is of the form $C' \parallel n\langle \mathit{let}\, x{:}T' = o\ \mathit{blocks\ for}\ o_s\ \mathit{in}\, t\rangle$ or $C' \parallel n\langle \mathit{stop}\rangle$. Depending on the situation, $\acute{\Xi} \vdash \acute{C}$ accepts the incoming call by one of the CALLI-rules from Table 4.8.

Subcase: $\gamma? = \nu(\Phi').n\langle \mathit{return}(v)\rangle?$
Inverting rule L-RETI gives that n is input-return enabled after t, i.e., $\Xi_0 \vdash t \rhd o_r \overset{\lfloor\gamma?\rfloor}{\leftarrow} o_s$, where γ is a return. By Definition 3.3.3 of enabledness, this means $\mathit{pop}\ n\ t = \nu(\Phi'').n\langle \mathit{call}\ o_s.l(\vec{v})\rangle!$, and by Lemma A.5.5(2), $\acute{C}$ is of the form $C' \parallel n\langle \mathit{let}\, x{:}T = o_r\ \mathit{blocks\ for}\ o_s\ \mathit{in}\, t\rangle$, hence the step can be taken by rule RETI.

Case: O-SWAPREPLAY$_\Theta$
The relation $\preccurlyeq_\Theta$ is dealt with by Lemma C.2.2.

Case: O-II
With the switching Lemma C.2.5(1). The steps for rules O-OO and O-OI are covered similarly by the other parts of Lemma C.2.5 on the preceding page. □

C.3 Soundness

Proof of Soundness (Lemma 5.2.1 on page 98). We have to show that if $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$, then $\Xi_0 \vdash C_1 \sqsubseteq_{obs} C_2$.

So assume $\Xi_0 \vdash C_1 \sqsubseteq_{trace} C_2$ and an observer $\bar{\Xi}_0, c_b{:}\mathsf{barb} \vdash C_0$ such that $(C_1 \parallel C_0) \Downarrow_{c_b}$, i.e., $(C_1 \parallel C_0) \Longrightarrow C' \downarrow_{c_b}$. The context $\bar{\Xi}_0$ corresponds to Ξ_0 with the roles of assumptions and commitments exchanged. Since $c_b{:}\mathsf{barb} \vdash C_1 \parallel C_0 : ()$ (cf. rule T-PAR and subsumption), the merging Lemma A.4.2 and decomposition (Lemma A.4.12) yield

$$\Xi_0, c_b{:}\mathsf{barb} \vdash C_1 \stackrel{t_1^+}{\Longrightarrow} \Xi', c_b{:}\mathsf{barb} \vdash C_1' \tag{C.1}$$

and

$$\bar{\Xi}_0, c_b{:}\mathsf{barb} \vdash C_0 \stackrel{\bar{t}_1^+}{\Longrightarrow} \bar{\Xi}', c_b{:}\mathsf{barb} \vdash C_0' \,, \tag{C.2}$$

where $C' \equiv \nu(\Phi' \setminus \Phi).C_0' \between C_1'$. By Lemma A.4.3, $C' \downarrow_{c_b}$ implies that either C_1' or C_0' strongly barbs on c_b. Due to the typing assumptions, as in the sequential setting, only $C_0' \downarrow_{c_b}$ is possible, i.e.,

$$\bar{\Xi}', c_b{:}\mathsf{barb} \vdash C_0' \xrightarrow{succ} ,$$

where, in abuse of notation, the success-reporting external label $succ$ is of the form $\nu(b{:}c_b)\, n\langle[o_{succ}]\, call\ b.succ()\rangle!$ with o_{succ} as the representative of the success-reporting clique.By weakening and by definition of $\sqsubseteq_{trace}$ (Definition 5.1.6), we have, in particular for $[o_{succ}]$, that $\Xi_0, c_b{:}\mathsf{barb} \vdash C_2 \stackrel{t_2^+}{\Longrightarrow} \Xi'', c_b{:}\mathsf{barb} \vdash C_2''$ for some trace t_2^+ such that

1. $\Xi_0 \vdash {}_o{\downarrow}\, t_2^+ = {}_o{\downarrow}\, t_1^+$, for all environment objects $o \in [o_{succ}]$ and
2. $\Xi_0, c_b{:}\mathsf{barb} \vdash t_2^+ \preccurlyeq_\Delta t_1^+ : trace$.

Since neither t_1^+ nor t_2^+ mention the additional class name c_b, the latter statement can be strengthened to $\Xi_0 \vdash t_2^+ \preccurlyeq_\Delta t_1^+ : trace$, and dualized to $\bar{\Xi}_0 \vdash \bar{t}_2^+ \preccurlyeq_\Theta \bar{t}_1^+ : trace$, which implies $\bar{\Xi}_0, c_b{:}\mathsf{barb} \vdash \bar{t}_2^+ \preccurlyeq_\Theta \bar{t}_1^+ : trace$. Hence by the closure Lemma C.2.2 for $\preccurlyeq_\Theta$, $\bar{\Xi}_0, c_b{:}\mathsf{barb} \vdash C_0 \stackrel{\bar{t}_2^+}{\Longrightarrow}$. Further by the composition Lemma A.4.6

$$C \Longrightarrow C'' \,,$$

where $C = C_0 \between C_2 = C_0 \parallel C_2$ and $C'' \equiv \nu(\Phi'' \setminus \Phi).C_0'; \between C_2''$. Since additionally, $C'' \downarrow_{c_b}$ due to condition 1, the result follows. □

C.4 Completeness

Proof of Lemma 5.2.8 on page 104 (total correctness). Almost identical to the proof in the deterministic setting (cf. Lemma 3.3.23). We show $\Xi_0 \vdash C_t \stackrel{r}{\Longrightarrow} \Xi \vdash C$ for all prefixes $r \preccurlyeq t$. So let $t = r\ s$. As usual, Ξ abbreviates the pair of $\Delta, \Sigma; E_\Delta$ and $\Theta, \Sigma; E_\Theta$. The proof proceeds by induction on the prefix r, using the following induction hypotheses, where the last one concerning the locks is added compared to the sequential setting.

1. $\Xi \vdash C :: s$ (cf. Definition 3.3.22).

2. Depending on the enabledness condition after trace r, the thread in C_t is of the form as shown in Table A.6.

3. All locks are free.

Case: $r = \epsilon$
Without initial component objects, part 1 is trivially satisfied. When C_t is initially *input* enabled, $\Delta_0 \vdash \odot$. By the construction from Definition 5.2.7, C_t contains no thread in this situation, i.e., the condition of input-enabled threads in part 2 is met (with $t_{ie} = \epsilon$). If otherwise $\Theta_0 \vdash \odot$, i.e., the empty trace is *output*-enabled, the initial code contains

$$n\langle t_0 \rangle = n\langle \mathit{let}\ x{:}c_i\ \mathit{in}\ x.\rangle\!\rangle; x.\mathit{start}()\rangle \ , \tag{C.3}$$

for some class c_i, where the named thread n is hidden via a ν-binder. Thus, the initial configuration starts as follows, where in the reduction, the let-bound variable x is omitted after the second step, as it is never referenced other than in the calls of $\rangle\!\rangle$ and start:

$$\begin{array}{ll}
\Xi_0 \vdash \nu(n{:}\mathit{thread}).(C_0' \parallel n\langle t_0 \rangle) & \Longrightarrow \\
\Xi_0 \vdash C_0' \parallel \nu(o_\odot{:}c_i, n{:}\mathit{thread}).(o_\odot[F_i, M_i] \parallel n\langle o_\odot.\rangle\!\rangle; o_\odot.\mathit{start}()\rangle) & \Longrightarrow \\
\Xi_0 \vdash C_0' \parallel \nu(o_\odot{:}c_i, n{:}\mathit{thread}).(o_\odot[F_i, M_i] \parallel n\langle o_\odot.\mathit{start}()\rangle) & \Longrightarrow \\
\Xi_0 \vdash C_0' \parallel \nu(o_\odot{:}c_i, n{:}\mathit{thread}).(o_\odot[F_i, M_i] \parallel n\langle M_i.\mathit{start}(o_\odot)()\rangle) & \Longrightarrow \\
\Xi_0 \vdash C_0' \parallel \nu(o_\odot{:}c_i, n{:}\mathit{thread}).(o_\odot[F_i, M_i] \parallel n\langle t^o_{\mathit{sync}}()\rangle) \ . &
\end{array}$$

Hence, after this initial reduction, n is of the form as required by part 2. The lock of the only object $o_\odot$ is set to be free by $\rangle\!\rangle$, covering part 3. Note that the above initial reduction is *deterministic* (up-to structural congruence, of course). Note also that the object $o_\odot$ is not the same as the symbol $\odot$.

Case: $r = r'\ a$
We are given $\Xi_0 \vdash C_t \stackrel{r'}{\Longrightarrow} \Xi \vdash C$, and distinguish according to the nature of the next label a.

Subcase: $a = \nu(\Delta', \Sigma', \Theta').n\langle \mathit{call}\ o_r.l(\vec{v})\rangle?$
Different from the single-threaded case, the incoming call can be caused by a *new* thread, in which case $\Sigma' \vdash n$, and where the incoming label is justified by L-CALLI$_0$ in the legal trace system.[2]

Subsubcase: L-CALLI$_{1,2}$, i.e., $a = \nu(\Delta', \Theta').n\langle \mathit{call}\ o_r.l(\vec{v})\rangle?$,
and Σ' is empty. In this case the thread n is input enabled (or stronger input return enabled) after r'. Thus the named thread is of one of the corresponding forms of Table A.6, i.e., either blocked or stopped. The reduction looks as

[2] As we do not allow to send thread names as arguments of method calls and returns, Σ' either is empty, or $\Sigma' = n{:}\mathit{thread}$. If we allowed the sending of thread names, the theory would not change in crucial ways. Of course, connectivity assertions of the form $o \hookrightarrow n$ ("object o may have knowledge of the thread name n") would have to be considered, but that would be a mild generalization. In particular, since objects do not "communicate with" threads —objects communicate with each other by *executing* threads— nor do threads communicate with each other —other than via storing values in objects— there would be no evolving clique structure of thread names. In [12], we considered this generalization.

follows:

$$
\begin{array}{ll}
\Xi \vdash C & = \\
\Xi \vdash n\langle t_{ie}\rangle \parallel C' & \xrightarrow{a} \\
\acute{\Xi} \vdash n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C' \parallel C(\Theta') & \xrightarrow{\tau} \\
\acute{\Xi} \vdash n\langle let\ x{:}T = M_r.l(o_r)(\vec{v})\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C' \parallel C(\Theta') & = \\
\acute{\Xi} \vdash n\langle let\ x{:}T = t_{body}[o_r/s][\vec{v}/\vec{x}]\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C' \parallel C(\Theta') & = \\
\acute{\Xi} \vdash n\langle let\ x{:}T = t^i_{sync}; t^o_{sync}\ in\ o_r\ returns\ x\ to\ o_s; t_{ie}\rangle \parallel C' \parallel C(\Theta')\ . &
\end{array}
$$

The external a-step is justified by CALLI_1 or CALLI_2 of the external semantics. Note that the rule CALLI_0 for calls by a new thread does not apply here, The code of t_{body} is part of the definition of C_t (cf. Definition 5.2.7, equation (5.11)). The code t^i_{sync} for input synchronization is given in Definition B.2.2 on page 225. At this point, the preconditions of the lemma for input synchronization are satisfied, and we can continue as in the sequential case. In addition to the sequential case, the lock is free after the reduction after t^i_{sync}, by executing $\rangle$, as required by part 3.

Subsubcase: L-CALLI_0, i.e. $a = \nu(\Delta', \Sigma', \Theta').n\langle call\ o_r.l(\vec{v})\rangle?$, with $\Sigma' = n{:}thread$. Similar.

The remaining cases work similar, too. □

C.4.1 Definability: disentangling

To get a grip on the complications due to concurrency, we define a clean reduction of the observer as a strict alternation between the synchronization steps of different threads. For the code of the synchronization code, see Definition B.2.2 on page 225 and B.2.11 on page 229.

Definition C.4.1 (Clean reduction)**.** *Let C_t be the component given by Definition 5.2.7. A clean reduction $\Xi_0 \vdash C_t \stackrel{s}{\Longrightarrow}$ is defined by induction on the length of s.*

1. *If s is the empty trace, the empty reduction sequence is clean.*
2. *Assume $\Xi_0 \vdash C_t \stackrel{s}{\Longrightarrow} \Xi \vdash C$ is a clean reduction, and let n be the thread of $\gamma?$, resp. of $\gamma!$.*
   (a) *Then $\Xi_0 \vdash C_t \stackrel{s}{\Longrightarrow} \Xi \vdash C \stackrel{\gamma?}{\xrightarrow[n]{}\!\!\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ is clean, where the sequence from C to $\acute{C}$ consists of the incoming communication followed by all of the corresponding input synchronization code t^i_{sync}, but no steps of other threads.*
   (b) *Then $\Xi_0 \vdash C_t \stackrel{s'}{\Longrightarrow} \Xi \vdash C \stackrel{\gamma!}{\Longrightarrow\!\!\xrightarrow[n]{}} \acute{\Xi} \vdash \acute{C}$ is clean, where the sequence from C to $\acute{C}$ consists of all of the corresponding input synchronization code t^o_{sync} followed by the outgoing communication, but no steps of other threads.*

The next lemma states that, given an arbitrary reduction sequence s of the programmed C_t, one can always come up with a different clean sequence possible by C_t, obtained from s by switching labels in *reverse* order. In some sense, the lemma therefore is the opposite to the switching Lemma C.2.5.

Lemma C.4.2 (Disentangle)**.** *Let C_t be the component as given by Definition 5.2.7. If $\Xi_0 \vdash C_t \overset{s}{\Longrightarrow} \Xi \vdash C$, then there exists a* clean *reduction $\Xi_0 \vdash C_t \overset{s'}{\Longrightarrow} \Xi \vdash C$ with $\Xi_0 \vdash s \sqsubseteq_\Theta^{switch} s'$.*

Proof. With the help of the Lemma B.2.25, which allows to disentangle the s step by step. □

For clean traces, we can prove exactness analogous to the sequential case.

Lemma C.4.3 (Exactness/partial correctness)**.** *Let t be a legal trace and $\Xi_0 \vdash C_t$ given by Definition 5.2.7.*

$$\textit{If} \quad \Xi_0 \vdash C_t \overset{s}{\Longrightarrow} \quad \textit{is a clean reduction, then} \quad \Xi_0 \vdash s \preccurlyeq_\Theta t : \mathit{trace} \ . \tag{C.4}$$

Proof. Analogous to the proof of Lemma 3.3.24 on page 75. □

Proof of Lemma 5.2.10 on page 107 (exactness/partial correctness). Given $\Xi_0 \vdash C_t \overset{s}{\Longrightarrow}$, then by disentangling (Lemma C.4.2), also $\Xi_0 \vdash C_t \overset{s'}{\Longrightarrow}$ by some clean reduction, with $\Xi_0 \vdash s \sqsubseteq_\Theta^{switch} s'$. Exactness for clean reductions from Lemma C.4.3 gives, $\Xi_0 \vdash s' \preccurlyeq_\Theta t$, and hence $\Xi_0 \vdash s \sqsubseteq_\Theta t$ by transitivity of $\sqsubseteq_\Theta$, as required. □

Proof of Corollary 5.2.11 on page 107. There are two directions to show. We are given the legal trace $\Xi_0 \vdash t : \mathit{trace}$. Construct C_t according to Definition 5.2.7 on page 103. By total correctness of Lemma 5.2.8, $\Xi_0 \vdash C_t \overset{t}{\Longrightarrow}$.

Case: $\Leftarrow$
By the closure Lemma C.2.6, immediately $\Xi_0 \vdash C_t \overset{s}{\Longrightarrow}$, as required.

Case: $\Rightarrow$
The reverse direction follows by exactness from Lemma 5.2.10. □

Proof of completeness (Theorem 5.2.12 on page 107). In the sequential setting, the proof that resembles this one is not the proof of completeness (Theorem 3.3.29, proof at page 219), but the proof for the corresponding property for $\sqsubseteq_{trace}^{nondet}$ (Lemma 3.3.26, proof at page 216).

Assume an augmented trace t_1^+ with $\Xi_0 \vdash C_1 \overset{t_1^+}{\Longrightarrow}$, and let $[o_1]$ be an arbitrary clique of the observer after t_1^+. According to Definition 5.1.6 we need to show that $\Xi_0 \vdash C_2 \overset{t_2^+}{\Longrightarrow}$ for some trace t_2^+, s.t.

1. $\Xi_0 \vdash {}_o\!\downarrow t_1^+ = {}_o\!\downarrow t_2^+$, for all $o \in [o_1]$, and
2. $\Xi_0 \vdash t_2^+ \preccurlyeq_\Delta t_1^+ : \mathit{trace}$.

Note that the replay relation is considered from the perspective of the environment: the observer cannot distinguish whether one behavior is done once, or more than once.

First assume that t_1^+ is empty. The result follows by choosing $t_2 = t_2^+ = \epsilon$.

So assume $t_1^+ \neq \epsilon$. We start with part 2 and concentrate on the case where the last interaction of the clique $[o_1]$ is an *output* (from the perspective of C_1, so for the observer, it is an input), i.e.,

$$t_1^+ = r_1^+ \ \gamma! \ s_1^+ \ . \tag{C.5}$$

Consider the dual trace $\bar{t}_1^+$, i.e., the trace from the perspective of the receiver of the communication and the observer. As t_1^+ is legal (using soundness of the legal trace system from Lemma A.5.9), the complement is legal, too (by trace duality from Lemma A.5.10), i.e.,

$$\bar{\Xi}_0 \vdash \bar{r}_1^+ \ \gamma? \ \bar{s}_1^+ : \mathit{trace}\,. \tag{C.6}$$

It is easy to see —there are no arguments to the succ-call and hence there is no connectivity information involved; furthermore, extending a weakly balanced trace by a call of a new thread does not break the balance conditions— that also the trace extended by one outgoing success-reporting action is legal, i.e.,

$$\bar{\Xi}_0' \vdash \bar{r}_1^+\gamma?\bar{s}_1^+\mathit{succ}! : \mathit{trace},$$

where succ abbreviates $(\nu b{:}c_b, n{:}\mathit{thread}).n\langle[o_1]\mathit{call}\ b.\mathit{succ}()\rangle!$, and where the context $\bar{\Xi}_0'$ is given by extending the environment $\bar{\Delta}_0$ to $\bar{\Delta}_0, c_b{:}\mathsf{barb}$. Note that the sender clique of $\mathit{succ}!$ is the receiver of $\gamma?$.

Consider the component $\bar{\Xi}_0' \vdash C_{\bar{t}_1^+\,\mathit{succ}!}$, and let us abbreviate the observer $C_{\bar{t}_1^+\,\mathit{succ}!}$ as C_O, and furthermore let $\bar{\Xi}_b$ stand for the context $c_b{:}\mathsf{barb}$. Since initially, C_1 and C_O are static, $C_1 \between C_O = C_1 \parallel C_O$. By total correctness of C_O (Lemma 5.2.8) and composition (Lemma A.4.6), $\Xi_b \vdash C_1 \parallel C_O \Longrightarrow \Xi_b \vdash \acute{C}_{1,O} \downarrow_{c_b}$, or more explicitly:

$$\Xi_b \vdash C_1 \parallel C_O \overset{\substack{t_1^+\\ \bar{t}_1^+}}{\Longrightarrow} \Xi_b \vdash \acute{C}_{1,O} \downarrow_{c_b}\,, \tag{C.7}$$

where the internal reduction $\Longrightarrow$ is decorated by the two complementary traces and where furthermore $\acute{C}_{1,O} = \nu(\acute{\Phi} \setminus \Phi).\acute{C}_1 \between \acute{C}_O$ (= $\nu(\acute{\Phi}).\acute{C}_1 \between \acute{C}_O$ since Φ contains no bindings for object or thread names). As $\Xi_0 \vdash C_1 \sqsubseteq_{may} C_2$, we can replace C_1 by C_2 and still observe success (Definition 5.1.7), i.e., $\Xi_b \vdash C_2 \parallel C_O \Longrightarrow\downarrow_{c_b}$. By trace decomposition (Lemma A.4.12),

$$\Xi_b \vdash C_2 \parallel C_O \overset{\substack{t_2^+\\ \bar{t}_2^+}}{\Longrightarrow} \Xi_b \vdash \acute{C}_{2,O} \downarrow_{c_b} \tag{C.8}$$

for some trace t_2^+, more precisely:

$$\Xi_b, \Xi_0 \vdash C_2 \overset{t_2^+}{\Longrightarrow} \Xi_b, \acute{\Xi}_2 \vdash \acute{C}_2 \quad \text{and} \quad \Xi_b, \bar{\Xi}_0 \vdash C_O \overset{\bar{t}_2^+}{\Longrightarrow} \Xi_b, \bar{\acute{\Xi}}_2 \vdash \acute{C}_O\,, \tag{C.9}$$

with $C_{2,O} = \nu(\acute{\Phi}_2).\acute{C}_2 \between \acute{C}_O$. Disentangling the reduction on the right for the observer with Lemma C.4.2, we obtain a clean reduction

$$\Xi_b, \bar{\Xi}_0 \vdash C_O \overset{\bar{u}_2^+}{\Longrightarrow} \Xi_b, \bar{\acute{\Xi}}_2 \vdash \acute{C}_O \quad \text{where} \quad \Xi_b, \bar{\Xi}_0 \vdash \bar{t}_2^+ \sqsubseteq_\Theta^{\mathit{switch}} \bar{u}_2^+. \tag{C.10}$$

Duality for switching from Lemma C.2.3 yields $\Xi_b, \Xi_0 \vdash t_2^+ \sqsupseteq_\Theta^{\mathit{switch}} u_2^+$, and hence the reduction for C_2, the left reduction of (C.9), can be reordered as follows, using the switching Lemma C.2.5:

$$\Xi_b, \Xi_0 \vdash C_2 \overset{u_2^+}{\Longrightarrow} \Xi_b, \acute{\Xi}_2 \vdash \acute{C}_2\,. \tag{C.11}$$

Using the composition Lemma A.4.6 again, (C.10) and (C.11) together yield

$$\Xi_b \vdash C_2 \parallel C_O \overset{\substack{u_2^+ \\ \Longrightarrow \\ \bar{u}_2^+}}{} \Xi_b \vdash \acute{C}_{2,O} \downarrow_{c_b} , \tag{C.12}$$

i.e., a variant of (C.8) where $\bar{u}_2^+$ is *clean*. Note that $\acute{C}_{2,O}$ is not affected by reordering t_2^+, resp., $\bar{t}_2^+$ to u_2^+, resp., $\bar{u}_2^+$. For the observer, this means

$$\Xi_b, \bar{\Xi}_0 \vdash C_O \overset{\bar{u}_2^+ \, succ'!}{\Longrightarrow} \tag{C.13}$$

by a *clean* reduction. Note that $succ'!$ may be an α-variant of $succ!$ (which is defined as $(\nu b{:}c_b, n{:}thread).n\langle[o_1]call\ b.succ()\rangle!$). By partial correctness for clean reductions from Lemma C.4.3,

$$\Xi_b, \bar{\Xi}_0 \vdash \bar{u}_2^+ \ succ'! \preccurlyeq_\Theta \bar{t}_1^+ \ succ! \, . \tag{C.14}$$

Since $succ$, resp., $succ'$ is unique, i.e., no α-variant occurs in $\bar{u}_2^+$ or in $\bar{t}_1^+$, we can shorten the two traces with the help of Lemma C.2.1:

$$\Xi_b, \bar{\Xi}_0 \vdash \bar{u}_2^+ \preccurlyeq_\Theta \bar{t}_1^+ \, . \tag{C.15}$$

Without the trailing label $succ$, we can strengthen that statement to

$$\bar{\Xi}_0 \vdash \bar{u}_2^+ \preccurlyeq_\Theta \bar{t}_1^+ . \tag{C.16}$$

By definition, this is equivalent to $\Xi_0 \vdash u_2^+ \preccurlyeq_\Delta t_1^+$, covering part 2 of $\sqsubseteq_{trace}$.

For part 1, we argue as follows. Still, $[o_1]$ is the arbitrarily chosen environment clique after t_1^+, i.e., a clique of the observer, which is also the sender clique of $succ!$ after t_1^+. Equation (C.14) from above means by Definition 3.1.8 of $\preccurlyeq_\Theta$, that for all component cliques[3] $[o_2']_{/_{\bar{\Xi}_2'}}$ after $\bar{u}_2^+ \ succ'!$, there exists an α-renaming $\bar{v}_1^+ \ succ'!$ of $\bar{t}_1^+ \ succ!$ such that

$$\Xi_b, \bar{\Xi}_0 \vdash {}_{o'}\!\downarrow \bar{u}_2^+ succ'! \preccurlyeq {}_{o'}\!\downarrow \bar{v}_1^+ succ'! \, , \tag{C.17}$$

for all objects o' from $[o_2']_{/_{\bar{\Xi}_2'}}$ (after $\bar{u}_2^+ succ'!$). Considering specifically the success-reporting clique $[o_1]$, we have $\Xi_b, \bar{\Xi}_0 \vdash {}_{o}\!\downarrow \bar{u}_2^+ \ succ'! \preccurlyeq {}_{o}\!\downarrow \bar{v}_1^+ \ succ'!$ for some renaming $\bar{v}_1 succ'!$ and for all objects of that clique. Since the label $succ'$ is *unique*, $\Xi_b, \bar{\Xi}_0 \vdash {}_{o}\!\downarrow \bar{u}_2^+ = {}_{o}\!\downarrow \bar{v}_1^+$ for all o of $[o_1]$, which can be strengthened to $\bar{\Xi}_0 \vdash {}_{o}\!\downarrow \bar{u}_2^+ = {}_{o}\!\downarrow \bar{v}_1^+$ since the type/class c_b is neither mentioned in $\bar{v}_1^+$ nor in $\bar{u}_2^+$. By dualizing we obtain

$$\Xi_0 \vdash {}_{o}\!\downarrow u_2^+ = {}_{o}\!\downarrow v_1^+ \, , \tag{C.18}$$

as required. □

[3] Component cliques from the dual perspective of $\Xi_b, \bar{\Xi}_0$, i.e., the cliques of the observer C_O.

Index

List of Figures

List of Tables

Listings