Theory Comput Syst (2008) 43: 322–361
DOI 10.1007/s00224-007-9047-0

Abstract Interface Behavior of Object-Oriented Languages with Monitors

Erika Ábrahám · Andreas Grüner · Martin Steffen

Published online: 3 October 2007

Abstract We characterize the observable behavior of multi-threaded, object-oriented components with *re-entrant monitors*. We show that a compositional approach leads to observable uncertainty wrt. monitor operations at the interface which we capture by may- and must-approximations for potential, resp. necessary lock ownership. The concepts are formalized in an object calculus. We show the soundness of the abstractions.

Keywords Formal semantics · Object oriented languages · Thread-based concurrency · Monitors · Open systems · Observable interface behavior

1 Introduction

1.1 Motivation

Compositionality is the key to describe and verify large systems. A compositional argument, that a program meets its specification, uses only the abstract descriptions or specifications of the program's constituent components and, in particular, must not refer to internal representation-dependent details of those components. Thus, for

E. Ábrahám
Albert-Ludwigs-University Freiburg, Freiburg, Germany
e-mail: eab@informatik.uni-freiburg.de

A. Grüner
Christian-Albrechts-University Kiel, Kiel, Germany
e-mail: ang@informatik.uni-kiel.de

M. Steffen (✉)
University of Oslo, Oslo, Norway
e-mail: msteffen@ifi.uio.no

compositional arguments it is crucial to separate clearly between the component's internal and its external behavior.

Now, which criteria should separate the externally visible from the internal behavior? An elegant, simple, and mathematically well-founded approach to that question is to take the observational standpoint: The starting point is the syntax and the operational semantics of a language; in general, this part is a priori given or at least straightforward. It is straightforward, as no internal details are abstracted away yet, and no questions of observability are yet involved. With the given syntax and internal semantics at hand, a component or *open* program is nothing else than a syntactical program fragment *interacting* with the syntactical rest of the program, its environment, context, or observer.

In this paper, we tackle the question of abstract interface behavior for concurrent object-oriented class-based languages with re-entrant *monitors*. Using an appropriate object calculus, the answer is given in the form of an operational semantics for components, distinguishing component internal steps from external steps which represent observable component-environment interactions occurring at the interface. A representation-independent, abstract account of the behavior of open programs is also necessary for compositional optimization of components: only if the two programs show the same external behavior it is guaranteed that one can replace one for the other without changing the interaction with any client code.

1.2 Approach and Contribution

Before embarking on the technical development starting in Sect. 2, we survey the approach in a little more detail. Below we discuss how to capture the legal system behavior at the border between a component and its environment. Afterwards, in Sect. 1.2.2, we sketch the particular problems in the chosen setting, i.e., when dealing with monitors in an object-oriented setting. Section 1.2.3 discusses the issue of full abstraction.

1.2.1 Capturing Open System Behavior

The observable interface behavior of an open program can be represented as message traces, i.e., sequences of component-environment interactions at the interface. Using some standard notation, this may be written as

$$C \stackrel{t}{\Longrightarrow} \acute{C}, \tag{1}$$

where t is the *trace* of interface actions by which C evolves into $\acute{C}$, potentially executing internal steps, as well, not recorded in t. Following the structural approach to operational semantics, the reduction of (1) represents the behavior of the component by a syntactic rewriting of the component C into $\acute{C}$, where C, resp. $\acute{C}$ contains the user-syntax as well as the run-time configurations of the component. It is customary to distinguish in the formulation of the operational semantics between user-syntax to represent the programs at compile time and *additional* syntactic material (the run-time syntax, not available to the programmer) to represent the configurations of the

component at run-time. The corresponding syntactic definitions are given in detail in Sect. 2.1.

We think, however, of the open program C not to act in isolation, it rather interacts with its environment. So consider $C \parallel E$, where E is C's environment, i.e., both together $C \parallel E$ form a *closed* program. Instead of considering unrestricted traces as in the judgment (1), we are interested in traces t where there *exists* an environment E such that

$$C \parallel E \stackrel{t}{\underset{\bar{t}}{\Longrightarrow}} \acute{C} \parallel \acute{E} \tag{2}$$

by which we mean: component C exhibits trace t and E produces the dual trace $\bar{t}$, both together canceling out to internal steps. In other words, our goal is to formulate an open semantics with the environment *existentially abstracted away.*

As there are countably infinite many possible environments E, the challenge is to capture in the semantics of the open system what is common to *all* those environments. This requires an abstract representation of all potential E of the judgment (2), in form of *assumptions* about the environment. This means, instead of providing an operational semantics formalizing reductions in the style of judgment (1), the semantics specifies the behavior of C under certain assumptions Ξ_E about the environment, where Ξ_E acts as an abstract representation of the environment. Following standard notation from logics, we do not write $\Xi_E \parallel C$, but rather $\Xi_E \vdash C$, such that the reductions will look like[1]

$$\Xi_E \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi}_E \vdash \acute{C}. \tag{3}$$

Clearly, an operational semantics in the style of (3) is (quite) more complex than a formulation which ignores the environments as in (1), since it requires an appropriate abstract characterization of the absent environment. Indeed, we take the characterization of the interface behavior one step further still: In the same way as (3) represents the environments abstractly as Ξ_E, a second step abstract away from the component C existentially, as well. This yields a formalization

$$\Xi \stackrel{t}{\Longrightarrow} \acute{\Xi}, \tag{4}$$

which describes the traces t which are possible *at all* at the interface between an arbitrary component and an arbitrary environment. We call such traces *legal.* The Ξ in judgment (4) represents now both the existentially abstracted environment E and the component C of judgment (2).

1.2.2 Multi-Threading Concurrency and Re-entrant Monitors

Above we discussed the observable interface behavior in a general framework. Given a specific language, the characterization of the allowed, legal interface behavior gives

[1]To avoid later confusion: The symbol Ξ used later does not only formalize assumptions about the environment, but also *commitments* of the component, to make the setting symmetric. The notation Ξ_E with subscript E is used only here for explanatory reasons.

insight into the semantical nature of the considered language and its features. This paper concentrates on the following features:

- *types and classes:* the languages are statically typed, and only well-typed programs are considered.
- *references:* each object carries a unique *identity*. New objects are dynamically allocated on the heap.
- *concurrency:* the mentioned languages feature concurrency based on *threads* (as opposed to processes or active objects [5]).
- *monitor synchronization:* objects can play the role of monitors [6, 12], guaranteeing that synchronized methods are executed mutually exclusive. Recursion—direct or indirect—via method call requires *re-entrant* monitors.

These key aspects of modern class-based object-oriented languages like *Java* [11] or $C^{\#}$ [9] are formalized in an abstract object calculus. The interface behavior is formulated in an assumption-commitment framework and based on three orthogonal abstractions:

- a static abstraction, i.e., the type system;
- an abstraction of the stacks of recursive method invocations, representing the recursive nature of method calls in a multi-threaded setting;
- finally as the main contribution, an abstraction of *lock ownership*.

The contribution of this paper over our previous work in this field (e.g., [3] dealing with deterministic, single-threaded programs, or [4] considering thread classes) is to capture re-entrant monitor behavior, the basic synchronization and mutex-mechanism of, e.g., multi-threaded *Java*.

In comparison with the mentioned work, the setting here is *simpler* in one respect: We disallow object instantiation across the interface here; of course, instantiation of objects from classes as such is supported, only not across the boundary between component and environment. The resulting semantical consequences of allowing such interaction have been investigated at length elsewhere, for instance in [26]. In a nutshell, such a framework makes it necessary to equip the interface behavior with an abstract, approximative representation of the heap, i.e., to represent the object's pointer structure, their "connectivity". This is characteristic of *class-based* object-oriented languages and a semantical consequence when considering instantiation of classes from an observational standpoint. This complication is conceptually orthogonal from the aspect on which we concentrate here, namely monitor locks. Indeed, it would be principally straightforward to combine the findings concerning abstract heap representations with the interface descriptions characterizing monitors. For sake of clarity, we ignore this heap aspect here by simply disallowing cross-border instantiation.

Incorporating monitors into the formal calculus is not only pragmatically motivated —after all, *Java* and similar languages offer monitor synchronization—but also semantically interesting, because the observable equivalences induced by a language offering synchronized methods and one without are incomparable.

Such a characterization of the abstract interface behavior is relevant and useful for the following reasons. Firstly: the set of traces according to judgment (3) is in general more restricted than the one obtained when ignoring the environments altogether.

This means, when *reasoning* about the trace-based behavior of C, for instance for the purpose of verification, with more precise knowledge of the possible traces we can carry out stronger arguments about C. Secondly, an application for a trace description is black-box testing, in that one describes the behavior of a component in terms of the interface traces and then synthesize appropriate test drivers from it. Obviously it makes no sense to specify interface behavior which is not possible, at all, since in this case one could not generate a corresponding tester. Currently we are developing a corresponding test language and tool. Finally, and not as the least gain, the formulation gives *insight* into the inherent semantical nature of the language, as the assumptions Ξ_E and the semantics captures the existentially abstracted environment behavior. For instance, one insight to be learned from [2, 3] and also from this paper is, that the presence of classes in the language necessitates an abstract representation of the heap as part of Ξ_E.

One main result of the paper, namely that the abstract trace semantics appropriately abstracts from the concrete, internal semantics, is formulated as a *soundness* result (cf. Lemma 3.25): If a component actually exhibits a trace according to judgment (1), then the abstract trace characterization of judgment (4) accepts the trace as possible.

1.2.3 Observational Equivalence and Full Abstraction

The observational approach has in particular been used to give a convincing answer to the question, when two programs are equivalent: Two components C_1 and C_2 are equivalent if they can not be discriminated in the following sense:

for *all* environments E, letting $E \parallel C_1$ and $E \parallel C_2$ run, one sees no difference,

where $E \parallel C$ means the closed program consisting of C and the "rest" E (the context, the observer, the environment). In general, one might not choose the parallel construct $\parallel$ as composition operation, and can define a context $\mathcal{C}[\_]$ abstractly as a "program with a hole"; in our setting, the context or environment, however, will be composed using $\parallel$.

This allows to define *observational equivalence* of two program fragments C_1 and C_2 as follows: $C_1 \equiv_{obs} C_2$ if $\mathcal{C}[C_1]$ and $\mathcal{C}[C_2]$ yield the same results, for all contexts, only that we have not clarified what it actually means, that two closed programs yield the same observational result. Indeed, different choices are possible here and have been investigated in the literature. The simplest possible external result or observation about a closed program is that it halts, 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 [19] 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. The important notion of testing equivalence has been introduced by de Nicola and Hennessy [8].

The approach gives a natural and easy definition of when two programs are equivalent, but does not tell what actually the denotation or meaning of a program *is*. The quantification over all possible contexts gives the contextual definition its strength and simplicity. It makes it hard to apply, however, 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* [18, 21]: 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. With the observational definition as reference, the denotational semantics is *sound*, if $C_1 \equiv_{\mathcal{D}} C_2$ implies $C_1 \equiv_{obs} C_2$. The inverse implication, hence "full" abstraction, corresponds to *completeness*.

In our setting, the denotation of an open program is, in first approximation, a set of *interface traces*, and indeed the development of the abstract interface behavior in terms of legal traces can be motivated to provide one corner stone for full abstraction. However, we do not address full abstraction in this paper. It is nonetheless worthwhile, to put the formalization of the abstract interface behavior presented here into perspective and point out what has been achieved (and what is missing) for full abstraction wrt., say, may testing equivalence.

As indicated, the notion of full abstraction is based on a comparison between an observational, contextual equivalence (such as may-testing equivalence) on the one hand, and a denotational equivalence (e.g. based on having the same set of interface traces) on the other hand. Such a comparison has two directions: the observational equivalence is taken as reference, and the denotationally given one must neither be too abstract nor too concrete, i.e, too discriminating. In general, completeness is the trickier direction. When only interested in soundness, one can enrich the denotations with as many details as wished, for instance, include internal implementation details into the interaction traces without losing soundness. By adding details to the semantics, the corresponding equivalence just gets unnecessarily discriminating, but stays sound, compared to the observational equivalence. Completeness, i.e., being not too abstract (but still containing enough discriminating power to remain sound) is harder. The task is to show that if two programs are observationally equal, they have the same set of traces (if we stick to that picture). Contrapositively: if two components have different traces, i.e., if there exists at least one trace that one component is able to exhibit but the other not, then there must be an observer or environment, that can distinguish those components. This renders the proof of completeness a constructive argument: given a (distinguishing) trace, *program* an environment in the given language that observes this trace. Clearly, this programming task can succeed only if the given trace is legal. The need for such a characterization for the completeness part of full abstraction is an additional motivation for the definition of the legal traces, as sketched in Sect. 1.2.1 and worked out later in Sect. 3.

Even if capturing the interface behavior is an important step towards completeness and full abstraction, two steps remain to be done for the full result. One is, the con-

structive proof itself, i.e., programming the observing environment for a given legal trace. This can be seen as the completeness of the legal-trace abstraction as provided in Sect. 3. The second missing piece is that simply taking the set of traces as semantics, as we pretended in slight simplification, will not do the job, it is too concrete. So on top of the characterization of legal traces, one needs to relax the definition by considering the traces only up-to certain equivalences, which capture the observational uncertainty inherent in the language. We are currently working on the details and will report on them in a subsequent publication. Working out observational equivalence and full abstraction is left for future work.

Overview The paper is organized as follows. Section 2 contains syntax and operational semantics of the calculus. Section 3 contains an independent characterization of the interface behavior of an open system, especially capturing the effects of lock ownership. Furthermore, it contains the basic soundness results of the abstractions. Section 4 concludes with related and future work.

2 A Multi-Threaded Calculus with Monitors

This section presents the calculus, which is based on a multi-threaded object calculus, similar to the one presented in [10] and in particular [13].

2.1 Syntax

The abstract syntax of the calculus is given in Table 1. Names n (see the clause for values v) are used to refer to classes, objects, and threads. In the text, we generally use o and its syntactic variants as names for objects, c for classes, and n for thread names and when being unspecific. A program is given by a collection of classes where a class $c[(O)]$ carries a name c and defines its methods and fields in O. An object $o[c, F, n]$ keeps a reference to the class c it instantiates, stores the current value of the fields or instance variables, and maintains a *lock* n, referring to the name of the thread holding the lock. The special name $\bot_{thread}$ (which is not a value) denotes that the lock is free. The ensemble of methods or method suite M is kept in the class. A method $\varsigma(self{:}c).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body t abstracted over the ς-bound "self" parameter and the formal parameters of the method [1]. We distinguish between synchronized and non-synchronized methods conventionally by superscripts l^s resp. l^u, and write just l when unspecific. The methods are stored in the classes, but the fields are kept in the objects, of course. For uniform treatment, the syntax represents fields as methods without parameters, except the self-parameter, and whose body is either a value or yet undefined. Immediately after instantiation, all fields carry the undefined reference $\bot_c$, where c is the (return) type of the field. Furthermore, the lock is free for new objects. Besides objects and classes, the dynamic configuration of a program contains named threads $n\langle t\rangle$ as active entities.

A thread t is basically a sequence of expressions, where the let-construct is used for sequencing and for local declarations.[2] Expressions include method calls $v.l(\vec{v})$,

[2]The sequential composition $t_1; t_2$ of two threads is syntactic sugar for $\mathsf{let}\, x{:}T = t_1\, \mathsf{in}\, t_2$, where x does not occur free in t_2.

Table 1 Abstract syntax

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

the creation of new objects $\mathsf{new}\, c$ where c is a class name, and *thread creation* $\mathsf{new}\langle t\rangle$. We use f for instance variables or fields and $l = \varsigma(s{:}T).\lambda().v$, resp. $\varsigma(s{:}T).\lambda().\bot_c$ for field variable definition. Field access is written as $v.l()$ and field update as $v'.l := \varsigma(s{:}T).\lambda().v$. By convention, we abbreviate the latter constructs by $l = v$, $l = \bot_c$, $v.l$, and $v'.l := v$. We will also use $v_\bot$ to denote either a value v or a symbol $\bot_c$ for being undefined. Note that the syntax does not allow to set a field back to undefined, using $v.l := \varsigma(s{:}T).\lambda().\bot_c$, resp., $v.l := \bot_c$, for short.

Apart from disallowing instantiation across the interface between component and environment, as mentioned shortly in the introduction, we impose the following two restrictions on the language: firstly, we disallow direct access (read or write) to fields across object boundaries. Secondly, we forbid that any occurrence of thread creation $\mathsf{new}\langle t\rangle$ contains a self-parameter, i.e., a name occurring bound by ς. The reason is that a new thread must start its life "outside" any monitor.

The available types are given in the following grammar:

$$T ::= B \mid \mathsf{None} \mid \mathit{thread} \mid [l{:}U, \ldots, l{:}U] \mid [(l{:}U, \ldots, l{:}U)] \mid n$$

$$U ::= T \times \cdots \times T \to T$$

Besides base types B if wished, the type *thread* denotes the type of thread names, and None represents the absence of a return value. The name n of a class serves as the type for the named instances of the class. Finally we need for the type system, i.e., as auxiliary type constructions, the type or interface of unnamed objects, written $[l_1{:}U_1, \ldots, l_k{:}U_k]$ and the interface type for classes, written $[(l_1{:}U_1, \ldots, l_k{:}U_k)]$. We write $\mathsf{Unit} \to T$ for $T_1 \times \cdots \times T_k \to T$ when $k = 0$.

2.2 Type System

The type system presented next characterizes the well-typed programs. The derivation rules are split into two sets: one for typing on the level of components, i.e., global configurations, and secondly one for their syntactic sub-constituents.

So Table 2, to start with, defines the typing on the level of global configurations, i.e., for "sets" of objects, classes, and threads. On this level, the typing judgments are

Table 2 Static semantics (components)

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

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

$$\frac{\Delta, o{:}c \vdash C : \Theta \qquad \Delta \vdash c : [\![\ldots]\!]}{\Delta \vdash \nu(o{:}c).C : \Theta}\ \text{T-NU}_o^1 \qquad \frac{\Delta \vdash C : \Theta, o{:}c \qquad \Theta \vdash c : [\![\ldots]\!]}{\Delta \vdash \nu(o{:}c).C : \Theta}\ \text{T-NU}_o^2$$

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

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

of the form

$$\Delta \vdash C : \Theta, \tag{5}$$

where Δ and Θ are *name contexts*, i.e., 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. Δ must contain at least all external names referenced by C and dually Θ mentions the names offered by C. For a pair Δ and Θ of assumption and commitment context to be *well-formed* we furthermore require that the domains of Δ and Θ are disjoint except for thread names.

The empty configuration is denoted by $\mathbf{0}$; it 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 the rule for parallel composition are merged disjointly, which is indicated by writing Θ_1, Θ_2 (analogously for the assumption contexts).

Remark 2.1 (Thread names and parallel composition) Note that T-PAR does not allow a thread name to occur on both sides of the parallel composition. The typing excludes terms of the form $n\langle t_1 \rangle \parallel n\langle t_2 \rangle$ as part of the component. Indeed, the operational semantics will not need to consider the behavior of the parallel composition of a thread n with another one of the same name.

The ν-binder hides the bound object or thread name inside the component (cf. the rules T-NU$_t$ resp., T-NU$_o^1$ and T-NU$_o^2$). In the T-NU-rules, we assume that the bound name o, resp. n is new to Δ *and* Θ. Also, in those rules, 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 one difference of let-bindings for variables and the introduction of names via the ν-operator: the language construct to introduce names is the new-operator, which opens a new local scope and a named component "running in parallel".

Let-bound variables are *stack* allocated and checked in a stack-organized variable context Γ. Names created by new are *heap* allocated and thus checked in a "parallel" context (cf. again the assumption-commitment rule T-PAR). The rules for named classes introduce the name of the class and its type into the commitment (cf. T-NCLASS); The code of the class $[(O)]$ is checked in an assumption context where the name of the class is available.

An instantiated object will be available in the exported context Θ by rule T-NOBJ. Running threads are treated similarly, except that they always possess the type None, which expresses that they do not return with a value.[3]

The last rule is a rule of *subsumption*. It expresses a very simple form of subtyping: we allow that an object respectively class contains *at least* the members which are required by the interface. This corresponds to width subtyping. Note, however, that each named object has exactly one type, namely its class.

Definition 2.2 (Subtyping) The relation $\leq$ on types is defined as identity for all types except for object interfaces where we have:

$$[(l_1{:}U_1, \ldots, l_k{:}U_k, l_{k+1}{:}U_{k+1}, \ldots)] \leq [(l_1{:}U_1, \ldots l_k{:}U_k)].$$

For well-formed name contexts Δ_1 and Δ_2 , we define in abuse of notation $\Delta_1 \leq \Delta_2$, if Δ_1 and Δ_2 have the same domain and additionally $\Delta_1(n) \leq \Delta_2(n)$ for all names.

The same definition is applied, of course, also for name contexts Θ, used for the commitments. The relations $\leq$ are obviously reflexive, transitive, and antisymmetric.

The typing rules of Table 3 formalize typing judgments for threads and objects and their syntactic sub-constituents. Besides assumptions about the provided 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 identical to the ones from [13] and [4]. We allow ourselves to write $\vec{T}$ and $\vec{v}$ for $T_1 \times \cdots \times T_k$ and $v_1, \ldots, v_k$ and similar abbreviations, where we assume that the number of arguments match in the rules. Different from the object-based setting are the ones dealing with objects and classes. Rule T-CLASS is the introduction rule for class types, the rule of instantiation of a class T-NEWC requires reference to a class-typed name. In the rule T-MEMB and T-FUPDATE we use 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 $T = [(\ldots, l{:}U, \ldots)]$. Note also that the deadlocking expression stop has every type.

[3]For the thread in T-NTHREAD, the type None can be generated by the atomic thread stop. In principle, a variable could have the type None, as well, but there are no values except variables of this type.

Table 3 Static semantics (2)

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

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

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

$$\frac{\Gamma;\Delta \vdash c : [\![\ldots, l : \mathsf{Unit} \to c', \ldots]\!]}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda().\perp_{c'} : c'}\ \text{T-Undef}$$

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

$$\frac{\Gamma;\Delta \vdash v : c \qquad \Gamma;\Delta \vdash c : T \qquad \Gamma;\Delta \vdash v' : T.l}{\Gamma;\Delta \vdash v.l := v' : c}\ \text{T-FUpdate}$$

$$\frac{\Gamma;\Delta \vdash c : [\![T]\!]}{\Gamma;\Delta \vdash \mathsf{new}\, c : c}\ \text{T-NewC} \qquad \frac{\Gamma;\Delta \vdash t : T}{\Gamma;\Delta \vdash \mathsf{new}\langle t\rangle : \mathit{thread}}\ \text{T-NewT}$$

$$\frac{}{\Gamma;\Delta \vdash \mathit{currentthread} : \mathit{thread}}\ \text{T-CurrT} \qquad \frac{\Gamma;\Delta \vdash e : T_1 \qquad \Gamma, x{:}T_1;\Delta \vdash t : T_2}{\Gamma;\Delta \vdash \mathsf{let}\, x{:}T_1 = e\, \mathsf{in}\, t : T_2}\ \text{T-Let}$$

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

$$\frac{\Gamma;\Delta \vdash v : c \qquad \Gamma;\Delta \vdash c : [\![\ldots, l{:}\mathsf{Unit} \to T, \ldots]\!] \qquad \Gamma;\Delta \vdash e_1 : T_2 \qquad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\, \mathsf{undef}(v.l())\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : T_2}\ \text{T-Cond}_\perp$$

$$\frac{}{\Gamma;\Delta \vdash \mathsf{stop} : T}\ \text{T-Stop} \qquad \frac{\Gamma(x) = T}{\Gamma;\Delta \vdash x : T}\ \text{T-Var} \qquad \frac{\Delta(n) = T}{\Gamma;\Delta \vdash n : T}\ \text{T-Name}$$

2.3 Operational Semantics

As the typing system, the operational semantics, is given in two stages. Section 2.3.1 starts with component-internal steps, i.e., those definable without reference to the environment. In particular, the steps have no observable external effect and are formulated independently of the assumption and commitment contexts. The external steps presented in Sect. 2.3.2, define the interaction of the component with the environment. The external steps are defined in reference to assumption and commitment contexts. The static part of the contexts corresponds to the static type system from Sect. 2.2 on component level and takes care that, e.g., only well-typed values are received from the environment.

Table 4 Internal steps

$n\langle \mathsf{let}\, x{:}T = v\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle t[v/x]\rangle$	RED
$n\langle \mathsf{let}\, x_2{:}T_2 = (\mathsf{let}\, x_1{:}T_1 = e_1\, \mathsf{in}\, e)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x_1{:}T_1 = e_1\, \mathsf{in}\ (\mathsf{let}\, x_2{:}T_2 = e\, \mathsf{in}\, t)\rangle$	LET
$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, v = v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x{:}T = e_1\, \mathsf{in}\, t\rangle$	COND_1
$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, v_1 = v_2\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x{:}T = e_2\, \mathsf{in}\, t\rangle$	COND_2
$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, \mathsf{undef}(\varsigma(s{:}c)\lambda().\bot_{c'})\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x{:}T = e_1\, \mathsf{in}\, t\rangle$	$\text{COND}_1^{\bot}$
$n\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, \mathsf{undef}(\varsigma(s{:}c)\lambda().v)\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x{:}T = e_2\, \mathsf{in}\, t\rangle$	$\text{COND}_2^{\bot}$
$n\langle \mathsf{let}\, x{:}T = \mathsf{stop}\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{stop}\rangle$	STOP
$c[\![F, M]\!] \parallel n\langle \mathsf{let}\, x{:}c = \mathsf{new}\, c\, \mathsf{in}\, t\rangle \rightsquigarrow$ $c[\![F, M]\!] \parallel \nu(o{:}c).(o[c, F, \bot_{thread}] \parallel n\langle \mathsf{let}\, x{:}c = o\, \mathsf{in}\, t\rangle)$	NEWO_i
$c[\![F, M]\!] \parallel o[c, F', n'] \parallel n\langle \mathsf{let}\, x{:}T = o.l^u(\vec{v})\, \mathsf{in}\, t\rangle \rightsquigarrow$ $c[\![F, M]\!] \parallel o[c, F', n'] \parallel n\langle \mathsf{let}\, x{:}T = M.l^u(o)(\vec{v})\, \mathsf{in}\, t\rangle$	CALL_i^u
$c[\![F, M]\!] \parallel o[c, F', \bot_{thread}] \parallel n\langle \mathsf{let}\, x{:}T = o.l^s(\vec{v})\, \mathsf{in}\, t\rangle \rightsquigarrow$ $c[\![F, M]\!] \parallel o[c, F', n] \parallel n\langle \mathsf{let}\, x{:}T = M.l^s(o)(\vec{v})\, \mathsf{in}\, release(o); t\rangle$	$\text{CALL}_{i_1}^s$
$c[\![F, M]\!] \parallel o[c, F', n] \parallel n\langle \mathsf{let}\, x{:}T = o.l^s(\vec{v})\, \mathsf{in}\, t\rangle \rightsquigarrow$ $c[\![F, M]\!] \parallel o[c, F', n] \parallel n\langle \mathsf{let}\, x{:}T = M.l^s(o)(\vec{v})\, \mathsf{in}\, t\rangle$	$\text{CALL}_{i_2}^s$
$o[c, F, n] \parallel n\langle \mathsf{let}\, x{:}T = release(o)\, \mathsf{in}\, t\rangle \xrightarrow{\tau} o[c, F, \bot_{thread}] \parallel n\langle t\rangle$	RELEASE
$o[c, F', n'] \parallel n\langle \mathsf{let}\, x{:}T = o.l()\, \mathsf{in}\, t\rangle \xrightarrow{\tau}$ $c[\![O]\!] \parallel o[c, F'] \parallel n\langle \mathsf{let}\, x{:}T = F'.l(o)(\vec{v})\, \mathsf{in}\, t\rangle$	FLOOKUP
$o[c, F, n'] \parallel n\langle \mathsf{let}\, x{:}T = o.l := v\, \mathsf{in}\, t\rangle \xrightarrow{\tau} o[c, F.l := v, n'] \parallel n\langle \mathsf{let}\, x{:}T = o\, \mathsf{in}\, t\rangle$	FUPDATE
$n\langle \mathsf{let}\, x{:}T = currentthread\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle \mathsf{let}\, x{:}T = n\, \mathsf{in}\, t\rangle$	CURRENTTHREAD
$n_1\langle \mathsf{let}\, x{:}T = \mathsf{new}\langle t\rangle\, \mathsf{in}\, t_1\rangle \rightsquigarrow \nu(n_2{:}T).(n_1\langle \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t_1\rangle \parallel n_2\langle t\rangle)$	NEWT

2.3.1 Internal Steps

Table 4 contains the internal reduction steps (the ones for conditionals, sequencing via let, thread creation, etc., are straightforward), distinguishing between confluent steps (i.e., steps not leading to race conditions), written $\rightsquigarrow$, and other internal transitions, written $\xrightarrow{\tau}$. The first 5 rules deal with the basic sequential constructs, all as $\rightsquigarrow$-steps, and where in COND_2, $v_1 \neq v_2$ is assumed. The basic evaluation mechanism is substitution (cf. rule RED). Note that the rule 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 which makes the reduction strategy deterministic. The stop-thread terminates for good, i.e., the rest of the thread will never be executed (cf. rule STOP).

The step 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. The lock of a new object is free and thus initialized with $\bot_{thread}$. 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.

The CALL_i-rules treat internal method calls, i.e., a call to an object contained in the configuration, where for synchronized methods, $\text{CALL}_{i_1}^s$ takes the free lock and

Table 5 Structural congruence

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

Table 6 Reduction modulo congruence

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

adds a release-statement at the end of the method body. Rule $\textsc{Call}^{s}_{i_2}$ describes reentrant calls. In the call-steps, $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when the method suite $[M]$ equals $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$.

The rule $\textsc{Call}^{u}_{i}$ deals with non-synchronized methods, in which case the lock is ignored. Field access is formalized by FLOOKUP. Note that the step is a $\xrightarrow{\tau}$-step, not a confluent one, as it accesses the instance state of an object. The same holds for field update in rule FUPDATE, where $[c, (l_1 = f_1, \ldots, l_k = f_k, f = v').f := v, n]$ stands for $[c, l_1 = f_1, \ldots, l_k = f_k, f = v, n]$. 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, *names* to objects may well be exported. The above reduction relations 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 5 where in the fourth axiom, n does not occur free in C_1. The congruence relation is imported into the reduction relations in Table 6. Note that all syntactic entities are always tacitly understood modulo α-conversion.

2.3.2 *External Steps*

The external steps of the operational semantics describe the interactions between a component and its environment. They are given in terms of a labeled transition system, where the labels represent the corresponding interaction:

$$\gamma ::= n\langle call\ o.l(\vec{v})\rangle \mid n\langle \mathsf{return}(v)\rangle \mid \nu(n{:}T).\gamma \quad \text{basic labels}$$

$$a ::= \gamma? \mid \gamma! \quad \text{receive and send labels}$$

A component exchanges information with the environment via *calls* and *returns*. Note that there are no separate external labels for object instantiation as we have

forbidden cross-border instantiation, i.e., we do not consider the situation that the environment instantiates classes of the component and vice versa. In the labels of the transitions, n is the thread that issues the call or returns from the call. Besides that, a thread name may appear as an argument of a method call or as a return value. Scope extrusion of a name across the interface is indicated by the ν-binder. Given a label $\nu(\Xi).\gamma'$ where Ξ is a name context such that $\nu(\Xi)$ abbreviates a sequence of single $\nu(n{:}T)$ 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 and refer to it by $\lfloor\gamma\rfloor$. Furthermore, *thread*(γ) denotes the thread of the label. The definitions are used analogously for send and receive labels. Note that for incoming labels, Ξ contains only bindings to environment objects and thread names, as the environment cannot create component objects; dually for outgoing communication. We write shortly γ_c for call and γ_r for return labels.

The external semantics is formalized as labeled transitions between judgments of the form

$$\Delta, \Sigma \vdash C : \Theta, \Sigma, \tag{6}$$

where Δ, Σ represent the *assumptions* about the environment of the component C and Θ, Σ the *commitments*. The assumptions require 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[4] $\Delta, \Sigma \vdash C : \Theta, \Sigma$ that Δ, Σ, and Θ are pairwise disjoint. Thus, the transitions are of the following form:

$$\Delta, \Sigma \vdash C : \Theta, \Sigma \to a\acute{\Delta}, \acute{\Sigma} \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}.$$

The assumption context Δ, Σ can be seen as an abstraction of the (not-present) environment.

Notation 2.3 We abbreviate the triple of name contexts Δ, Σ, Θ as Ξ. Furthermore we understand $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta}$ as $\acute{\Xi}$, etc.

The steps of the operational semantics for open systems checks the *static* assumptions, i.e., whether at most the names actually occurring in the core of the label are mentioned in the ν-binders of the label, and whether the transmitted values are of the correct types. This is covered in the following definition.

Definition 2.4 (Well-formedness and well-typedness of a label) We call a label $a = \nu(\Xi).\lfloor a\rfloor$ *well-formed,* written $\vdash a$, if $dom(\Xi) \subseteq fn(\lfloor a\rfloor)$ and if Ξ is a well-formed

[4]The judgment from (6) is the same as used for typing in (5), only that here, by convention, we explicitly mention the binding part for thread names as Σ to stress the mentioned invariant.

Table 7 Typechecking labels

$$\frac{;\acute{\Xi} \vdash \vec{v} : \vec{T} \quad a = n\langle call\ o_r.l(\vec{v})\rangle?}{\acute{\Xi} \vdash a : \ \vec{T} \to \_}\ \text{LT-CALLI} \qquad \frac{;\acute{\Xi} \vdash v : T \quad a = n\langle return(v)\rangle?}{\acute{\Xi} \vdash a : \ \_ \to T}\ \text{LT-RETI}$$

name-context for (object and thread) names, i.e., no name bound in Ξ occurs twice. The assertion

$$\acute{\Xi} \vdash o.l? : \vec{T} \to T \tag{7}$$

("an incoming call of the method labeled l in object o expects arguments of type $\vec{T}$ and gives back a result of type T") is given by the following rule, i.e., implication:

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

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, the full $\acute{\Xi}$ is consulted, since the argument types $\vec{T}$ or the result type T may refer to both component and environment classes. For outgoing calls, $\acute{\Xi} \vdash o.l! : \vec{T} \to T$ 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 expected type $\vec{T}$, resp., T, and relative to the name context $\acute{\Xi}$ is asserted by

$$\acute{\Xi} \vdash a : \vec{T} \to \_ \quad \text{resp.,} \quad \acute{\Xi} \vdash a : \_ \to T, \tag{8}$$

as given by Table 7. We use $\Xi \vdash a : wt$ as notation to assert well-typedness.

Besides *checking* whether the assumptions are met before a transition, the contexts are *updated* by a transition step, i.e., extended by new names, whose scope extrudes. All external transitions may exchange *bound* names in the label, i.e., bound references to objects and threads, but not to classes since class names cannot be communicated. For the binding part $\Xi' = \Delta', \Sigma', \Theta'$ of a label $\nu(\Xi').\gamma$, we distinguish references to existing objects and threads whose scope extrudes across the border. For incoming communication, with the binding part $\Xi' = \Delta', \Sigma'$, the bindings Δ' and Σ' are object references respectively thread names transmitted by *scope extrusion*. For object references, the distinction is based on the class types which are never transmitted. In the incoming step, Δ' extends the assumption context Δ and Σ' extends the assumption and the commitment context. For outgoing communication, the situation is dual. Cf. Definition 2.5.

Definition 2.5 (Context update) For a name context Ξ and an incoming label $a = \nu(\Xi').\lfloor a \rfloor$ where n is a thread name s.t. $\Xi' \vdash n$, we define $\acute{\Xi} = \Xi + a$ as:

$$\acute{\Theta} = \Theta + \Theta', \qquad \acute{\Delta} = \Delta + (\Delta', \odot_n), \quad \text{and} \quad \acute{\Sigma} = \Sigma + \Sigma'.$$

In case $\Xi' \not\vdash n$, i.e., the thread is not new to the component, the summand $\odot_n$ is omitted. We write $\Xi + a$ for the update of Ξ. The update for outgoing communication is defined dually.

In the definition, the special symbol $\odot_n$ is used to remember whether a new thread n starts its life at the component side or at the environment side. 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 known both at the environment and the components started on exactly one side. Hence, in the shown situation in Definition 2.5 of an *incoming* communication, the thread with the new name n has its origin in the environment. The information about $\odot_n$ is important since in the situation where, e.g., $\Delta \vdash \odot_n$, i.e., thread n starts in the environment, the first interfaces interaction of n must not only be a call, but it must be an *incoming* call. The above definition assumes that thread names are not communicated as arguments in method calls across the interface, i.e., the only way a new thread name becomes known at the interface is that the thread itself actively crosses the border. It is straightforward to extend the definitions to also cover the possibility that a new thread gets known at the interface by communicating the name as argument of a method call. See for instance [4] where this has been considered for a calculus featuring thread classes (but no monitors).

The operational rules of Table 8 use two additional expressions

$$\textit{blocks} \quad \text{and} \quad \textit{returns } v.$$

The three CALLI-rules deal with incoming calls. For all three cases, the contexts are *updated* to $\acute{\Xi}$ to include the information concerning new objects and threads. Furthermore, it is *checked* whether the label is type-correct and that the step is possible according to the (updated) assumptions $\acute{\Xi}$. In the rules, $fn(\lfloor a \rfloor)$ refers to the free names of $\lfloor a \rfloor$ (which equal $names(\lfloor a \rfloor)$).

The three rules for incoming calls deal with three different situations as to when an incoming call may happen: A call of a thread which is new to the component plus two different situations, where the name of the calling thread is already known in the component.

The first call rule CALLI_0 deals with the situation, that the thread n enters the component for the first time. This is assured by the premise $\Sigma' \vdash n$, where Σ', according to our conventions, is the part of the bindings Ξ' transmitted boundedly, which is responsible for thread names. The last three premises (which are identical for the other two CALLI-rules, as well) assure well-formedness of the label and well-typedness of the transmitted values. Additionally, the context Ξ is updated to $\acute{\Xi}$ by the information about new names transmitted via label a.

For reentrant method calls (cf. rule CALLI_1), the thread is blocked, i.e., it has left the component previously via an outgoing call. Rule CALLI_2 treats likewise a situation, where the thread is already contained in the component nonetheless, but all method calls of the thread have been answered. As a consequence, the component contains the entity $n\langle\mathsf{stop}\rangle$. As the thread n must have crossed the border before, the marker for its creator $\odot_n$ must be contained in either Δ or in Θ. The premise $\Delta \vdash \odot_n$ assures that n had started its life on the environment side. This bit of information is important as otherwise one could mistake the code $n\langle\mathsf{stop}\rangle$ for the code of a (deadlocked) outgoing call.

Outgoing calls are dealt with in rule CALLO. To distinguish the situation from component-internal calls, the receiver must be part of the environment, expressed by $\Delta \vdash o_r$. Starting with a well-typed component, there is no need in re-checking now

Table 8 External steps

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

$$\frac{a=\nu(\Xi').\ n\langle call\ o_r.l(\vec{v})\rangle?\quad t_{blocked}=\text{let } x'{:}T'=blocks\text{ in } t\quad \acute{\Xi}\vdash o_r.l?:\vec{T}\to T\quad \acute{\Xi}=\Xi+a\quad \acute{\Xi}\vdash\lfloor a\rfloor:\vec{T}\to\_}{\Xi\vdash\nu(\Xi_1).(C\parallel n\langle t_{blocked}\rangle)\xrightarrow{a}\acute{\Xi}\vdash\nu(\Xi_1).(C\parallel n\langle \text{let } x{:}T=o_r.l(\vec{v})\text{ in } returns\ x;t_{blocked}\rangle)}\ \textsc{CalliI}_1$$

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

$$\frac{a=\nu(\Xi').\ n\langle call\ o_r.l(\vec{v})\rangle!\quad \Xi'=fn(\lfloor a\rfloor)\cap\Xi_1\quad \acute{\Xi}_1=\Xi_1\setminus\Xi'\quad \acute{\Delta}\vdash o_r\quad \acute{\Xi}=\Xi+a}{\Xi\vdash\nu(\Xi_1).(C\parallel n\langle \text{let } x{:}T=o_r.l(\vec{v})\text{ in } t\rangle)\xrightarrow{a}\acute{\Xi}\vdash\nu(\acute{\Xi}_1).(C\parallel n\langle \text{let } x{:}T=blocks\text{ in } t\rangle)}\ \textsc{CalloO}$$

$$\frac{a=\nu(\Xi').\ n\langle return(v)\rangle?\quad \acute{\Xi}=\Xi+a\quad \acute{\Xi}\vdash\lfloor a\rfloor:\_\to T}{\Xi\vdash\nu(\Xi_1).(C\parallel n\langle \text{let } x{:}T=blocks\text{ in } t\rangle)\xrightarrow{a}\acute{\Xi}\vdash\nu(\Xi_1).(C\parallel n\langle t[v/x]\rangle)}\ \textsc{RetI}$$

$$\frac{a=\nu(\Xi').\ n\langle return(v)\rangle!\quad \Xi'=fn(\lfloor a\rfloor)\cap\Xi_1\quad \acute{\Xi}_1=\Xi_1\setminus\Xi'\quad \acute{\Xi}=\Xi+a}{\Xi\vdash\nu(\Xi_1).(C\parallel n\langle \text{let } x{:}T=returns\ v\text{ in } t\rangle)\xrightarrow{a}\acute{\Xi}\vdash\nu(\acute{\Xi}_1).(C\parallel n\langle t\rangle)}\ \textsc{RetO}$$

that only values of appropriate types are handed out, as the operational steps preserve well-typedness ("subject reduction").

Note that the steps of Table 8 are independent of *lock manipulations,* e.g., an incoming call, which hands over the message via one of the CALLI-rules does not attempt to obtain the lock; this is done by the internal steps from Table 4. This *decouples* the responsibilities of component and environment in the spirit of the assumption/commitment set-up. Whether an incoming call can be sent by the environment depends *only* on the past interface interaction and the environment, *but not* on an internal state of the component!

The rules RETO and RETI deal with the return actions. The return steps work similar as the calls. Returns are simpler than calls in that only one value is communicated, not a tuple (and 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 $\nu(\Xi')$ as before.

Finally, we characterize the *initial* situation. Initially, we assume that the component contains at most one thread and no objects. More precisely, assume that $\Xi_0\vdash C_0$ is the initial judgment. Then C_0 contains no objects. Concerning threads: if $\Delta_0\vdash\odot$, then C neither contains a thread. If otherwise, $\Theta_0\vdash\odot$, C contains exactly one thread and is of the form $C\equiv\nu(n{:}thread).C'$. In particular, for the context Ξ_0 it means, that it contains only class names, but neither thread names nor object names. These conditions imply, that initially only calls are possible, but no returns. If initially, $\Delta_0\vdash\odot$, i.e., the initial thread starts in the environment, then only $\textsc{CalliI}_0$ is applicable, specializing the premise $\Delta\vdash o$ to $\Delta\vdash\odot$. If initially $\Theta_0\vdash\odot$, then only CALLO is applicable.

3 Interface Behavior

Next we characterize the possible ("legal") interface behavior as interaction traces between component and environment. "Half" of the work has been done already in the definition of the external steps in Table 8: For incoming communication, for which the environment is responsible, the assumption contexts were used to check whether they originate from a realizable environment. Concerning the reaction of the component, no such checks were necessary. After all, the code of the program is given; so the *reaction* of the component is not only realizable, but a fortiori "realized". To characterize when a given trace is *legal,* we need to require that the behavior of the component side, i.e. the outgoing communication, adheres to the dual discipline we imposed on the environment in the definition of the semantics. Now, we analogously abstract away from the program code, rendering the situation symmetric.

The calls and returns of each thread must be "parenthetic", i.e., each return must have a prior matching call, and we must take into account whether the thread is resident inside the component or outside. In particular, we must take into account restrictions due to the fact that the method bodies are executed in *mutual exclusion* wrt. individual objects.

Remark 3.1 (Atomic communication) For the operational semantics of Sect. 2.3, the lock-taking is part of the *internal* steps (cf. the $\textsc{Call}_i^s$-rules). The handing-over of the call at the interface and the actual entry into the synchronized method body is non-atomic; in other words: at the interface, objects are *input-enabled.*

An alternative scheme would be *atomic* lock grabbing, i.e., the lock is atomically taken by the interface interaction. This would simplify the logical characterization as to when a lock is guaranteed to be taken resp. free, based on the interface trace, because the uncertainty of observation as to when the lock is actually taken, is then gone: When an outgoing call is performed, e.g., it is guaranteed that the lock is taken at that very point.

This, however, would contradict the clean separation of concerns between the *assumption* and the *commitment* contexts. The assumption contexts represent the (worst-case) abstraction of the environment, and dually the commitment over-approximates the actual situation of the component. So conceptually, for incoming communication, the assumption contexts are consulted to check, whether there is exists an realizable environment responsible for that step (and dually for outgoing communication). With atomic locking, the enabledness of an *incoming* call would depend on the commitment context, and dually *outgoing* communication on the assumptions about the environment. This reverses the two roles of assumptions and commitments, at least wrt. lock-availability—all other aspects such as type checking, connectivity, etc. remain—and thus breaks the clean separation of responsibilities in the semantics, rendering it less compositional.

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

$$\Xi \vdash r \rhd s : \mathit{trace} \tag{9}$$

stipulating that under assumptions Δ, Σ and with the commitments Θ, Σ, the trace s is legal (remember the conventions from Notation 2.3).

Table 9 Balance (for one thread)

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

$$\frac{\vdash s_1 : balanced^{+} \qquad \vdash s_2 : balanced^{+} \qquad s_1, s_2 \neq \epsilon}{\vdash s_1\ s_2 : balanced^{+}}\ \text{B-II}$$

$$\frac{\vdash s_1 : balanced^{-} \qquad \vdash s_2 : balanced^{-} \qquad s_1, s_2 \neq \epsilon}{\vdash s_1\ s_2 : balanced^{-}}\ \text{B-OO}$$

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

Roughly, the assertions used in the operational semantics are grouped into those for static typing and those for connectedness. Here, without the code of the program, we need an additional assertion concerning the balance of calls and returns ("enabledness"). In the operational semantics, such an assertion was not even needed for the behavior of the environment, since, for instance, an incoming return step of a thread is possible only when the thread is blocked. Thus the program syntax takes care that calls and returns happen only in a well-balanced manner. Without code, we need an independent characterization.

3.1 Balance Conditions

We start with auxiliary definitions concerning the parenthetic nature of calls and returns. Starting from an initial configuration, the operational semantics from Sect. 2.3 assures strict alternation of incoming and outgoing communication and additionally that there is no return without matching prior call.

Definition 3.2 (Balance) Let $s \downarrow_n$ be the projection of trace s onto thread n. The balance of a thread n in a sequence s of labels is given by the rules of Table 9. We write $\vdash s : balanced^{n}$ if $\vdash s : balanced_n^{+}$ or $\vdash s : balanced_n^{-}$. We call a (not necessarily proper) prefix of a balanced trace *weakly balanced.* We write $\vdash s : wbalanced_n^{+}$ if the trace is weakly balanced in n, i.e., if the projection of the trace on n is weakly balanced, and if the last label is an incoming communication or if $s \downarrow_n$ is empty; dually for $\vdash s : wbalanced_n^{-}$. The function *pop* (on the projection of a trace onto a thread n) is defined as follows:

1. $pop\ s = \bot$, if s is balanced in n.
2. $pop\ (s_1\ a\ s_2) = s_1\ a$ if $a = \gamma_c?$ and s_2 is $balanced_n^{+}$.
3. $pop\ (s_1\ a\ s_2) = s_1\ a$ if $a = \gamma_c!$ and s_2 is $balanced_n^{-}$.

We use $pop\ n\ r$ for $pop\ (r \downarrow_n)$.

To be explicit, we refer to a balanced trace also as strongly balanced.

Note that the communication labels alone do not contain enough information to determine their source and target. For call labels $\nu(\Xi).n\langle call\ o.l(\vec{v})\rangle$, only the target of the communication, the callee o is contained, the caller remains anonymous. This is justified by the fact that the callee does not get hold of the identity of the caller. The identity of the caller can therefore not be observed and should thus not be mentioned in the interface behavior. Return labels $\nu(\Xi).n\langle \mathsf{return}(\vec{v})\rangle$ do not mention any communication partner. However, even without being explicitly mentioned, the communication partners are determined by the communication history. For instance, the source of a return is target of the matching call. For a call it is assured that it leaves the same clique that the previous communication, call or return, has entered.

Based on a weakly balanced history, we defined the source and target of a communication event at the end of a trace with the help of the function *pop*.[5]

Definition 3.3 (Sender and receiver) Let $r\ a$ be the non-empty projection of a balanced trace onto the thread n. Sender and receiver of label a after history r are defined by mutual recursion and pattern matching over the following cases:

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

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

For $\gamma_c?$ resp. $\gamma_r?$, the definition is dual.

Note that source and target are well-defined. In particular, the recursive definition terminates. Furthermore the weak balance of the argument guarantees that the call of *pop* yields a well-defined result and that the case distinction is exhaustive.

$\Delta, \Sigma \vdash r \rhd a : \Theta, \Sigma$ asserts that after r, the action a is enabled. Input enabledness checks 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 3.4 (Enabledness) Given $\gamma = \nu(\Xi).n\langle call\ o_r.l(\vec{v})\rangle$. Then call-*enabledness* of γ after history r and in the contexts Δ, Σ and Θ, Σ is defined as:

$$\Delta, \Sigma \vdash r \rhd \gamma? : \Theta, \Sigma \text{ if } \begin{array}{l} pop\ n\ r = \bot \\ pop\ n\ r = r'\gamma'! \end{array} \text{ and } \Delta \vdash \odot_n \text{ or} \tag{10}$$

$$\Delta, \Sigma \vdash r \rhd \gamma! : \Theta, \Sigma \text{ if } \begin{array}{l} pop\ n\ r = \bot \\ pop\ n\ r = r'\gamma'? \end{array} \text{ and } \Theta \vdash \odot_n \text{ or} \tag{11}$$

[5]Since we apply the definition onto the projection of a trace onto a thread, we omit in the function the thread name as parameter.

For return labels $\gamma = \nu(\Xi).n\langle \mathsf{return}(v)\rangle$, $\Xi \vdash r \rhd \gamma!$ abbreviates $pop\ \ n\ \ r = r'\nu(\Xi').n\langle call\ o_2.l(\vec{v})\rangle?$, and dually for incoming returns $\gamma?$.

We also say, the thread is *input-call enabled* after r if $\Delta, \Sigma \vdash r \rhd \gamma_c? : \Theta, \Sigma$ for some incoming call label $\gamma_c?$, 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.

We further combine enabledness and determining sender and receiver (cf. Definitions 3.4 and 3.3) into the notation

$$\Xi \vdash r \rhd o_s \xrightarrow{a} o_r. \tag{12}$$

3.2 Side Conditions for Monitors

Next we address the restrictions imposed by the fact that the methods are *synchronized.* We assume in the following that *all* methods are synchronized, unless stated otherwise. We proceed in two stages. The first step in Sect. 3.2.1 concentrates on individual threads: given the interaction history of a single thread, we present two abstractions, one characterizing situations where the thread *may* hold the lock of a given object, and a second one where, independent of the scheduling, the thread *must* hold the lock. The second step in Sect. 3.2.2 takes a global view, i.e., considers all threads, to characterize situations in a trace which are (in-)consistent with the fact that objects act as monitors. The formalization is based on a *precedence* or *causal* relation of events of the given trace. This precedence relation formalizes three aspects that regulate the possible orderings of events in a trace:

mutual exclusion: If a thread has taken the lock of a monitor, interactions of other threads with that monitor must either occur *before* the lock is taken, or *after* it has been released again.

data dependence: no value (unless generated new) can be transmitted before it has been received.

control dependence: within a single thread, the events are linearly ordered.

The formalization of mutual exclusion is complicated by the fact that the locks are not taken atomically, i.e., we often do not have *immediate* information when the lock is taken and relinquished. Instead we must work with the may- and must-approximations calculated in Sect. 3.2.1 below. This uncertainty of observation influences also data dependence: The point it time where a value is "*received*" is not when it is handed over at the interface, what counts in this respect is when the value enters the monitor.

3.2.1 Lock Ownership

We start by characterizing when, given a history of interaction of a single thread, it *may* own the lock of an object. The "may"-uncertainty is due to the fact that the actual lock manipulation is separated by the corresponding visible interface interaction by some internal i.e., non-observable reduction steps.

Table 10 Potential lock ownership for Θ-locks

$$\frac{\vdash s_2 : \mathit{balanced} \qquad s_2 \neq \epsilon \qquad \Xi \vdash s_1 : \Diamond o}{\Xi \vdash s_1\ s_2 : \Diamond o}\ \text{M-}\Diamond$$

$$\frac{\mathit{receiver}(s\ \gamma_c?) = o}{\Xi \vdash s\ \gamma_c? : \Diamond o}\ \text{M-I}\Diamond_1 \qquad \frac{\mathit{receiver}(s\ \gamma_c?) \neq o \quad \Xi \vdash s : \Diamond o}{\Xi \vdash s\ \gamma_c? : \Diamond o}\ \text{M-I}\Diamond_2$$

$$\frac{\Xi \vdash s\ : \Diamond o}{\Xi \vdash s\ \gamma_c! : \Diamond o}\ \text{M-O}\Diamond$$

Definition 3.5 (May lock ownership) Given a sequence s of interactions of a single thread and a component object o, the judgment $\Xi \vdash s : \Diamond o$ ("after s, the thread of s may own the lock of o.") is given by the rules of Table 10. For environment locks, i.e., when o is an environment object, the definition is dual.

Observing that $\Xi \vdash t : \Diamond_n o$ is decidable (Lemma 3.11 below) we consider $\Xi \vdash t : \Box_n o$ as boolean predicates and write $\Xi \vdash t : \neg\Diamond_n o$ for $\Xi \nvdash t : \Diamond_n o$ (and later analogously for the must-predicate $\Box$).

Rule M-$\Diamond$ states that a balanced tail s_2 can be ignored, lock-wise. To assure that the premise is invoked on a proper prefix of the trace in the conclusion, we insist that s_2 is not the empty trace. The two M-I$\Diamond$-rules deal with incoming calls, depending on the receiver of the communication (remember that we use γ_c to refer to call labels and γ_r for return labels). If the call concerns the object o in question, the thread may own the lock afterwards. So this is an "introduction rule" for $\Diamond$-information. Remember that the receiver of a call γ_c is the object mentioned in the label (cf. Definition 3.3). If the receiver is distinct from o (cf. rule M-I$\Diamond_2$), the thread may own the lock of o, if that was the case already before the call.[6] Note that we do not have a corresponding rule for incoming *return* labels. Intuitively it means that an incoming return does not affect the information that the thread may own a given component lock. Since the same remark applies to the must-relation, discussed below, one can summarize that incoming returns do not carry any information wrt. ownership of Θ-locks. An outgoing call finally does not affect the $\Diamond$-information,[7] i.e., if a thread may own a lock before the outgoing call, it may do so afterwards (cf. rule M-O$\Diamond$).

Example 3.6 ($\Diamond$-predicate) We illustrate the meaning of the $\Diamond$-predicate on a very simple example. The example is in particular intended to avoid possible misconceptions what "potential" lock ownership means. We concentrate on component locks as opposed to environment locks, and it is enough to consider one single thread. Consider the following trace consisting of only one incoming call:

$$t = t'\ \gamma_c? = t'\ n\langle \mathit{call}\ o_r.l(o)\rangle?. \tag{13}$$

[6]The premise $\mathit{receiver}(\gamma_c) \neq o$ can be omitted. It is useful, however, to separate M-I$\Diamond_1$ from M-I$\Diamond_2$.

[7]This is in contrast to the $\Box$-knowledge. An outgoing call turns the knowledge that a thread may hold a lock into the stronger assertion that it now *must* hold the lock (cf. rule M-O$\Box_2$ from Table 11 below).

The receiving object o_r is a component object and assume that the locks of both o and o_r are free before the call occurs, i.e., after t'. According to the rules of Table 10 we have

$$\Xi \vdash t : \Diamond o_r \quad \text{and} \quad \Xi \vdash t : \neg\Diamond o \tag{14}$$

where Ξ is some appropriate initial context, left unspecified in this example. So, how do we interpret the two assertions of (14)? Well, $\Xi \vdash t : \Diamond o$ does *not* assert that there exists a component for which there is an execution such that the thread holds the lock of o. This interpretation would be consistent with the assertion for o_r on the left-hand side of (14). But applying this interpretation to $\Xi \vdash t : \neg\Diamond o$ for object o reveals the problem: it is perfectly possible that there *exists* a component C which performs t, i.e., $\Xi \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ where in $\acute{C}$, the thread n owns the lock of o (which contradicts the above-mentioned interpretation of $\neg\Diamond o$): after delivering the call to o_r, the thread may acquire the lock of o, as well, using internal steps.

Having clarified what $\Xi \vdash t : \Diamond o$ does not mean, now what does it assert? The correct interpretation is, that *for all* components C such that $\Xi \vdash C \stackrel{t}{\Longrightarrow}$, there *exists* a post-configuration $\acute{\Xi} \vdash \acute{C}$ such that $\Xi \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ where the thread owns the lock of o. Coming back to the original sample trace of (13): given t, there are many possible components C able to perform that trace, but for all of them it is possible that the thread holds the lock of the receiver object o_r after t. For o, however, there exist a component C, for which the lock of o is free. In the simplest case C may be such that the method l of object o_r does not invoke any method of o which would be a component-internal call and not visible at the interface. This is expressed by the negative assertion on the right-hand side of (14).

Now to the *definite* knowledge that a thread owns the lock of a given object. Note that the definition of $\Box o$ is not independent of $\Diamond o$, but builds upon it, but not vice versa.

Definition 3.7 (Must lock ownership) Given a sequence s of interactions of a single thread and a component object o, the judgment $\Xi \vdash s : \Box o$ ("after s, the thread of s must own the lock of o") is given by the rules of Table 11. For environment locks, i.e., when o is an environment object, the definition is dual.

Definition 3.5 and 3.7 where given using the interactions of a single thread. To lift the definition to traces of multiple threads, we use projection and write $\Xi \vdash t : \Box_n o$ for $\Xi \vdash (t \downarrow_n) : \Box o$, and analogously for $\Diamond_n o$.

The first rule M-I$\Box_1$ deals with incoming calls. Since the lock is not acquired atomically, an incoming call alone does not guarantee that the thread owns the callee's lock; it potentially owns it according to rule M-I$\Diamond_1$. If however the lock of an object is necessarily owned before the call, the same is true afterwards. Thus rule M-I$\Box_1$ corresponds to M-I$\Diamond_2$, but there is no rule for $\Box$ analogous to M-I$\Diamond_1$. A single incoming call cannot change a given lock from $\Diamond$-status or even from not-$\Diamond$-status to $\Box$-status, i.e., $\Box o$ can only be true after the communication, if it was true already before, which is what M-I$\Box_1$ (and the absence of an analogue of M-I$\Diamond_1$) stipulates. Rule M-I$\Box_2$ deals with incoming returns. As for incoming calls, the lock is owned for sure after

Table 11 Necessary lock ownership for Θ-locks

$$\frac{\Xi \vdash t : \Box o}{\Xi \vdash t\ \gamma_c? : \Box o}\ \text{M-I}\Box_1 \qquad \frac{\Xi \vdash t\ \gamma_r?\ \gamma_r'! : \Diamond o}{\Xi \vdash t\ \gamma_r?\ : \Box o}\ \text{M-I}\Box_2$$

$$\frac{\Xi \vdash t : \Diamond o}{\Xi \vdash t\ \gamma_c! : \Box o}\ \text{M-O}\Box_1 \qquad \frac{\Xi \vdash t : \Box o}{\Xi \vdash t\ \gamma_r! : \Box o}\ \text{M-O}\Box_2$$

the communication, if this was true before already. Hence $\Xi \vdash t : \Box o$ as premise. We need to be careful, however. After the return γ_r in question, the thread may continue *internally* i.e., without performing a further interface communication, and this internal reduction may relinquish the lock! This may be the case if the mentioned internal reduction includes the very last internal steps of a synchronized method call, before the call actually returns at the interface, re-establishing balance. In other words, after $\gamma_r?$, the component may be in a state where internally, the lock has already been released, only that the fact has not yet been manifest at the interface. This is captured in the premise $\Xi \vdash r\ \gamma_r?\ \gamma_r'! : \Diamond o$, i.e., the trace $r\ \gamma_r?$ is *extended* by one additional outgoing return $\gamma_r'!$, and if the thread *may* have the lock after this extended trace, then it must have the lock after $\gamma_r?$.

The M-O$\Box$-rules cover outgoing communication. Remember that outgoing communication leaves the $\Diamond$-information unchanged. For $\Box$-information, this is different and characteristic of the non-atomic lock-handling: an incoming call is the sign that we *may* have the lock of a component object, but only a following outgoing call is the observable sign that the component *must* have the lock (see M-O$\Box_1$).

Remark 3.8 ($\Diamond$ vs. $\Box$) Example 3.6 should have cautioned us not to jump to conclusions how to interpret the words "necessary lock ownership". Fortunately, for $\Box$, the interpretation is more straightforward in that no quantifier-alternation is involved. The assertion $\Xi \vdash t : \Box o$ stipulates that *for all* components which perform t, and *for all* post-configurations after t, i.e., for all situations $\Xi \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$, the thread owns the lock of o in $\acute{C}$.

This also makes clear that $\Diamond$ and $\Box$ are not dual to each other, in the sense that $\neg\Diamond o$ is not the same as $\Box\neg o$ (when "$\neg o$" is interpreted as "the thread does not have the lock"; we will not use the notation $\neg o$ later). Furthermore it sheds light on the fact that $\Diamond$ and $\Box$ are not defined at the same time (or one is derived from the other via modal duality), but that the $\Diamond$-predicate is defined first, and $\Box$ later, using $\Diamond$.

Let us illustrate the definitions on a few example traces.

Example 3.9 Consider the following trace:

$$t = t'\ \gamma_c?\gamma_r! = n\langle call\ o_r.l()\rangle?\ n\langle return()\rangle!. \tag{15}$$

Assume that in the prior history t', thread n is strongly balanced and all locks are free. Now, for t, thread n is strongly balanced as well, more precisely $\Xi_0 \vdash t : balanced^-$,

when Ξ_0 is the initial static context, which in particular asserts with $\Delta_0 \vdash \odot$ that the initial activity starts in the environment.

According to rule M-I$\Diamond_1$, $\Xi_0 \vdash t'\ \gamma_c? : \Diamond_n o_r$. However, we cannot derive the stronger assertion $\Xi_0 \vdash t'\ \gamma_c? : \Box_n o_r$, because the corresponding M-I$\Box$-rules require that, in order to hold, already $\Xi_0 \vdash t' : \Box_n o_r$, which is not the case.

Adding the subsequent return $\gamma_r!$ in (15) changes the situation as follows. The only candidate rule which applies for a trailing return for $\Diamond$ is M-$\Diamond$. It does not apply for t of our example, i.e., we have $\Xi_0 \vdash t : \neg\Diamond o_r$.

For $\Box$, the rule for output M-O$\Box$ does not apply since the premise $\Xi_0 \vdash t'\ \gamma_c? : \Box o_r$ does not hold (as explained above) for which we write $\Xi_0 \vdash t : \neg\Box o_r$. In other words, there exists a component which can perform t such that there *exists* a configuration after t where the thread does not have the lock. This was already implied by the stronger $\Xi_0 \vdash t : \neg\Diamond o_r$, of course. It is worth noting that during *no* point in the trace t, the lock is definitely taken in the following sense: for all (not necessarily proper) prefixes t' of t we have $\Xi_0 \vdash t' : \neg\Box o_r$, as just illustrated. Of course at some point in the *internal* execution between call $\gamma_c?$ and return $\gamma_r!$, the thread must have held the lock, only that for all components, in the configurations between $\gamma_c?$ there are points where the lock is *not* taken (immediately after the call and immediately before returning), and points where the lock is taken. But that is not enough to justify the $\Box$-assertion, only $\Diamond$ holds in between.

Let us replace t by a slightly more complex interaction, where at the end there exist some environment objects:

$$u = t'\ \gamma_c'?\ \gamma_r'! = t'\ n\langle call\ o_r.l(o_1)\rangle?\ n\langle return(o_2)\rangle!. \tag{16}$$

Assume that the argument references o_1 and o_2 are environment objects, and that after t', their locks are definitely free. According to the definition of the $\Diamond$- and $\Box$-predicate, we have $\Xi_0 \vdash u : \neg\Diamond o_1$ and $\Xi_0 \vdash u : \neg\Box o_1$, and the same for o_2: The $\Diamond$ information is only introduced for the receiver of a call (by rule M-I$\Diamond_1$, more precisely its dual), but not for arguments of a call nor for arguments of a return. Concerning the $\Box$-information: After the incoming call, $\Box o_1$ does not hold, as this would require $\Diamond o_1$ to hold as premise. After the return at the end, neither $\Box o_1$ nor $\Box o_2$ holds, since the only candidate rule, the dual of M-I$\Box_2$, does not apply.

Lemma 3.10 (Termination) *Used in a goal-directed manner and invoked on a weakly balanced trace, the derivation systems from Table 10 and 11 always terminate.*

Proof As the definition of $\Box$ uses $\Diamond$ but not vice versa, we can check termination separately, starting with $\Diamond$.

Given a (finite) weakly balanced trace r resp. its projection onto one chosen thread, $\Xi \vdash r : \Diamond o$ terminates since each of the premises mentions only a proper prefix of the trace of the conclusion (and furthermore, the functions *pop* and *receiver* terminate). Thus, also $\Xi \vdash r : \Box o$ terminates, since each rule give rise to a recursive call only of a proper prefix in the premise, or to a call of $\Diamond o$ (in M-I$\Box_2$ and M-O$\Box$), which terminates. □

Lemma 3.11 (Decidability) *Given a weakly balanced trace t, the relations $\Xi \vdash t : \Diamond_n o$ and $\Xi \vdash t : \Box_n o$ are decidable.*

Proof With termination for each $\Xi \vdash t : \Diamond_n o$ resp. $\Xi \vdash t : \Box_n o$ by Lemma 3.10, it remains to check that there are only finitely many derivations for a given judgment $\Xi \vdash t : \Diamond_n o$ resp. $\Xi \vdash t : \Box_n o$.

The $\Diamond$- and $\Box$-systems are almost goal-directed, but not quite. Goal-directed means that the premises of each rule are determined by the conclusion. This would imply that the derivation system describes directly a recursive function (since we have termination).

Starting with the system for $\Diamond o$: the rule that destroys goal-directedness is M-$\Diamond$, since the balanced trailing s_2 can be chosen differently. It is straightforward to see that we could additionally require that s_2 is the *maximally* balanced trailing trace, without changing the system.[8] This makes the system goal-directed and entails thus decidability. The combination of rules of the $\Box$-system are goal-directed from the start: the very last interaction determines the choice of the rule. This entails decidability. $\Box$

Decidability allows to consider the assertions $\Xi \vdash t : \Diamond_n o$ and $\Xi \vdash t : \Box_n o$ as boolean predicates (and analogously for $\Box$) and justifies the notation $\Xi \vdash t : \neg\Diamond_n o$ for $\Xi \nvdash t : \Diamond_n o$ we had used earlier in the examples.

The next lemma shows the expected implication between the two relations: if a thread necessarily owns a lock, then it also may own the lock. Of course this is only the intuitive meaning of the modal assertions; more precisely, $\Xi \vdash t : \Box_n o$ is intended to mean that for all components performing t, afterwards n holds the lock of o, and $\Xi \vdash t : \Diamond_n o$ means, that if a component C can perform t, then there exists a post-configuration after t where n holds the lock. Lemma 3.13 is a proof-theoretic statement about the relationship between the two derivation systems, which, luckily, matches with our intuition. We need a few simple properties of the $\Diamond$- and $\Box$-system first.

Lemma 3.12 ($\Diamond$ and $\Box$) *Let t a be a weakly balanced trace, o be a* component *object, and n a thread.*

1. *Let $a = \gamma?$. If $\Xi \vdash t : \Diamond_n o$, then $\Xi \vdash t\ \gamma? : \Diamond_n o$.*
2. *Let $a = \gamma_c!$. If $\Xi \vdash t : \Diamond_n o$ then $\Xi \vdash t\ \gamma_c! : \Diamond_n o$.*
3. *Let $a = \gamma!$. If $\Xi \vdash t\ \gamma! : \Diamond_n o$, then $\Xi \vdash t : \Diamond_n o$.*
4. *Let $a = \gamma!$. If $\Xi \vdash t\ \gamma! : \Diamond_n o$, then $\Xi \vdash t\ \gamma! : \Box_n o$.*
5. *Let $a = \gamma?$. If $\Xi \vdash t\ \gamma? : \Box_n o$, then $\Xi \vdash t : \Box_n o$.*

Proof By straightforward induction, using the properties of balance and weak balance (see also [26]). $\Box$

Lemma 3.13 ($\Box$ implies $\Diamond$) *Assume a weakly balanced trace t. If $\Xi \vdash t : \Box_n o$ then $\Xi \vdash t : \Diamond_n o$.*

[8] Different derivations in the system, i.e., different choices wrt. the balanced s_2 do not influence the success of the derivations; in other words, one has "coherence" of the derivation system.

Proof Assume $\Xi \vdash t : \Box_n o$, i.e., $\Xi \vdash s : \Box o$, where s is the projection of t to n. Now proceed by induction on the rules of Table 11.

Case: M-I$\Box_1$
We further distinguish wrt. receiver of the call. If $receiver(s'\ \gamma_c?) = o$, then the result follows directly by M-I$\Diamond_1$. If otherwise $receiver(s\ \gamma_c?) \neq o$, the result follows by induction and rule M-I$\Diamond_2$.

Case: M-I$\Box_2$
The premise $\Xi \vdash s\ \gamma_r?\ \gamma_r'! : \Diamond o$ implies with Lemma 3.12(3), $\Xi \vdash s\ \gamma_r? : \Diamond o$, as required.

Case: M-O$\Box_1$
By the premise of the rule and Lemma 3.12(2).

Case:M-O$\Box_2$
By induction and Lemma 3.12(1). □

3.2.2 Mutual Exclusion

So far we concentrated on each thread in isolation; the definitions of $\Diamond_n o$ and $\Box_n o$ have been used on projection of the global trace onto the thread n in question. This cannot be the whole story, as mutual exclusion is a global property concerning more than one thread. Especially for the $\Diamond$-information, concentrating on the thread-local view does not give the whole picture: $\Xi \vdash t : \Diamond_n o$ means, after t, the thread n may own the lock, based on local knowledge only, i.e., it may have the lock provided *none of the other threads* locks out the thread n in question. The formalization is based on a *precedence* relation on the events of a trace. An event is an *occurrence* of a label in a trace, i.e., as usual, events are assumed unique. In the following we do not strictly distinguish (notationally) between labels and events, i.e., we write $\gamma?$ for an event labeled by an incoming communication etc. To formalize the dependencies for mutual exclusion, we need to require that certain events are positioned *before* the lock has been taken, or *after* it has been released. So the following definition picks out relevant events of a trace. In the definition, $\preccurlyeq$ denotes the prefix relation. The $\acute{\Diamond}$-function ("after may") designates the labels *after* the point where the lock may be taken, for a given pair of thread and monitor. The $\grave{\Box}$-function ("before must") picks out the point before a thread enters the monitor.

Definition 3.14 Let t be the projection of a weakly balanced trace onto a thread n. Then the set of events $\acute{\Diamond}(t, o)$ is given by:

$$\acute{\Diamond}(t, o) = \{a \mid sa \preccurlyeq t \text{ is the longest prefix s.t. } \Xi \vdash s : \Diamond o\}. \tag{17}$$

Furthermore, the set of events $\grave{\Box}(t, o)$ is given as:

$$\begin{aligned}\grave{\Box}(t, o) = \{a_1 \mid\ & \Xi \vdash t : \Box o, s\ a_1\ a_2 \preccurlyeq t \text{ is the longest prefix s.t.} \\ & \Xi \vdash s : \neg\Diamond o,\ \Xi \vdash sa_1a_2 : \Box o\}.\end{aligned} \tag{18}$$

We use the following abbreviations: $\acute{\Diamond}_n(t, o)$ stands for $\acute{\Diamond}(t \downarrow_n, o)$ and $\acute{\Diamond}_{\neq n}(t, o) = \bigcup_{n' \neq n} \acute{\Diamond}(t \downarrow_{n'}, o)$, and analogously for $\grave{\Box}$.

Note that the "set" given by $\grave{\Diamond}$ in Definition 3.14 contains one element or is empty. The same holds for $\grave{\Box}$.

Based on these auxiliary definitions, we now introduce the three types of dependencies we need to consider. We start with data dependence.

Definition 3.15 (Data dependence) Given a trace r, reference o, and input label $\gamma?$, we write $\vdash_\Theta r : \gamma? \twoheadrightarrow^d o$ (in words: "o is potentially data-dependent on event/label $\gamma?$ of trace r"), if $o \in \mathit{names}(\gamma)$, where $r'\gamma?$ is a prefix of r. When given a tuple $\vec{o}$ of names, $\vdash_\Theta r : \vec{\gamma}? \twoheadrightarrow^d \vec{o}$ is meant as asserting $\vdash_\Theta r : \gamma_i? \twoheadrightarrow^d o_i$, for all o_i from $\vec{o}$. Then:

$$\begin{array}{l} D_\Theta(r\ \gamma!) = \{\vec{\gamma}? \twoheadrightarrow \gamma!\} \quad \text{provided } \vdash_\Theta r : \vec{\gamma}? \twoheadrightarrow^d \mathit{fn}(\gamma!) \cap \Delta(r) \\ D_\Theta(r\ \gamma?) = \{\}. \end{array} \tag{19}$$

For $\vdash_\Delta$ and D_Δ, the definitions are applied dually.

The definition states that, from the perspective of the component, arguments of an outgoing communication must either be generated previously by the component, or must have entered the component from the outside. The complexity of the technical definition is explained as follows. First of all, we calculate the dependence in (19) only for object references occurring free in the output label; those that occur under a ν-binder are generated by the component itself, and do not constitute a data dependence. For the same reason we consider only those free object references, which originally have been passed to the component during the history; we denote all ν-bound environment objects in r by $\Delta(r)$ (dually for component objects). Finally, each such object in $\gamma!$ may be potentially data dependent on *more* than one incoming label in the history r. It suffices to add *one* data dependence edge, which is non-deterministically chosen.

Definition 3.16 (Control dependence) Given a trace ra, where $n = \mathit{thread}(a)$, we write $\vdash r : a' \twoheadrightarrow^c a$, if $r \downarrow_n = r'a'$ for some label a'. We write $C(ra)$ for $\{a' \twoheadrightarrow a \mid \vdash r : a' \twoheadrightarrow^c a\}$.

Note that the set $C(r\ a)$ contains one element, i.e., one edge, or is empty.

Definition 3.17 (Mutual exclusion) Given a trace ra and a component object o, the label a gives rise to the precedence edges wrt. component locks given by:

$$\begin{array}{l} M_\Theta(r\gamma_c?, o) = \grave{\Diamond}_{\neq n}(r, o) \twoheadrightarrow \gamma_c? \\ M_\Theta(r\gamma_r?, o) = \{\} \\ M_\Theta(r\gamma!, o) = \gamma! \twoheadrightarrow \grave{\Box}_{\neq n}(r, o), \grave{\Diamond}_{\neq n}(r, o) \twoheadrightarrow \grave{\Box}_n(r\gamma!, o) \end{array} \tag{20}$$

For environment locks, the definition is dual.

Incoming calls can introduce a dependence with *other* threads n' competing for the concerned lock of the callee. Interactions of a thread n' occurring in the history r *after* n' has applied for the lock (but before $\gamma_c?$) makes evident that n' succeeded in

entering the monitor. Hence the corresponding monitor interactions of n' must have happened before the current incoming call succeeds in entering the monitor. Incoming returns do not introduce new dependencies wrt. Θ-locks (short for component locks), since the return releases the corresponding lock or keeps it, but does not acquire a lock nor competes for it.

Outgoing communication, however, does introduce dependencies, as they in many cases indicate that a lock definitely is taken or transiently has been taken since the last interaction of that thread. This introduces two types of dependencies. First, if there are other definite lock owners, then the current action $\gamma!$ must precede the monitor interactions of those successful competitors since the outgoing label is a definite sign that the thread of γ has held the lock of o before that step. This explains the edges $\gamma! \rightarrowtriangle \grave{\Box}_{\neq n}(r, o)$ in the definition. Secondly, $\gamma!$ does not only indicate that the thread in question had the lock prior to the step (at least transiently), but can also introduce definite lock ownership after the step (in particular, an outgoing call can introduce must-ownership). Hence, the monitor interactions of all competitors observed in the trace must precede the point, where the current thread n acquires the lock. This explains the dependence $\acute{\Diamond}_{\neq n}(r, o) \rightarrowtriangle \grave{\Box}_n(r\,\gamma_c!, o)$.[9] See also the trace of (24) in Example 3.18. The case of an outgoing return is illustrated by the trace of (22).

3.2.3 Examples

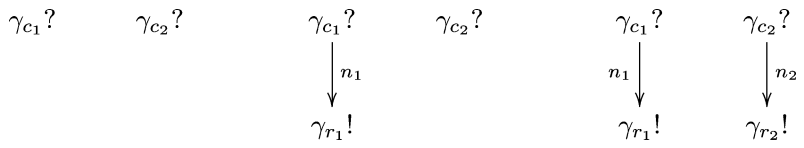
Let us illustrate the system on a few examples.

Example 3.18 (Mutex) We assume that the labels in the examples are not dependent on each other wrt. transmitted data. With such dependencies, the diagrams shown below would simply contain additional data dependence edges.

We start with the following trace

$$t = \gamma_{c_1}?\gamma_{c_2}?\gamma_{r_1}!\gamma_{r_2}!, \tag{21}$$

where $\gamma_{c_1}?$ and $\gamma_{r_1}!$ are interactions of thread n_1 and $\gamma_{c_2}?$ and $\gamma_{r_2}!$ belong to thread n_2.



After the two incoming calls, no dependence between the two actions is yet visible. Neither does the third step $\gamma_{r_1}!$, apart from the "intra-thread" dependence stipulating

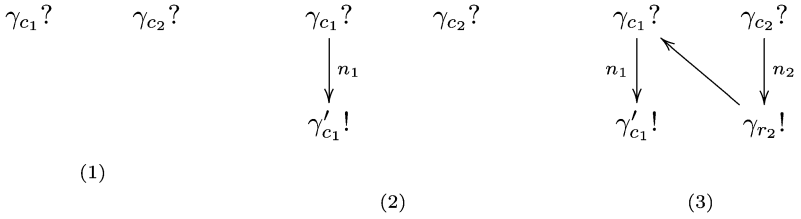
[9] In case the lock has been owned by thread n already prior to the outgoing communication, these edges are not actually needed; the condition only adds edges expressing dependencies which are already covered. Especially for outgoing returns, this means, one could formulate the corresponding clause of $M_\Theta(r\,\gamma_r!, o)$ without that edge.

that the corresponding call must precede the return, of course. After the fourth step, which adds the precedence $\gamma_{c_2}? \twoheadrightarrow \gamma_{r_2}!$, there are still no dependencies between n_1 and n_2. In particular, the outgoing return $\gamma_{r_2}!$ does not introduce such a dependence. This outgoing reaction makes evident that n_2 must have had the lock previously, indeed at some point between $\gamma_{c_2}?$ and the reaction $\gamma_{r_2}!$ now. Since, however, the return action of n_2 may come any time after the actual release of the lock and the same holds for the return action of thread n_1, the observation of the trace from (21) does not allow to derive any order in which the monitor is actually entered. Note that the same absence of precedence would hold for the alternative trace where the two incoming calls occur in reversed order at the interface (and/or the two returns occur in reversed order).

Now we replace the first outgoing return by an outgoing call:

$$t = \gamma_{c_1}?\gamma_{c_2}?\gamma'_{c_1}!\gamma_{r_2}!. \tag{22}$$

The evolution of the dependencies now looks as follows:

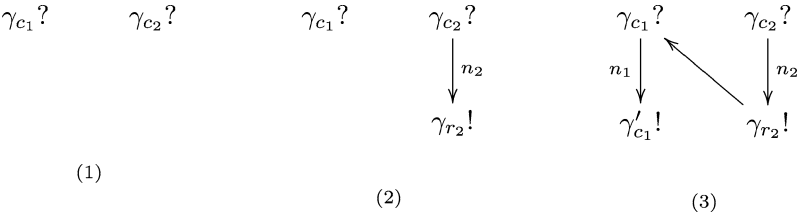


From the first to the second diagram, the outgoing call $\gamma'_{c_1}!$ still does not introduce a dependence between n_1 and n_2, even if now the outgoing call of n_1 shows that this thread must actually own the lock at that point. Intuitively, the reason again is that thread n_2 currently has shown only an incoming call: it may be that the call of n_2 did not succeed in entering, or alternatively that it had successfully entered the monitor and has *left it* again already, only that the return has not been visible at the interface. The reaction of n_2 in the fourth step shows that the first alternative cannot be true: Since n_1 at that point definitely ("□") holds the lock, and since the return-reaction of n_2 makes clear that n_2 must have held the lock at some point between n_2's call-return pair, it follows that $\gamma_{r_2}!$ must precede $\gamma_{c_1}?$.

What happens if the reactions of n_1 and n_2 are seen in reverse order:

$$t = \gamma_{c_1}?\gamma_{c_2}?\gamma_{r_2}!\gamma'_{c_1}!. \tag{23}$$

The dependencies now evolve as follows:

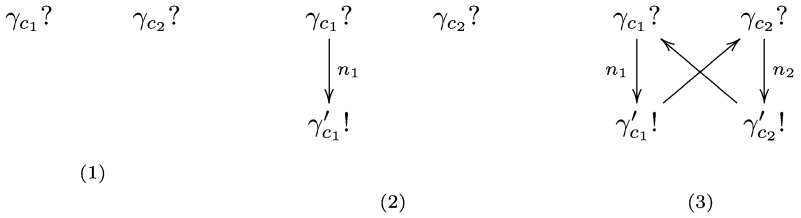


After the return in the third step, we cannot order n_1 and n_2, since we do not know whether n_1 has successfully entered the monitor and perhaps has left it already. Apart from the fact that the order of incoming calls is different, the situation is identical to the one after three steps of (21). The outgoing call in the last step shows, that n_1 has the lock and that therefore n_2, which transiently had the lock, must have executed its monitor actions before n_1. The diagrams after having seen (22) and (23) coincide. This is what one would expect considering the fact that each component showing (22) as interface behavior must necessarily also show (23) and vice versa. In other words: seeing one trace or the other must not lead to different conclusions wrt. the order in which the actions are actually executed on the monitor.

Finally, we have a look at the trace where both threads react by a call:

$$t = \gamma_{c_1}?\gamma_{c_2}?\gamma'_{c_1}!\gamma'_{c_2}!, \tag{24}$$

where the corresponding dependencies look as follows:



The sequence differs from the one for (22) only in the last diagram, where in addition to $\gamma'_{c_2}! \rightarrowtriangle \gamma_{c_1}?$, also the dependence $\gamma'_{c_1}! \rightarrowtriangle \gamma_{c_2}?$ is added. This yields a cycle in the precedence graph, showing that the forth step is not possible. Indeed, this cycle directly corresponds to the knowledge that both n_1 and n_2 must own the lock after the four interactions, which violates the mutual exclusion requirement.

Example 3.19 (Data dependence) Consider the following trace

$$\begin{aligned} t &= \gamma_{c_1}?\,\gamma_{c_2}?\,\gamma_{r_1}!\,\gamma_{r_2}! \\ &= \nu(o'{:}c).n_1\langle call\ o.l(o')\rangle?\ \nu(o''{:}c).n_2\langle call\ o.l(o'')\rangle? \\ &\quad n_1\langle return(o'')\rangle!\ n_2\langle return(o')\rangle!, \end{aligned} \tag{25}$$

from Fig. 1, consisting of two consecutive (synchronized) incoming calls of the same object o via different threads followed by their corresponding outgoing returns.

Concerning our legal trace system, trace t represents a legal trace. However, there exists no component which is able to perform t. The reason lies in the data dependence between the two consecutive calls. Consider the case that the thread of the first call, n_1, obtains the lock of object o. Then, obviously, the second call is blocked until the first call relinquishes the lock and subsequently returns (giving back the lock and performing the return at the interface, however, are not atomic). But in this case the return value of thread n_1 cannot be o'', since o'' is introduced to the component by the second call which cannot be processed, as mentioned before.

Starting with the second call leads to the similar problem, as the return value to the second call is introduced by the (blocked) first call.

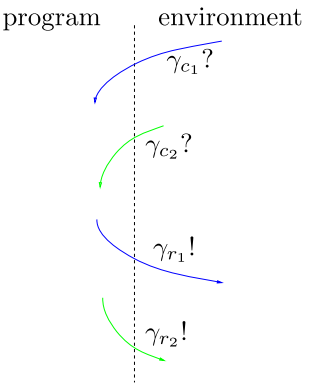


Fig. 1 Data dependence

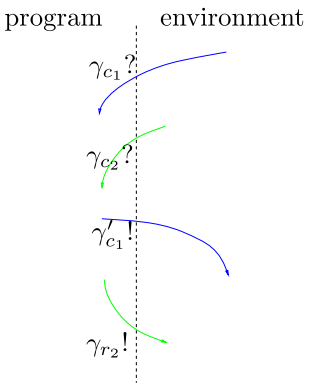


Fig. 2 Data dependence

Based on the control flow information alone, the trace of Example 3.19 is acceptable. One way to understand the problem is that, e.g., the first outgoing return of thread n_1 does not only reveal the definite information that the thread n_1 must have entered the monitor at some point,[10] but that also the *other thread* n_2 must have owned the lock before. The currently *"pending"* call $\gamma_{c_2}?$ of that trace is the only source of the value o'' sent in the return, and obviously the mentioned call of n_2 cannot hand over its argument without entering the monitor.

Example 3.20 Consider the trace $t = \gamma_{c_1}?\,\gamma_{c_2}?\,\gamma'_{c_1}!\,\gamma_{r_2}!$, in expanded form

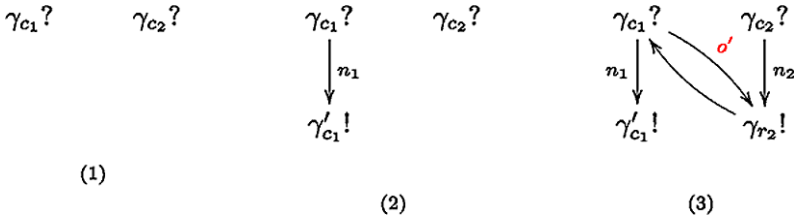
$$t = \nu(o'{:}c).n_1\langle \mathit{call}\ o.l(o')\rangle?\, n_2\langle \mathit{call}\ o.l()\rangle?\, n_1\langle \mathit{call}\ o'.l()\rangle!\, n_2\langle \mathit{return}(o')\rangle! \quad (26)$$

The trace considered now differs from the one of (25) in two points. The incoming call from thread n_1 is now answered by an outgoing call in the third interaction, not a

[10]Being an outgoing return in response to the call means that *after* the return we have definite information that n_1 does no longer hold the lock, as formalized by the previous derivation systems. But informally, i.e., without having a formal characterization yet, we know that n_1 in between the incoming call and the outgoing return must have held the lock.

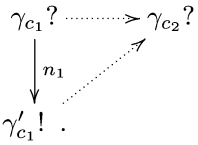
return. Secondly, the data dependence from the second incoming call (by thread n_2) to the outgoing communication by thread n_1 in the third step is removed by omitting o'' from (25). The trace is schematically shown in Fig. 2. Is it easy to see that the trace is impossible. Unlike in Example 3.19, where the impossibility was basically caused by an inconsistent, cyclic data dependence (and the fact that the methods are synchronized), now the data dependence alone is acyclic—thread n_1 must enter the monitor before n_2 to hand-over the o' needed by n_2, but not vice versa. This trace is impossible because if n_1 were to enter the monitor before n_2, which is required by the data dependency, it implied that n_1 kept the lock and n_2 *could not* enter the monitor. This consequence is independent of the scheduling.

Let us consider step by step, which knowledge about the order of events one can conclude from the information seen in the trace.



The left-most figure describes the information after the first two incoming calls. At that point, the order in which the two threads actually enter the monitor of o. After seeing n_1's reaction, the outgoing call $\gamma'_{c_1}!$, the picture changes as follows (see the diagram in the middle). First it is clear that $\gamma_{c_1}?$ must precede the reaction $\gamma'_{c_1}!$, hence the corresponding arrow. We additionally know that after the three interactions, thread n_1 *necessarily* owns the lock. Looking at n_2 *in isolation* we can derive that n_2 *may* own the lock at that point. Of course, globally, it is clear that it cannot have the lock now, since n_1 owns it.

Does this may- and must knowledge about the locks tells us something about the *order* in which n_1 and n_2 enter the monitor? At first sight, the following "argument" seems plausible: since n_1 now is definitely inside the monitor, thread n_2 has lost in the race between $\gamma_{c_1}?$ and $\gamma_{c_2}?$. Consequently $\gamma_{c_2}?$ can be executed, as it would seem, on the monitor, only *after* thread n_1 has left it again (if ever), which would justify a precedence arrow from $\gamma_{c_1}?$ and also from $\gamma'_{c_1}!$ to $\gamma_{c_2}?$:



This reasoning is flawed. It is true that, after the three interactions, thread n_2 definitely does not have the lock since n_1 owns it (or judged locally, it *may* have the lock, but not necessarily so), but it is perfectly possible that n_2 has taken the lock before n_1, and has *relinquished* it again already, only that the return label, which makes this fact visible, has not yet appeared in the trace! Therefore, the trace at this stage contains

not enough information to derive a definite order between the visits of n_1 and n_2 to the monitor.

Now, what changes when seeing the 4th label, the outgoing return of n_2? The return carries the reference o', which introduces a causal dependency between the first label $\gamma_{c_1}?$ and the return now. Furthermore, this implies that not only the return $\gamma_{r_2}!$ must be after $\gamma_{c_1}?$, but the same ordering is "inherited" for $\gamma_{c_1}?$ and $\gamma_{c_2}?$. This latter fact is a consequence of mutal exclusion. Whereas the precedence of $\gamma_{c_1}?$ over $\gamma_{r_2}!$ holds also if the methods in question are not synchronized or if the two threads do not compete for the same monitor, the precedence of $\gamma_{c_1}?$ over also the call $\gamma_{c_2}?$ means that data can be handed over from thread n_1 to n_2 by actually *entering* the monitor. So n_1 must enter the monitor first to deliver the data, and n_2 must enter the monitor afterwards to actually read the data.

One piece of information is still missing. The fact that thread n_2 now shows an outgoing communication—a return in this example—gives a further new bit of information. After three steps, as we argued, $\gamma_{c_1}?$ and $\gamma_{c_2}?$ are unordered, since in particular $\gamma_{c_2}?$ may have happened before $\gamma_{c_1}?$ and has already left the monitor again, without advertising this by the corresponding return. This uncertainty is resolved now in the 4th label. The return now rules out the possibility, that n_2 only tried to enter by was locked out by n_1, and thus makes clear that n_2 indeed had entered the monitor (which has been unclear so far). Since we know that n_2 *must* have the lock now, n_1 must have entered (and again left) the monitor *before.* This justifies the arrows from $\gamma_{c_2}?$ and $\gamma_{r_2}!$ to $\gamma_{c_1}?$ in the right-most diagram.

At this stage, we hence have derived a contradiction: the causal dependence graph contains a *cycle.*

3.3 Legal Traces System

Table 12 specifies *legality* of traces; the rules combine all mentioned conditions, type checking, balance, and in particular restrictions due to monitor behavior. We use the same conventions and notations as for the operational semantics (cf. Notation 2.3). The judgments in the derivation system are of the form

$$G_\Delta;\, \Delta, \Sigma \vdash r \rhd s : \mathit{trace}\, \Theta, \Sigma; G_\Theta \quad \text{resp.} \quad G; \Xi \vdash r \rhd s : \mathit{trace}\,. \tag{27}$$

In comparison to the judgments used in the operational semantics, the judgment from (27) contains a graph G_Θ as representation of *control, data,* and *mutex*-edges wrt. component locks (cf. Sect. 3.2.2), and dually G_Δ for environment locks. We adapt Notation 2.3 appropriately, writing G for the pair (G_Θ, G_Δ).

Before we turn to the derivation system, we need to adopt the definitions for type checking to the new setting of (27). The next definition determines the type *expected* for the transmitted values in a label. 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). For type checking in the operational semantics, Definition 3.21 was not needed, since the expected return type is stored as part of the block-syntax $\mathsf{let}\, x : T = \mathit{blocks}\, \mathsf{in}\, t$.

Definition 3.21 (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:

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

$$\frac{a = \gamma_r? \qquad pop(r\ a) = r'\ \nu(\Xi').n\langle call\ o_r.l(\vec{v})\rangle! \qquad \acute{\Xi} = \Xi + \Xi(r\ a) \qquad \acute{\Xi} \vdash o_r.l! : \vec{T} \to T}{\Xi \vdash r \rhd a : \vec{T} \to T}$$

In the rules, $\Xi(r\ a)$ 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.

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

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

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

Cf. also Definition 2.4, and in particular (7), checking well-typedness when given the expected type. We finally combine the enabledness check (Definition 3.4), the calculation of the sender and receiver from Definition 3.3, and the determination of the expected type as follows:

Notation 3.22 (Enabledness, communication partners, expected type) We write

$$\Xi \vdash r \rhd o_s \xrightarrow{a} o_r : \vec{T} \to T \tag{28}$$

(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).

Now to Table 12. We write $\Xi \vdash t : trace$, if there exists a derivation of $G_\emptyset$; $\Xi \vdash \epsilon \rhd t : trace$ according to Table 12, where $G_\emptyset$ is the empty dependence graph. We write $\Xi \vdash_\Delta t : trace$, if there exists a derivation of $G_\emptyset$; $\Xi \vdash \epsilon \rhd t : trace$, where only the *assumption contexts* are checked in the rules but not the commitments, i.e., the premises $\acute{\Xi} \vdash a : wt$ and $\vdash \acute{G} : ok\ remain$ in the rules for incoming communication L-CALLI and L-RETI, but for the outgoing communication, the corresponding premises are *omitted*. The situation is dual for $\Xi \vdash_\Theta t : trace$, which checks legality from the perspective of the component.

As base case, the empty future is always legal L-EMPTY, and distinguishing according to the first action a of the trace, the rules check whether a is possible. This

Table 12 Legal traces (dual rules omitted)

$$\Xi; G \vdash r \rhd \epsilon : trace \qquad \text{L-EMPTY}$$

$$\frac{\Xi \vdash r \rhd o_s \xrightarrow{a} o_r : \vec{T} \to \_ \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_ \quad \acute{G}_\Theta = G_\Theta \cup G_\Theta(ra, o_r) \quad \acute{G}_\Delta = G_\Delta \cup G_\Delta(ra, o_s) \quad \vdash \acute{G}_\Delta : ok \quad a = \nu(\Xi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \acute{\Xi}; \acute{G} \vdash r\ a \rhd s : trace}{\Xi; G \vdash r \rhd a\ s : trace} \text{ L-CALLI}$$

$$\frac{\Xi \vdash r \rhd o_s \xrightarrow{a} o_r : \_ \to T \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T \quad \acute{G}_\Theta = G_\Theta \cup G_\Theta(ra, o_r) \quad \acute{G}_\Delta = G_\Delta \cup G_\Delta(ra, o_s) \quad \vdash \acute{G}_\Delta : ok \quad a = \nu(\Xi').\ n\langle return(v)\rangle? \quad \acute{\Xi}; \acute{G} \vdash r\ a \rhd s : trace}{\Xi; G \vdash r \rhd a\ s : trace} \text{ L-RETI}$$

check is represented by checking whether the dependencies collected in the pair G are consistent, i.e., that the two graphs are *acyclic*. This is asserted by $\vdash G : ok$. Furthermore, the contexts are updated appropriately, and the rules recur checking the tail of the trace. The update for the dependence graph G_Θ given by the union the graph G_Θ before the step with

$$G_\Theta(ra, o) = M_\Theta(ra, o) \cup C(ra) \cup D_\Theta(ra), \tag{29}$$

where the argument o refers to the monitor relevant in that step, i.e., the monitor introduction potential inconsistencies. The definition for G_Δ is dually.

The rules are completely symmetric wrt. incoming and outgoing communication (and the dual rules are omitted). L-CALLI for incoming calls works similar to the CALLI-rules in the semantics. The premise $\Xi \vdash r \rhd o_s \xrightarrow{a} o_r$ checks whether the incoming call a is enabled and determines the sender and receiver at the same time (cf. (12) for the definition). The receiver o_r, of course, is mentioned directly, but o_s is calculated from the history r. In case of incoming communication, the relevant monitor for G_Θ is the receiver, and for G_Δ, the sender of the step.

Remember from Sect. 3.1 that the sender given by, e.g., $sender(r\ \gamma_c?)$ is not necessarily the "*real*" sending object which remains anonymous, but the last environment object the corresponding thread has entered in the past via an interface action. The sender in this sense is exactly the object, whose lock is relevant when updating/checking the dependencies in G_Δ. A consequence of the clean decoupling of component and environment in the assumption/commitment formulation of the legal traces is, that for incoming communication, the update of the graph G_Θ cannot introduce a cycle: incoming communications are checked for legality using the *assumptions,* not the commitments.

3.4 Soundness of the Abstractions

The section contains the basic soundness results of the abstractions. The first one in concerned is one basic invariant, namely the preservation of well-typedness under reduction, called subject reduction.

Lemma 3.23 (Subject reduction) $\Xi_0 \vdash C \stackrel{s}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$, *then* $\acute{\Xi} \vdash \acute{C}$.

Proof By induction on the number of reduction steps. That each internal step, structural congruence, and the external steps preserve well-typedness is shown by straightforward inspection of the rules, resp. induction. □

The following lemma expresses that the $\Diamond$- and $\Box$-assertions about the lock ownership appropriately catch the actual situation in a component.

Lemma 3.24 (Soundness of lock ownership) (1) $\Xi \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ *and* $\Xi \vdash t : \Box_n o$, *then thread n has the lock of o in* $\acute{C}$.

(2) *If* $\Xi \vdash C \stackrel{t}{\Longrightarrow}$ *and* $\Xi \vdash t : \Diamond_n o$ *and there does not exist an* $n' \neq n$ *such that the lock is owned by* n', *then* $\Xi \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ *for some* $\acute{\Xi} \vdash \acute{C}$ *s.t. the thread n has the lock of o in* $\acute{C}$.

Proof For part 3.24 observe that for $\Xi \vdash t : \Box_n o$ to hold, the number of *outgoing* calls with object o as sender must be strictly larger than the number of incoming returns with o as receiver. This implies that the lock must have been taken and not yet been released.

Concerning part 3.24 for $\Diamond$-information: Assume $\Xi \vdash t : \Diamond_n o$ for some thread n. Let s be the projection of t to n, i.e., $s = t \downarrow_n$, and we have $\Xi \vdash s : \Diamond_o$ by the rules of Table 10. If $s = s'\ a$ for some label and $\Xi \vdash s' : \Diamond o$.

Case: M-I$\Diamond_1$
In this case, the component object o is the receiver of the call. Since not other thread owns the lock of the object, thread n can proceed by an internal $\stackrel{\tau}{\rightarrow}$-step and take the lock by rule $\textsc{Call}^s_{i_1}$.

Case: M-I$\Diamond_2$
In this case of n in s' is an *outgoing* communication. Note that s' cannot be empty, as the premise of M-I$\Diamond_2$ requires $\Xi \vdash s' : \Diamond o$, which is not the case for $s' = \epsilon$. Thus by Lemma 3.12(4) $\Xi \vdash s' : \Box o$, which implies by part 3.24 of the lemma, that the tread n actually possesses the lock after s', and this does not change by an incoming call.

The remaining two rules work similarly, observing that a strongly balanced interaction of a thread (for rule M-O$\Diamond$) does not affect whether a thread owns a lock or not. □

Lemma 3.25 (Soundness of abstractions) *Assume* $\Xi_0 \vdash C$ *and* $\Xi_0 \vdash C \stackrel{t}{\Longrightarrow}$. *Then*

1. $\Xi_0 \vdash_\Theta t$: *trace and*
2. $\Xi_0 \vdash_\Delta t$: *trace implies* $\Xi_0 \vdash t$: *trace*.

Proof For part 1: The assertion $\Xi_0 \vdash_\Theta t$: *trace* can be split into three orthogonal parts (cf. the rules from Table 12): Well-typedness, weak balance, and acyclicity of the graph of dependencies. The operational rules of Table 8 assure that, for each thread and for all prefixes of t, the number of outgoing calls is always larger or equal the number of incoming returns, and dually, the number of incoming calls is larger

or equal the number of outgoing returns. This implies that each thread in t is weakly balanced (see [26]). That the typing conditions of $\Xi_o \vdash t : \mathit{trace}$ are met follows from subject reduction (Lemma 3.23). As for the acyclicity check: the control-dependence and the data-dependency edges are straightforward. The precedence expressed by the mutex-edges is justified by the soundness of lock-ownership (Lemma 3.24).

Part 2 follows from part 1 by definition of $\Xi_0 \vdash t : \mathit{trace}$, which combines $\vdash_\Delta$ and $\vdash_\Theta$. □

4 Conclusion

In this paper we characterized the external observable behaviour of multi-threaded object-oriented components with re-entrant monitors. We defined an abstract semantics in form of an object calculus, and showed soundness of the abstraction.

There is much work done in the field of abstract and fully abstract semantics for different languages, but less for object-orientation and for concurrency. The thesis [24] presents a fully abstract model for *Object-Z*, an object-oriented extension of the *Z* [23, 25] specification language. It is based on a refinement of the simple trace semantics called the complete-readiness model, which is related to the readiness model of Olderog and Hoare [20]. Viswanathan [28] investigates full abstraction in an object calculus with subtyping. The setting is slightly different from the one here, as the paper does not compare a contextual semantics with a denotational one, but a semantics by translation with a direct one. The paper considers neither concurrency nor aliasing.

Recently, Jeffrey and Rathke [14] extended their work [13] on trace-based semantics from an object-based setting to a core of *Java*, called *JavaJr*, including classes and subtyping. Another recent work by Poetzsch-Heffter and Schäfer [22] investigates a representation-independent behavioral semantics, as they call it, for object-oriented components. This semantics corresponds to a formalization of the interface behavior of open programs. The language features subclassing and inheritance, and especially the notion of an ownership-structured heap.

Koutsavas and Wand [17] present a sound and complete method for reasoning about contextual equivalence for a *Java*-like, class-based language, including inheritance, and based on bisimulation. The work is an extension of earlier, similar results for various λ-calculi with a store [16] and for imperative objects [15].

As future work, in a second part of this paper we will show completeness of the semantics, i.e., its full abstractness. Furthermore, we plan to extend the language with further features to make it more resembling *Java* or $C^\#$. Concerning the concurrency model, one should add thread-coordination using wait- and notify methods. Another interesting direction for extension concerns the type system, in particular to include *subtyping* and *inheritance*. For a first step in this direction we will concentrate on subtyping alone, i.e., relax the type discipline of the calculus to subtype polymorphism, but without inheritance. Concentrating on synchronized methods, this paper relied on an interleaving abstraction of the concurrent semantics. More complex interface behavior is expected when considering more general memory models.

Recently, we also started to characterize observability for the object concurrency model of the *Creol* language [7, 27], based on asynchronous method calls (futures

and promises). Another direction is to extend the semantics to a *compositional* one; currently, the semantics is open in that it is defined in the context of an environment. However, general composition of open program fragments is not defined.

Acknowledgements This work has been supported by the NWO/DFG project Mobi-J (RO 1122/9-4), by the DFG as part of the Transregional collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), and by the EU FP6 project Credo (IST-33826).

We thank the members of the PMA group in Oslo for many stimulating discussions.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer, New York (1996)
2. Ábrahám, E., Bonsangue, M.M., de Boer, F.S., Steffen, M.: Object connectivity and full abstraction for a concurrent calculus of classes. In: Li, Z., Araki, K. (eds.) ICTAC'04. Lecture Notes in Computer Science, vol. 3407, pp. 37–51. Springer, New York (2004)
3. Ábrahám, E., de Boer, F.S., Bonsangue, M.M., Grüner, A., Steffen, M.: Observability, connectivity, and replay in a sequential calculus of classes. In: Bonsangue, M., de Boer, F.S., de Roever, W.-P., Graf, S. (eds.) Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004). Lecture Notes in Computer Science, vol. 3657, pp. 296–316. Springer, New York (2005)
4. Ábrahám, E., Grüner, A., Steffen, M.: Dynamic heap-abstraction for open, object-oriented systems with thread classes. SoSYM J. (2007, accepted). This is a reworked version of the Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel technical report nr. 0601 and an extended version of the CiE'06 extended abstract
5. America, P.: Issues in the design of a parallel object-oriented language. Formal Aspects Comput. **1**(4), 366–411 (1989)
6. Brinch Hansen, P.: Operating System Principles. Prentice-Hall, Englewood Cliffs (1973)
7. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: de Nicola, R. (ed.) Proceedings of Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Vienna, Austria. Lecture Notes in Computer Science, vol. 4421, pp. 316–330 (2007)
8. de Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. **34**, 83–133 (1984)
9. ECMA International Standardizing Information and Communication Systems: C# Language Specification, 2nd edn. (Dec. 2002). Standard ECMA-334
10. Gordon, A.D., Hankin, P.D.: A concurrent object calculus: reduction and typing. In: Nestmann, U., Pierce, B.C. (eds.) Proceedings of HLCL '98. Electronic Notes in Theoretical Computer Science, vol. 16.3. Elsevier, Amsterdam (1998)
11. Gosling, J., Joy, B., Steele, G.L., Bracha, G.: The Java Language Specification, 2nd edn. Addison-Wesley, Reading (2000)
12. Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)
13. Jeffrey, A., Rathke, J.: A fully abstract may testing semantics for concurrent objects. In: Proceedings of LICS '02. IEEE Computer Society Press (July 2002)
14. Jeffrey, A., Rathke, J.: Java Jr.: a fully abstract trace semantics for a core Java language. In: Sagiv, M. (ed.) Proceedings of ESOP 2005. Lecture Notes in Computer Science, vol. 3444, pp. 423–438. Springer, New York (2005)
15. Koutavas, V., Wand, M.: Bisimulations for untyped imperative objects. In: Sestoft, P. (ed.) Proceedings of Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2005, Vienna, Austria. Lecture Notes in Computer Science, vol. 3924, pp. 146–161. Springer (2005)
16. Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: Proceedings of POPL '06, pp. 141–152. ACM (Jan. 2006)
17. Koutavas, V., Wand, M.: Reasoning about class behavior. In: Informal Workshop Record of FOOL 2007 (Jan. 2007)
18. Milner, R.: Fully abstract models of typed λ-calculi. Theor. Comput. Sci. **4**, 1–22 (1977)

19. Morris, J.H.: Lambda calculus models of programming languages. Ph.D. thesis, MIT (1968)
20. Olderog, E.-R., Hoare, C.A.R.: Specification-oriented semantics of communicating processes. Acta Inf. **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
21. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**, 223–255 (1977)
22. Poetzsch-Heffter, A., Schäfer, J.: A representation-independent behavioral semantics for object-oriented components. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS '07. Lecture Notes in Computer Science, vol. 4468. Springer, New York (2007)
23. Potter, B.F., Sinclair, J.E., Till, D.: An Introduction to Formal Specification and Z. Series in Computer Science. Prentice-Hall, Englewood Cliffs (1990)
24. Smith, G.P.: An object-oriented approach to formal specification. Ph.D. thesis, Department of Computer Science, University of Queensland (Oct. 1992)
25. Spivey, J.M.: The Z Notation: A Reference Manual. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1989)
26. Steffen, M.: Object-connectivity and observability for class-based, object-oriented languages. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel (2006), submitted 4th July, accepted 7 February 2007
27. The Creol language. http:www.ifi.uio.no/~creol (2007)
28. Viswanathan, R.: Full abstraction for first-order objects with recursive types and subtyping. In: Proceedings of LICS '98. IEEE Computer Society Press (July 1998)