A fully abstract semantics for UML components

F.S. de Boer[1,2], M.M. Bonsangue[2] *, M. Steffen[3], and E. Ábrahám[3]

[1] CWI, Amsterdam, The Netherlands
frb@cwi.nl

[2] LIACS, Leiden University, The Netherlands
marcello@liacs.nl

[3] Christian-Albrechts-University, Kiel, Germany
{ms,eab}@informatik.uni-kiel.de

Abstract. We present a fully abstract semantics for components. This semantics is formalized in terms of a notion of trace for components, providing a description of the component externally observable behavior inspired by UML sequence diagrams. Such a description abstracts from the actual implementation given by UML state-machines. Our full abstraction result is based on a may testing semantics which involves a composition of components in terms of cross-border dynamic class instantiation through component interfaces.

1 Introduction

The Unified Modelling Language (UML)[18] is widely adopted as the de facto industry standard for modelling object-oriented software systems. It consists of several graphical notations providing different views of the system being modelled. There are two basic types of diagrams: behavior diagrams and structure diagrams. These diagrams include sequence diagrams, state machines, class diagrams and component diagrams.

We use UML for investigating features such as state encapsulation, and name-passing in synchronous communication in combination with dynamic class instantiation. Basically, in UML a component is a set of classes with explicit contextual dependencies. Some instances of classes of a component are called ports. Components can communicate only through their ports. Most importantly, a port of a component can also instantiate new ports of another component. The explicit context dependencies of a component guarantee that ports have enough structural information about the environment. However the behavior of such an external environment is not under control of the component itself. In other words, a component is an open program, with implementation code containing calls to operations and constructors of interfaces that are not bound to any particular behavior specification.

From the point of view of a component, the ports of other components belong to the environment, and are internally known only as typed identifiers. Although the behavior of the environment is not fixed at priori, it has to obey to certain laws. For example, because the state of a port is encapsulated, external ports *cannot* always communicate

* The research of Dr. Bonsangue has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences

with each other. To illustrate this, consider a port of a component i that creates two new ports e_1 and e_2 of some component in the environment. The ports e_1 and e_2 are both external, but unable to communicate with each other unless the internal object i let one of them know the identity of the other. The above situation is characteristic of a framework with dynamic scope: new clusters of objects that know each other can be created as new external instances appear, and old clusters may merge as a consequence of a communication.

1.1 Contribution of this paper

In this paper we select a subset of UML notations suitable as basis for modelling component-based systems. Inspired by UML sequence diagrams, we give a denotational semantics to UML components in terms of traces of their externally observable events. A trace describes a sequence of interactions between the ports of a set of components. Here a port is an instance of a class of a component realizing one of its interface, and an interaction is a synchronization on an operation declared on one of the interface of a component.

We define an observational equivalence for components based on may testing, and show that ordinary traces are, in general, not fully abstract: two components can be observationally equivalent but their associated set of traces be different. Our main result is the characterization of trace abstractions that takes into account the clustering structure of objects dictated by their dynamic scope. These traces are full abstract with respect to may testing observational equivalence.

1.2 Related work

There is an increasing interest to give a rigorous foundation to UML for addressing, e.g., the needs for modelling safety critical applications. Some approaches are based on translating UML subsets into existing formalisms, like the π-calculus [19], other have proposed new meta-modelling language calculi as foundation for the semantics of UML, e.g. [11]. In this paper we present a variant of the UML subset considered by Damm et al. and formalized as a transition system [12]. The most significant departures from this work are that we do not consider asynchronous inter-object communications and do not distinguish among active, reactive and passive objects.

There are several full abstraction results for may-testing semantics for calculi of processes interacting in dynamically changing communication topology [6, 14]. The UML description of classes by state-machines combines mechanisms for dynamic process creation similarly to object calculi [1, 10, 20, 16] with synchronization mechanisms as in process calculi [9, 6, 14].

The closest work to our is Jeffrey and Rathke [16] fully abstract semantics of concurrent objects. While our components are open, programs in [16] are closed, in the sense we explained above, since their creation of a new object involves the specification of the behavior of the newly created object. Consequently, in their setting, the environment can be basically viewed as a static and a priori given group of objects. This contrasts with our setting, where the program itself creates dynamically its own

environment and imposes constraints on the communication topology of its environment.

Different from previous full abstraction results, the construction of a distinguishing context in the full abstraction proof requires a novel technique for the definition of a generic behavior capturing all instances of an external class. This we consider as one of the main technical contribution of our paper, that helps in a better understanding of the role of static class variables in class-based object-oriented languages like Java.

2 UML classes, state-machines and components

Next we describe the subset of UML we use in this paper. We use UML as an inspiration source, and have no pretence of fully formalizing the numerous concepts used in the UML diagrams. UML is an object-oriented modelling technique based on the concept of class. A *class* is a named description of a set of objects. Its signature consists of a finite set of attributes and a finite set of operations (one of them declared as constructor). Attributes and operations are typed either by basic types (like integers and Boolean) or by the identifier of a class or of an interface. An *interface* is a named description of a set of operations. Differently from a class signature, an interface does not declare any attribute. We say that an interface is *realized* by a class (that for simplicity we assume carrying the same name) if the set of operations of the interface is included in that of the class realizing it.

An *object* is an instance of a class. There are different kinds of inter-object communications in UML. We consider only communication via *synchronous operations* restricting to operations with two parameter only: one for passing the identity of the caller of the operation, and another for passing a value (that we will often assume to be the identity of another object). The execution of a synchronous operation involves a synchronization on the execution of an operation call by the sender and a corresponding trigger by the receiver. Such a synchronization results in an *assignment* of the value of the actual parameters of the operation call to the instance variables of the receiver that appear as formal parameters of the operation.

In contrast to a synchronous operation, a *primitive operation* is an operation acting directly on the instance variables of the objects, without any synchronization. Therefore the meaning of a primitive operation is defined in terms of a state transformation.

2.1 Abstract state-machines

In UML the behavior of an object is describe generically by means of an abstract state-machine associated to the class of which it is an instance. A *state-machine* is a kind of structured transition system that records the dependencies between the states of an object and its reaction to messages. More formally, a state machine associated to a class c consists of transitions of the form

$$l_1 \xrightarrow{[g]t/a}_c l_2$$

where l_1 is the entry location and l_2 is the exit location of the transition. Transitions may be *guarded* by a boolean guard g and labelled by a *trigger* t and an *action* a. The evaluation of the boolean guard g is assumed to be side-effect free.

A trigger t is of the form

$$op(x, y)$$

where op is the name of an operation (possibly the constructor) declared in the class c, while x, y are attributes of c used to store the identity of the caller and the value it pass when calling the operation.

An action a is either a primitive operation, a constructor call or a synchronous operation call. A constructor call is of the form $c.new(self, x)$, where new is the constructor of the class (or interface) c, and the attribute $self$ store the identity of t the caller object. The attribute x is typed by c and it will store the identity of the newly created object. A synchronous operation call is of the form

$$x.op(self, y)$$

where op is an operation declared in the class (or interface) typing the attribute x, that stores the identity of the callee of the operation. The attribute y is also declared in c and stores the value to be passed to the callee. We have not considered the more usual synchronous operations that return by means of a rendez-vous mechanism because we can encode this mechanism by means of an appropriate operation call and a respective trigger.

2.2 Components

In this paper we consider a *component* $\mathcal{C}$ as a part of a system consisting of a set of classes B and a set of interfaces $I = P \cup R$. Each class in B is associated with state-machine. The operations of the interfaces in I are typed only by other interfaces in the same set I. Interfaces in I can be either provided or required. Each *provided interface* $p \in P$ is realized by a class in B, and hence with the same name of p. A *required interface* $r \in R$ is an interface with a name different from that of any other class in B. It can be used by classes in B for typing their attributes. This way a component declares its dependencies on another components with interfaces in R as provided interfaces.

A class realizing a provided interface or depending on one or more required interfaces is called a *role*, and its instances are called *ports* [5]. An *internal class* is a class of a component that is not a role. Attributes of an internal class are typed only by primitive types or by classes within the same component, whereas attributes of a role may be typed also by the required interfaces. This means that a component is an open system, with its ports as the only points of interactions with environment: ports may be triggered by other ports in the environment, and call operations declared in the required interfaces, including the declared constructors. However, a class realizing a required interface is external, i.e., it belong to a different component. Encapsulation of the component internal implementation is ensured because instances of internal classes may synchronize only on operations of other objects within the same component, thus preventing a tight coupling between the component internal structure and the component environment.

Components can be composed by connecting the required interfaces of a constituent component with the provided interfaces (that for simplicity we assume to have the same name) that belongs to other constituent components. For simplicity we define interface

connection as set inclusion of operations. More formally, let $\mathcal{C}_1 = \langle B_1, P_1 \cup R_1 \rangle$ and $\mathcal{C}_2 = \langle B_2, P_2 \cup R_2 \rangle$ be two components. Their *composition* $\mathcal{C}_1 \oplus \mathcal{C}_2$ is defined as the component $\mathcal{C} = \langle B, I \rangle$ with $B = B_1 \cup B_2$ (that are assumed to be disjoint) and with $I = P \cup R$ obtained by taking $P = P_1 \cup P_2$ and $R = (R_1 \setminus P_2) \cup (R_2 \setminus P_1)$. For example, if one component provides all interfaces required by another one, then the component resulting from their composition has no required interfaces, and remains open to the environment only via its provided interfaces.

The above notion of component is inspired by that of UML as introduced in [18], but it differs in a number of crucial points. In particular, for simplicity we do not allow hierarchical composition of components (and hence we do not need delegation connectors), and, contrary to UML 2.0 we do not consider components as unit of instantiation but rather we consider a component as a static unit of abstraction with a dynamically growing number of ports.

2.3 Operational semantics

Next we define the operational semantics of a component in terms of the abstract state machines associated with each of its constituent classes.

Let $Class$ be a set of class (and interface) identifiers, with typical element c, and assume given, for each class name c, an infinite set $Obj(c)$ of names for the instances of the class c. We denote by Obj the union of $Obj(c)$ for all $c \in Class$. Further, let Att be a set of attributes (including loc and $self$) and Val be a set of values (including the undefined value nil).

A *object diagram* σ of a component $\mathcal{C} = \langle B, I \rangle$ is a partial function in $Obj \rightharpoonup (Att \to Val)$ assigning values to attributes of the existing instances of classes in B. The domain of an object diagram σ is denoted by $dom(\sigma)$, and the value $\sigma(o)(x)$ of the instance variable x of the object o is denoted by $\sigma(o.x)$. For all $o \in dom(\sigma)$ we require that $\sigma(o.self) = o$ and that $o \in Obj(c)$ for some class c in B.

Control information of each object o in an object-diagram is given by $\sigma(o.loc)$, assuming for each class that the attribute loc is used only to refer to the current location of the state machine of the class of which o is an instance. An object diagram is called *initial* if the only attributes different from nil are $self$ and loc.

The operational semantics of a component $\mathcal{C} = \langle B, P \cup R \rangle$ is defined in terms of a *transition relation* $\longrightarrow$ between object diagrams labelled by externally observable *communication events* of the form

$$e.op(i, v) \;\text{ and }\; i.op(e, v)\,, \tag{1}$$

where $e \in Obj(r)$, for some required interface $r \in R$, is the identity of an *external port*, and $i \in Obj(p)$, for some provided interface $p \in P$, is the identity of an *internal port* of $\mathcal{C}$. The idea is that i is an instance of the class of $\mathcal{C}$ realizing the interface p, whereas e is an instance of the class r realizing the interface r in another component. We will use this convention throughout this paper. The event $e.op(i, v)$ denotes the synchronization of the port e with the port i on the operation op provided by e. Similarly, $i.op(e, v)$ denotes the synchronization of the port i with the port e on an operation op provided by i. In both cases the synchronization involves the transmission of the value v.

We label the *transition relation* $\longrightarrow$ also with *creation events* of the form

$$new(o, u)$$

indicating the synchronization on the constructor new of the class c between the object creator o and the new instance u of c. As usual, a transition labelled by τ denotes an internal activity, such as the execution of a primitive operation or an intra-component synchronization.

The flow of control of each object is described according to the transitions of the state machine associated to the class of which it is an instance. For each transition

$$l_1 \stackrel{[g]t/a}{\Longrightarrow} l_2$$

of an abstract state machine we assume a unique intermediate location $l_{1,2}$ to model the interleaving point between the guard and trigger on the one hand, and the action on the other hand. Further, we assume for each boolean *guard* g an evaluation function g such that $g(\sigma, o)$ denotes the boolean result of the evaluation of g by the object o in the object diagram σ; note that guard evaluation is free of side effects, i.e., it does not affect the object diagram itself. Similarly, we assume for each primitive operation a, a state transformer function a such that $a(\sigma, o)$ denotes the object diagram that results from the application of a in the initial diagram σ by the object o. We consider only state transformations that change only instance variables of the object executing it. We do not allow, for example, that an object can assign values to instance variables of other objects within the same component.

The transition relation $\longrightarrow$ associated to a component $\mathcal{C} = \langle B, P \cup R \rangle$ is defined by distinguishing the following cases:

Internal synchronization: Let o and u be instances of the classes $c, d \in B$, respectively, both inside the component $\mathcal{C}$. Assume the object o is in a location $\sigma(o.loc) = l_1$ while the object u is in the intermediate location $\sigma(u.loc) = l_{3,4}$, where $\sigma(u.x) = o$ and $\sigma(u.y) = v$. If the guard $g(\sigma[o.x/u, o.y/v], o)$ evaluates to true then the synchronization of the objects o and u on the operation op is described by the following rule

$$\frac{l_1 \stackrel{[g]op(x,y)/-}{\Longrightarrow}_c l_2 \quad l_3 \stackrel{-/x.op(self,y)}{\Longrightarrow}_d l_4}{\sigma \stackrel{\tau}{\longrightarrow} \sigma'},$$

where σ' is the resulting object diagram with $\sigma'(o.x) = u$ and $\sigma(o.y) = v$. The flow of control of the objects o and u is described by their associated state machines and their new locations are $\sigma'(o.loc) = l_{1,2}$, $\sigma'(u.loc) = l_4$, respectively. Note that the evaluation of the guard is in parallel with the execution of the trigger, meaning that the guard g is evaluated in a state that take into account the new values of the actual parameters of the trigger.

Class instantiation: Let o be an instance of a class $c \in B$. Assume o is in the intermediate location $\sigma(o.loc) = l_{2,3}$ ready to execute a call to the constructor new of the class $d \in B$, with d in the same component of c. If the guard $g(\sigma[u.x/o], u)$ evaluates

to true then class instantiation is specified by the following rule

$$\frac{l_0 \xrightarrow{[g]new(x,y)/-}_d l_1 \quad l_2 \xrightarrow{-/d.new(self,x)}_c l_3}{\sigma \xrightarrow{new(o,u)} \sigma'},$$

where l_0 is the initial location of the state machine associated with the class d, and the domain of σ' extends that of σ with the name $u \in Obj(d) \setminus dom(\sigma)$ of the newly created object. The resulting object diagram σ' maps the new name u to the instance variables $o.x$, $u.y$ and $u.self$, while the caller o is assigned to the variable $u.x$. The locations of the two objects o and u are updated to $l_{1,2}$ and l_4, respectively. Finally, all other instance variables of u are set to the undefined value nil.

Primitive operation: Let o be an object of a class $c \in B$ of the component $\mathcal{C}$ with $\sigma(o.loc) = l_{1,2}$, and let op be a primitive operation. Then

$$\frac{l_1 \xrightarrow{-/op}_c l_2}{\sigma \xrightarrow{\tau} \sigma'},$$

where $\sigma' = op(\sigma, o)[l_2/o.loc]$. The execution of a primitive operation op generates a 'silent' transition transforming the object diagram σ according to the associated function $op(\sigma, o)$ and updating the location loc of the object o to l_2.

Synchronous operation call: Let i be a port instance of a role $c \in B$ of the component $\mathcal{C}$, and let $r \in R$ be a required interface of $\mathcal{C}$ declaring the synchronous operation op. Assume that in the object diagram σ the port i is in an intermediate location $\sigma(i.loc) = l_{1,2}$ where it can call a synchronous operation op of the external port $\sigma(i.x) = e$. Then

$$\frac{l_1 \xrightarrow{-/x.op(self,y)}_c l_2}{\sigma \xrightarrow{e.op(i,v)} \sigma'},$$

where $\sigma(i.y) = v$ and σ' is as σ, but for the location loc of i that is assigned to l_2. Note that because x typed by a required interface $r \in R$, there is no class in B with that name. Therefore e is an object not in $dom(\sigma)$.

Constructor call: A port i instance of a role $c \in B$ of the component $\mathcal{C}$ can create a new port $e \in Obj(r)$ of another component via a call of the constructor new declared in a required interface $r \in R$ of $\mathcal{C}$. This is described by the rule

$$\frac{l_1 \xrightarrow{-/r.new(self,x)}_c l_2}{\sigma \xrightarrow{new(i,e)} \sigma'},$$

where $\sigma(i.loc) = l_{1,2}$, $\sigma'(i.loc) = l_2$ and $\sigma'(i.x) = e$, for some $e \in Obj(r)$. Note that $e \notin dom(\sigma)$, because $r \in R$ is a required interface of $\mathcal{C}$.

Evaluation of a guard and a trigger: Let i be a port instance of a role $c \in B$ of the component $\mathcal{C}$, and assume that op is a synchronous operation declared by the provided interface $c \in P$. If in the object diagram σ the port i is in a location $\sigma(i.loc) = l_1$, and the guard $g(\sigma[i.x/e], i)$ evaluates to true, then its trigger op can be executed as consequence of the reception of the message $op(e, v)$ sent by an external port e. This inter-component synchronization is described by the rule

$$\frac{l_1 \xRightarrow{[g]op(x,y)/-}_c l_2}{\sigma \xrightarrow{i.op(e,v)} \sigma'},$$

where $\sigma'(i.loc) = l_{1,2}$, and $\sigma'(i.x) = e$ and $\sigma'(i.y) = v$ for some value v and object $e \in Obj(d)$ with $d \notin B$.

Port instantiation: A new instance i of a role $c \in B$ of a component $\mathcal{C}$ can be created by an external port e via a call to the constructor new declared in the provided interface $c \in P$. If the guard $g(\sigma[i.x/e], i)$ evaluates to true, this is described by the rule

$$\frac{l_0 \xRightarrow{[g]new(x,y)/-}_c l_1}{\sigma \xrightarrow{new(e,i)} \sigma'},$$

where $e \in Obj(d)$ with $d \notin B$ and $i \in Obj(c) \setminus dom(\sigma)$ is the identity of the newly created port. Here l_0 is the initial location of the state machine associated to c, and σ' extends σ by assigning $i.loc$ to $l_{0,1}$, $i.self$ and $i.y$ to i, and $i.x$ to e (all other instance variables of i are mapped to the undefined value nil).

Definition 1. *An execution ξ of a component $\mathcal{C}$ is a finite sequence*

$$\sigma_0 \xrightarrow{\ell_1} \sigma_1 \cdots \sigma_{n-1} \xrightarrow{\ell_n} \sigma_n$$

of labelled transitions starting from an initial object diagram σ_0.

From an execution sequence we can extract information about the order of creation among the objects of the component. In fact, given an execution ξ of a component $\mathcal{C} = \langle B, I \rangle$, we define the creation relation $<_\xi$ as the least binary transitive relation on Obj such that

$$o <_\xi u \text{ if } new(o, u) \text{ appears as a label in } \xi,$$

with new the constructor of the class of which u is an instance. Note that in general, the above creation relation will form a forest rather than a tree, because an execution does not record the creation of external ports by other external ports.

3 Testing semantics

In this section we define a may testing semantics for components. To define the notion of testing semantics, let $ISuccess$ be a distinguished interface consisting of the constructor new and one distinguished operation, $success$, with a parameter of type $ISucess$. We

say that a component $\mathcal{C}$ *succeeds*, denoted by $\mathcal{C}\downarrow$, if and only if we may observe only a single call to the $success$ operation by one of its port. More formally, $\mathcal{C}\downarrow$ if and only if there exists an execution ξ of C such that

$$\langle e.success(i, e)\rangle$$

appears as the only communication event in ξ, where e is an external port and i an internal one. This implies that a component may succeed only if $ISuccess$ is one of its required interface.

Definition 2. *Two components $\mathcal{C}_1$ and $\mathcal{C}_2$ with the same provided and required interfaces (not including $ISuccess$) are may-equivalent, denoted by $\mathcal{C}_1 \simeq \mathcal{C}_2$, if*

$$(\mathcal{C} \oplus \mathcal{C}_1)\downarrow \textit{ if and only if } (\mathcal{C} \oplus \mathcal{C}_2)\downarrow$$

for any other component $\mathcal{C}$.

This is a natural adaptation to components of the original definition of may testing semantics for concurrent processes [15]. Note that we allow only the tester component C to require the interface $ISuccess$ and hence to call the $success$ operation by one of its port.

4 Trace Semantics

In the rest of this paper we look for another characterization of the may-equivalence between components that avoids a universal quantification on the tester components. Our starting point are UML message sequence charts. They provide a visual representation of the interactions among of a set of objects in terms of the messages they exchange. Since component interfaces are intended to shield the details of a component implementation from the environment, a sensible semantics for components should abstract from synchronization among objects within the component.

For a given component $\mathcal{C}$, finite sequences of externally observable communication events thus specify the interactions between instances of internal classes realizing the provided interfaces and instances of external classes realizing the required interfaces. Such sequences abstract both from the interactions between instances of classes internal to the components and the interactions between instances of classes external to the component. However, these sequences can be ambiguous or describe information that cannot be implemented by any component. Consider for example the following sequence

$$e.op_1(i, e) \cdot e'.op_2(i, e') \cdot i.op_3(e'', e'')\,,$$

where e, e', and e'' are assumed to be three distinct external ports. The first two events indicate that both e and e' are known to the internal port i, for example because they have been both created by i. In order to justify the last event which involves a call of the operation op_3 of i by e'', there are three possible scenarios:

1. e'' has created i;

2. e'' has received its knowledge of i from e; and
3. e'' has received this knowledge from e'.

These different scenarios are due to three valid assumptions on object creation outside the component, namely e'' can be an ancestor of i,e can be an ancestor of e'', or e' can be an ancestor of e''.

This implicit non-determinism in a sequence of observable events thus allows different incompatible behaviors of the external objects. To resolve this non-determinism we associate to each sequence t of observable events a creation tree.

Definition 3. *A* trace t *is a finite sequence of communication events of the form* $o.op(u, v)$ *together with binary relation* $\prec_t$ *on* Obj *(called the* tree of creation*) such that for each name* u *(but one, the root of the tree) occurring in the sequence there is a unique different name* o *in the same sequence with* $o \prec_t u$*.*

In the sequel, we denote by $t\restriction_o$ the sub-trace of t with events involving the object o as either the caller or the callee of a synchronous operation. The associated tree of creation is restricted to the names appearing in the restricted sequence (but the root). Moreover, given a component $\mathcal{C} = \langle B, I\rangle$, we denote by $\partial_{\mathcal{C}}(t)$ the result of removing from the trace t all its events that are not externally observable, that is, those communication events involving instances of classes in B as caller or callee of a synchronous operation.

Definition 4. *We define a* trace of a component $\mathcal{C}$ *to be trace* t *consisting of a finite sequence of observable events induced by an execution* ξ *of* $\mathcal{C}$ *together with a creation tree* $\prec_t$ *such that for each ports* o, u *appearing in* t*, if* $o <_\xi u$ *then* $o \prec_t u$*.*

It should be observed that the creation tree of a trace of a component $\mathcal{C}$ is in fact an abstraction from the actual information on object creation since the latter may involve instances of classes that are strictly internal (or external) to $\mathcal{C}$, i.e., instances of classes that do not realize any provided (or required, respectively) interface. Consequently, the relation $\prec_t$ is more adequately described as the ancestor relation between ports appearing in t that are *indirectly* related because of a creation chain passing through internal objects that do not appear in t.

In general, a trace of a component may still contain impossible events. For example, consider the following execution of a component $\mathcal{C}$

$$\sigma_0 \xrightarrow{new(i,e)} \sigma_1 \xrightarrow{new(i,e')} \sigma_2 \xrightarrow{e.op_1(i,i)} \sigma_3 \xrightarrow{i.op_2(e,e')} \sigma_4$$

inducing the trace t

$$e.op_1(i, i) \cdot i.op_2(e, e')$$

with $i \prec_t e$ and $i \prec_t e'$. The root of the creation tree of t is the internal port i with both the external ports e and e' as children. However, the last communication appearing in t is not possible because the port e cannot possibly know the port e'. To exclude this case, we introduce the following notion of knowledge.

Definition 5. *Given a trace t, we define the set $\kappa(t, o)$ of objects that an object o* may knows *by induction on t:*

$$\kappa(\epsilon, o) = \{o\} \cup \{o' | o \prec_t o'\}$$

$$\kappa(t \cdot o'.op(o'', v), o) = \begin{cases} \kappa(t, o) \cup \{o'', v\} & o = o' \text{ and } v \in Obj \\ \kappa(t, o) \cup \{o''\} & o = o' \text{ and } v \notin Obj \\ \kappa(t, o) & \text{otherwise} \end{cases}$$

Intuitively, an object o knows itself, all objects it created, and those objects it received via some triggered operation. The above definition does not depend on a trace to be generated by an execution of a component. Note however, that given a trace t of a component $\mathcal{C}$ if an external port $e' \in \kappa(t, e)$ then the external port e' *may* also have knowledge of the external port e because an implementation of e and e' may involve the communication of the identity of e to e'. More generally, we can argue in a similar manner that if $e' \in \kappa(t, e)$ then the external objects e and e' may have the same knowledge.

Definition 6. *Given a trace t and a component $\mathcal{C}$, we define a* cluster *of external ports possibly having the same knowledge as an equivalence class of the equivalence relation $\simeq_t$, where $\simeq_t$ is the least equivalence relation such that*

$$e \simeq_t e' \text{ if } e' \in \kappa(t, e)\,.$$

Because objects in a cluster may share their knowledge, we define their shared knowledge $\kappa^*(t, e)$, also called *cluster knowledge*, as

$$\kappa^*(t, e) = \bigcup \{\kappa(t, e') \mid e \simeq_t e'\}\,.$$

We defined clusters only for external ports, because the flow of information of the internal ports is controlled by their respective implementation. For example if i knows e and another external port e' then this in itself does not imply that e may have knowledge of e'. This knowledge can only be obtained by a chain of communications originating from i.

A trace is called executable if external ports communicate only names known by some ports in the same cluster. Formally, we have the following definition.

Definition 7. *Given a component $\mathcal{C}$, a trace t is* executable *if for every prefix $t' \cdot i.op(e, v)$ of t we have that both i and v (if it is an object) are in $\kappa^*(t', e)$. We define $\mathcal{T}(\mathcal{C})$ to be the set of all executable traces of the component $\mathcal{C}$.*

Observe that executable traces are insensitive to the order in which ports are instantiated. Also, because the creation tree of a trace refers only to names that appears in the sequence of observable events (but possibly one, the root), executable traces concerns only with objects that do play a role in an inter-components communication (and not those objects that are created but never used in a communication).

The trace semantics defined above is compositional with respect to component composition.

Theorem 1. *For any two components $\mathcal{C} = \mathcal{C}_1 \oplus \mathcal{C}_2$ we have*

$$\mathcal{T}(\mathcal{C}) = \partial_{\mathcal{C}}(\mathcal{T}(\mathcal{C}_1) \cap \mathcal{T}(\mathcal{C}_2)) .$$

The proof of this compositionality result involves a fairly straightforward generalization of the compositional trace semantics for CSP (see [9]) to our setting.

The next theorem shows the correctness of the above compositional trace semantics with respect to the above may equivalence.

Theorem 2. *For any components $\mathcal{C}_1$ and $\mathcal{C}_2$ with the same provided and required interfaces (not including $ISucess$), if $\mathcal{T}(\mathcal{C}_1) = \mathcal{T}(\mathcal{C}_2)$ then $\mathcal{C}_1 \simeq \mathcal{C}_2$.*

The proof of this theorem follows from the compositionality result in Theorem 1 in a fairly standard manner. In the next section we investigate the converse of the above Theorem: are executable traces fully abstract with respect to may equivalence?

4.1 Trace definability

In order to show that executable traces can be implemented we introduce the notion of extended traces, that is, traces augmented with events for synchronization between external ports, so that they can be justified in terms of what external ports may know.

Definition 8. *An* extended trace *t of an executable trace t' of a component $\mathcal{C}$ is a trace with the same creation tree of t' and that extends the sequence of events of t' with additional external communication events of the form $e.op(e', v)$ (where op may denote a possible operation of an implementation of e i.e., an operation that is not specified by the required interface to which e belongs).*

In an extended trace the events themselves can be justified directly in terms of the exact knowledge of the ports (i.e. the objects created or received via a triggered operation).

Definition 9. *An* abstract implementation *of an executable trace is an extended trace t of an executable trace of a component $\mathcal{C}$ such that for every prefix $t' \cdot o.op(e, v)$ of t both objects o and v are in $\kappa(t', e)$.*

The following lemma can be proved in a straightforward manner by implementing a protocol for broadcasting new knowledge to all external ports within a cluster.

Lemma 1. *Every executable trace of a component $\mathcal{C}$ has an abstract implementation.*

We arrived at the following definability result.

Theorem 3. *For every executable trace $t \in \mathcal{T}(\mathcal{C})$ of a component $\mathcal{C}$ there exists another component $\mathcal{C}'$ with as provided interfaces those required by $\mathcal{C}$ and such t is also an executable trace of $\mathcal{C}'$.*

The sketch of the proof of the above theorem is as follows. Because t is an executable trace it has an abstract implementation by Theorem 1. Further, we can reduce the latter trace to a sequence s by prefixing it with creation events of the form $new(o, u)$ for each pair of names o and u with $o \prec_t u$, and new the constructor associated to the class of which u is an instance. This way, viewing the creation events above as a binding operator in the second argument, all names occurring in the sequence s are bound but for the root of the tree of creation.

Next, for every external port e in the new sequence s we define an implementation $S(e, s)$ corresponding with the subsequence s of creation and communication events of s involving e. This implementation uses the object names occurring in s as *instance variables* of the object e. Basically, it is constructed by transforming every event $o.op(e, v)$ into a corresponding operation call $o.op(self, v)$, every event $e.op(o, v)$ into a corresponding trigger $op(o, v)$, every creation event $new(e, o)$ into a corresponding constructor call $c.new(self, o)$, with new the constructor of the class c, for $o \in Obj(c)$, and, finally, the every creation $new(o, e)$ into the trigger $new(o, self)$.

As last step, for every required interface r of the component $\mathcal{C}$, we define the UML state-machine specifying the generic behavior of the class realizing the provided interface r of $\mathcal{C}'$ as the *non-deterministic* choice of the implementations $S(e, t)$, where e ranges over all instances of r appearing in t. By construction we have that t is an executable trace of $\mathcal{C}'$.

5 Trace abstractions

In this section we show that the reverse implication of Theorem 2 does not hold. Therefore executable traces are not fully abstract: there exist may-equivalent components with different sets of executable traces. Moreover, we define trace abstractions for obtaining a fully abstract semantics. We proceed by presenting three typical examples for which full abstraction fails and illustrate the need for respective abstractions on traces.

As a first example, consider a component $\mathcal{C}$ with two required interfaces, r_1 and r_2, both declaring a constructor new. Further, r_1 declares an operation op_1 with a parameter of type r_1, while r_2 declares an operation op_2 with a parameter of type r_1. Let c be a role of the component depending on r_1 and r_2. The transitions of its associated state machine are as follows:

$$l_0 \xrightarrow{/r_1.new(self,x)} l_1 \xrightarrow{/r_2.new(self,y)} l_2 \xrightarrow{/x.op_1(self,x)} l_3 \xrightarrow{/y.op_2(self,y)} l_4$$

Here x is an attribute of type r_1 and y is an attribute of type r_2. Observe that the transition of the above state machine are not guarded and there is no trigger. This state machine generates traces of the form

$$e_1.op_1(i, e_1) \cdot e_2.op_2(i, e_2)$$

with $i \prec e_1$ and $i \prec e_2$, $i \in Obj(c)$, $e_1 \in Obj(r_1)$ and $e_2 \in Obj(r_2)$. Consider now a similar component $\mathcal{C}'$ different from $\mathcal{C}$ in the state machine associated to the class c:

$$l_0 \xrightarrow{/r_1.new(self,x)} l_1 \xrightarrow{/r_2.new(self,y)} l_2 \left\langle \begin{array}{l} \xrightarrow{/x.op_1(self,x)} l_{3a} \xrightarrow{/y.op_2(self,y)} l_{4a} \\ \xrightarrow{/y.op_2(self,y)} l_{3b} \xrightarrow{/x.op_1(self,x)} l_{4b} \end{array} \right.$$

This state machine generates the same traces as the previous one and additionally also traces of the form

$$e_2.op_2(i, e_2) \cdot e_1.op_1(i, e_1)$$

with $i \prec e_1$ and $i \prec e_2$, that differ with the previous ones only with respect to the order of the synchronization on the operations op_1 and op_2. However there is no component that can distinguish these two kinds of traces because the external instances e_1 and e_2 cannot know each other and therefore cannot communicate or synchronize. In other words, the order between these observable events cannot be imposed by the environment because they belong to different clusters.

In general, the order between observable events involving external ports belonging to different clusters cannot be observed in the may-testing semantics. We can abstract from this information by the following closure condition on the traces of a given component.

Definition 10. *Given a component $\mathcal{C}$, a set T of executable traces is closed with respect to the order between events which actively involve external objects belonging to different clusters, if*

$$t \cdot r.op(s, v) \cdot r'.op'(s', v') \cdot t' \in T$$

such that

$$e' \notin \kappa^*(t \cdot r.op(s, v), e),$$

for $e \in \{r, s\}$ and $e' \in \{r', s'\}$, implies

$$t \cdot r'.op'(s', v') \cdot r.op(s, v) \cdot t' \in T\,.$$

This means that we can only swap events which belong to different clusters of the corresponding prefix of the trace, a phenomena typical of asynchronous processes [6]. In our case, however, this captures the dynamic evolution of clusters, which grow monotonically.

As a second example we consider the following two different state machines associated to a role c (with constructor new_c) of a component depending on a required interface r. This interface declares the constructor new_r and an operation op with a parameter typed by r itself. The first state machine creates an unbounded number of external instances of the required interface r by iteratively calling the constructor method new_r and synchronizes with each of them on the operation op:

$$l_0 \xrightarrow{new_c(x,self)/} l_1 \xrightarrow{/r.new_r(self,y)} l_2 \xrightarrow{/y.op(self,x)} l_1\,.$$

Observe that the iteration is expressed by the fact that, after the call of the operation op, the state machine return in the location l_1. The second state machine implements the above iteration via recursion: it recursively generates an unbounded number of port of c. Each of these ports creates an external instance of the required interface r and synchronize with it via the operation op:

$$l_0 \xrightarrow{new_c(x,self)/r.new_r(self,y)} l_1 \xrightarrow{/y.op(self,y)} l_2 \xrightarrow{/c.new_c(\mathtt{self},z)} l_3\,.$$

In term of traces, the component with the first state machine associated to c produces traces of the form

$$e_1.op(i_0, e_1) \cdot e_2.op(i_0, e_2) \cdots e_k.op(i_0, e_k)\,,$$

with $e \prec i_0$ and $i_0 \prec e_n$ for $n = 1, \cdots, k$. On the other hand, the component with the second state machine associated to c produces traces of the form

$$e_1.op(i_0, e_1) \cdot e_2.op(i_1, e_2) \cdots e_k.op(i_{k-1}, e_k)\,,$$

with $e \prec i_0$, $i_{n-1} \prec i_n$ and $i_{n-1} \prec e_n$ for $n = 1, \cdots, k$. Basically the two kinds of traces differ on the identities of the internal ports that create new instances of the required interface r. This difference cannot be observed by another component because each of the external ports e_n's form a different cluster, and objects in different clusters cannot share (and compare) their knowledge.

We can abstract from this difference by, roughly, a cluster-wise renaming of internal instances. Formally, given a component $\mathcal{C}$ we define a relation $t \simeq_\alpha t'$ between the executable traces t and t' if t' results from t by substituting (also in the creation tree) an internal instance i for every occurrence of an other internal instance j, with the same provided interface, in every event which actively involves an external object of a cluster of t. To preserve the dynamic cluster structure of the internal instances, we additionally require that i does not appear in those events which actively involve an object of the cluster. For example, the first trace above can be obtained by the second one by substituting i_0 for i_{n-1}, with $n = 2, \cdots, k$.

Definition 11. *Given a component $\mathcal{C}$, a set T of executable traces is closed with respect to cluster-wise renaming of internal instances,if*

$$t \in T \text{ and } t \simeq_\alpha t' \text{ implies } t' \in T$$

Finally, we abstract from some information about object creation in a trace t that is too specific, because, after all, the only relevant information concerns the dynamic cluster structure of the trace. Consider the following two traces of a component with a provided interface containing the operation op_p and a required interface containing the operation op_r:

$$e.op_p(i, i) \cdot i.op_r(e', e') \cdot i.op_r(e'', e'')$$

one time with creation tree $i \prec e \prec e' \prec e''$, and another time with creation tree $i \prec e$, $e \prec e'$ and $e \prec e''$. They are two different traces that, however, generate the same cluster structure. In general, the object creator of an instance can be replaced by any other object already existing within the same cluster.

Given a component, we therefore introduce an equivalence relation $t \cong t'$ on executable traces that holds if the traces t and t' specify the same sequence of events with the same dynamic cluster structure, i.e., t and t' have for every prefix the same cluster structure. Formally, a prefix t'' of a trace t consists of a prefix of its sequence of events together with a creation tree obtained by restricting that of t to the objects appearing in t''. So, we define $t_1 \cong t_2$ if for every two prefixes t_1' of t_1 and t_2' of t_2 with the same sequence of observable events σ, we have $o \simeq_{t_1'} u$ if and only if $o \simeq_{t_2'} u$, for every two objects o, u appearing in σ.

Definition 12. *Given a component $\mathcal{C}$, a set T of executable traces is closed with respect to to object creation if*

$$t \in B \text{ and } t \cong t' \text{ implies } t' \in B$$

We have arrived at the following definition of the fully abstract trace semantics $\mathcal{T}_a$ for components.

Definition 13. *Given a component $\mathcal{C}$ we define the set $\mathcal{T}_a(\mathcal{C})$ of its abstract traces as the smallest set of executable traces containing $\mathcal{T}(\mathcal{C})$ and being closed with respect the order between events that actively involve external objects belonging to different clusters, and, the cluster-wise renaming of internal instances.*

Correctness is straightforward because the above closure conditions do not affect may-equivalence.

Theorem 4. *For any components $\mathcal{C}_1$ and $\mathcal{C}_2$, $\mathcal{T}_a(\mathcal{C}_1) = \mathcal{T}_a(\mathcal{C}_2)$ implies $\mathcal{C}_1 \simeq \mathcal{C}_2$.*

6 Full abstraction

In this section we sketch a proof of full abstraction for the above semantics of components. Full abstraction is expressed by the following theorem.

Theorem 5. *May equivalent components have the same set of abstract traces.*

In the following we give a sketch of the proof that proceeds by contraposition. Suppose $\mathcal{C}_1$ and $\mathcal{C}_2$ are two may-equivalent components with different sets of abstract traces. Without loss of generality, let $t \in \mathcal{T}_a(\mathcal{C}_1) \setminus \mathcal{T}_a(\mathcal{C}_2)$. Since abstract traces are executable, by Theorem 1 there exists an abstract implementation t' of t.

This means that t' contains some protocol for broadcasting new knowledge so that the actual knowledge of external objects coincides with their possible knowledge (details are straightforward and omitted here).

Next we reduce the trace t' to a sequence σ by prefixing it with creation events of the form $new(o, u)$ for each pair of names o and u with $o \prec_{t'} u$, and new the constructor associated to the class of which u is an instance.

We can enrich the sequence σ with additional communication events modelling a protocol for fixing the order of execution among those events of the sequence involving external instances that belong to the same cluster. This protocol can be described using the mechanism of passing a baton between the external instances of the same cluster as in a relay team. Basically we insert between two synchronization events $s_1.op_1(r_1, v_2)$ and $s_2.op_2(r_2, v_2)$ involving two external ports e_1 and e_2 in the same cluster as sender or receiver of the operations, an external event $e_2.baton(e_1, e_1)$. Consequently, the execution of events of instances that belong to the same cluster is sequentialized.

Finally, in order to obtain an observable difference in the may testing semantics, we assume that each cluster of external objects in σ will create an instance o of the provided interface $ICluster$ and call after its last event the operation $cluster$ of o indicating the successful termination of the cluster. As a consequence, there will be as many instances

of the class $ICluster$ as actual clusters in the sequence σ. When the last instance is created, an instance of the required interface $ISuccess$ is created and its operation succ is called.

On the basis of the above sequence σ, we can construct a distinguishing components $\mathcal{C}$ with as provided interface those required by C_2 plus the interface $ICluster$ and as required interfaces those provided by C_2 plus the interface $ISuccess$. The two interfaces $ISuccess$ and $ICluster$ will be used to indicate the successful termination of all the clusters of external objects of σ. In the state machines associated to the classes realizing the provided interfaces of $\mathcal{C}$ we will use a pseudo-code to describe guards and primitive operations, in particular we will use test for equalities, assignments composed by standard operators like sequential composition ; and if-then-else.

Implementing abstract behaviors: First we discuss how to express in pseudo code the abstract behavior of an external instance e in σ. Let $\sigma\restriction_e$ denote the projection of σ onto all the events actively involving the external instance e (as sender, receiver, or creator). Let $\mathcal{R}(\sigma) = \{o_1, \ldots, o_k\}$ be the name space of all the (internal and external) object identities appearing in σ. For notational convenience, we use these object references also as instance variables in the pseudo code. In order to check for the *local* consistency of the object references stored in the variables of an external instance we introduce for each object reference o a unique fresh variable o' which will be used to store the actual reference received when the object reference o is expected. Let $o \doteq o'$ abbreviate the following pseudo code for for a guard checking the local consistency.

```
if o' = nil
then fail
else  if o ≠ nil
      then  if o ≠ o' then fail fi
      else  for l = 1,...,k do
            if o' = o_l then fail fi
            od
      fi
fi
```

Here fail is to denote the failure of the evaluation of the guard. This guard first checks whether o' is defined (if o' is undefined the statement aborts because the object reference o is expected). If so, we have two possibilities: either the variable o is already initialized, in which case we simply check whether o equals o', or o is not yet initialized, e.g., not yet received, in which case we check whether o' is different from all the other stored object references.

We can now define a concrete state machine $SM(\sigma\restriction_e)$ describing the abstract behavior of e in σ. For technical convenience we use prefixes of $\sigma\restriction_e$ as locations (with ϵ as initial location and $\sigma\restriction_e$ as final one) and specify the transitions of the state machine

by induction on the length of $\sigma\restriction_e$:

$$\sigma'\restriction_e \xrightarrow{/o.op(self,v)} (\sigma' \cdot o.op(e,v))\restriction_e$$

$$\sigma'\restriction_e \xrightarrow{/c.new(self,o)} (\sigma' \cdot new(e,o))\restriction_e$$

$$\sigma'\restriction_e \xrightarrow{[o \doteq o' \text{ and } v \doteq v']op(o,v)/} (\sigma' \cdot e.op(o,v))\restriction_e$$

The state machine $SM(\sigma\restriction_e)$ is thus obtained by a straightforward transformation of the events of $\sigma\restriction_e$ into corresponding actions. The third clause describes the call of a constructor method new which involves the storage of the newly cerated instance in the variable o, with $o \in Obj(c)$. In case of reception of an operation the guards additionally involves a check that the received object references do agree with the corresponding stored ones. Note that thus $SM(\sigma\restriction_e)$ checks only the local consistency of the name space of e. However the encoded protocol for broadcasting new knowledge to all the (external) objects belonging to one cluster will ensure also the global consistency of the name space of the cluster, i.e., any two external objects e and e' belonging to the same cluster assign the same value to any (private) instance variable $o \in \mathcal{R}(\sigma)$. Note however that we cannot ensure that this value is actually the expected object reference o itself!

Implementing the required interfaces: For every required interface r of the given component $\mathcal{C}_1$ we can define its implementation as a non-deterministic choice between the state machines $SM(\sigma\restriction_e)$, where e is an instance of r appearing in σ. However, for a full abstraction result, we also need a mechanism which allows such an instance to select its own 'predestined' behavior. The only way we know to implement such a selection is by means of a restricted use of *static class variables*: for each instance e of a required interface r, we introduce a static class variable $r.e$.

Static class variables are variables associated with a class and shared by all its instances only. In languages like Java, static class variables introduce another form of communication besides message passing. Here this means that we associate to each class c a special object with identity c containing the class variables of c. This means that the state transformations associated with primitive operations are not allowed to read and modify the instance variables of the object associated with the class of the instance executing the primitive operation call. In general we want static variables to have no influence on the knowledge of an object (so that two instances of the same class need not necessarily to know each other). This can be enforced by requiring that information stored in static class variables cannot be used in communications between objects, but can only be written and read for private purposes by any instance of a class. More syntactically, we can obtain this by allowing static class variables to appear only in guards (recall that guard evaluation has no side effect) and as parameters of a trigger (so to get assigned to a value). Static variables, however, cannot be communicated, and hence cannot appear neither as parameters of operation calls nor used by a state transformation associated by a primitive operation.

Let $e_1, \ldots e_l$ be the instantiations of r appearing in σ *in that order*. The following state machine with l_0 as initial location allows each instance e_i to select the right lo-

cation $\sigma\restriction_{e_i}$ where to continue the behavior of the e instance of r by means of a guard preceding the constructor trigger new_r:

$$l_0 \left\langle \begin{array}{l} \xrightarrow{[o_1 \doteq o_1' \text{ and } r.e_2{=}nil \text{ and } \cdots \text{ and } r.e_l{=}nil] new_r(o_1, r.e_1)/} \sigma\restriction_{e_1} \\ \xrightarrow{[o_2 \doteq o_2' \text{ and } r.e_1{\neq}nil \text{ and } r.e_3{=}nil \text{ and } \cdots \text{ and } r.e_l{=}nil] new_r(o_2, r.e_2)/} \sigma\restriction_{e_1} \\ \vdots \\ \xrightarrow{[o_l \doteq o_l' \text{ and } r.e_1{\neq}nil \text{ and } \cdots \text{ and } r.e_{l-1}{\neq}nil] new_r(o_l, r.e_l)/} \sigma\restriction_{e_l} \end{array} \right.$$

Note that static class variables are assigned to the identities of the instances of the class r. Further they do not introduce shared variable concurrency because in the above transitions guard evaluation and trigger (and hence the corresponding test and assignment of static variables) are executed atomically.

May testing It still remain to implement the class realizing the provided interfaces $ICluster$ The following state machine is associated to the class realizing the interface $ICluster$ so to ensures that only the last instance of $ICluster$ will create an instance of the interface $ISuccess$, thus indicating that all clusters of external objects have terminated successfully. Again, we use static class variables for the instances of $ICluster$ to 'count' how many instances have already been created (i.e. how many clusters have successfully terminated).

Assuming that the initial trace t contains m clusters of external objects, let $succ_i$, $i = 1, \ldots, m-1$ be $m-1$ static class variables of $ICluster$ (writing for simplicity $succ_i$ instead of $ICluster.succ_i$) in the following state machine associated to it:

$$l_0 \left\langle \begin{array}{l} \xrightarrow{[o_1 \doteq o_1' \text{ and } succ_2{=}nil \text{ and } \cdots \text{ and } succ_{m-1}{=}nil] new(o_1, succ_1)/} l_1 \\ \xrightarrow{[o_2 \doteq o_2' \text{ and } succ_1{\neq}nil \text{ and } succ_3{=}nil \text{ and } \cdots \text{ and } succ_{m-1}{=}nil] new(o_2, succ_2)/} l_2 \\ \vdots \\ \xrightarrow{[o_m \doteq o_m' \text{ and } succ_1{\neq}nil \text{ and } \cdots \text{ and } succ_{m-1}{\neq}nil] / ISuccess.new(self, x)} l_m \end{array} \right.$$

$$l_m \xrightarrow{/x.success(self,x)} l_{end}$$

By construction, an instance of $ISuccess$ will be created only after all events of each cluster in the trace t have occurred. Its identity is stored in x and the creator moves in a location from where it calls the operation $success$ of x. Since $ISuccess$ is the only required interface of $C \oplus C_2$, the latter call will generate the only observable event $e.sucess(i, e)$, where $e \in Obj(ISuccess)$ and $i \in Obj(ICluster)$.

Full abstraction: By construction it follows that $(\mathcal{C} \oplus \mathcal{C}_1)\downarrow$. Furthermore, by construction $(\mathcal{C} \oplus \mathcal{C}_2)\downarrow$ implies $t \in \mathcal{T}_a(\mathcal{C}_2)$. The latter follows basically because the context $\mathcal{C}$ forces $\mathcal{C}_2$ to behave as t up-to the closure conditions.

7 Conclusion and future work

We have presented a semantics specification of the behavior of UML-based components that is fully abstract with respect to may equivalence. To focus on the semantic issues involved we have chosen for simplified version of UML class diagrams, object diagrams,

state machines and components. However the concepts used are first step towards a semantic approach integrating the several diagrams present in UML. We have applied similar techniques to an extension of the concurrent object calculus with classes [3] and to a sequential object calculus with classes [4]. Both calculi do not consider class inheritance. In fact, and contrary to [16], we do not believe that our result can be applied to an object calculus with inheritance because of the fragile base class problem [21].

Our full abstraction result relies on the static class variables for the construction of the behavior to be associated with a class. The are the key mechanism that allows an object to select its own predestined behavior among those of all instances of a class. Without them we do not know how to construct the behavioral specification of a class from the set of behavior of all its instances. One possibility that we have explored [4] in the context of the object calculus with classes [2], is to restrict it to sequential objects.

The results introduced in this paper are robust enough to support an extension of the state-machine with class name passing, allowing processes to create instances of classes known only at run-time, a form of very late binding typical of component-based systems [22]. Further work is needed for extensions of our result to support more advanced features like inheritance hierarchies, and dynamic class allocation. The first will introduce another way to cross the component borderline, whereas dynamic allocation of behavior to classes (e.g., as studied in [13]) will make this borderline dynamic.

Our fully abstractness result is relevant for and applicable to the generation of test suites for systems of objects. It shows first of all which tests, as sequences of messages, are in fact the same (so it is relevant for defining a effective test suite). Moreover, it shows that to what extent we can abstract from the identities of the test objects. It is future work to apply our result to the theory of testing systems of objects in class based language.

Acknowledgements Thanks to the anonymous referees and Rocco De Nicola for their comments and suggestions that have improved the paper. This work benefited from discussion with Willem-Paul de Roever and other members of the NWO/DFG bilateral project MobiJ.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. E. Ábrahám, M.M. Bonsangue, F.S. de Boer, and M. Steffen. A Structural Operational Semantics for a Concurrent Class Calculus. Tech. rep. 0307 of the Univ. of Kiel, 2003.
3. M. Steffen, E. Ábrahám, M.M. Bonsangue, F.S. de Boer. Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes In *Proc. ICTAC 2004*, vol 3704 of LNCS, pp. 38-52. Springer, 2005.
4. E. Ábrahám, M.M. Bonsangue, F.S. de Boer, A. Grüner, and M.Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In *Proc. FMCO 2004*, vol. 3657 of LNCS, Springer, 2005.
5. F.S. de Boer, M.M. Bonsangue, and J. Guillen-Scholten. Components: From object to mobile channels. In H. Jifeng and Z. Liu (eds.), *Mathematical Frameworks for Component Software – Models for Analysis and Synthesis*, The World Scientific, 2005.
6. M. Boreale, R. De Nicola, and R. Pugliese. Trace and Testing Equivalence on Asynchronous Processes. *Information and Computation*, 172(2):139-164, 2002.

7. F.S. de Boer and M.M. Bonsangue. A compositional model for confluent dynamic data-flow networks. In *Proc. MFCS*, vol. 1893 of LNCS, Springer 2000.
8. M. Boreale and R. de Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120:279–303, 1995.
9. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
10. K. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002.
11. T. Clark, A. Evans, and E. Kent. The metamodelling language calculus: foundation semantics for UML. In *Proc. FASE 2001*, vol.2029 of LNCS pp. 17–31, Springer 2001.
12. W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A formal semantics of concurrency and communication in Real-Time UML In *Proc. FMCO 2002*, vol. 2582 of LNCS, Springer 2003.
13. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle II. *ACM ToPLaS* 24(2):153–191, 2002.
14. M. Hennessy. A fully abstract denotational semantics for the π-calculus. *Theoretical Computer Science*, 278(2):53-89, 2002.
15. M. Hennessy and R. de Nicola. Testing equivalence for processes. *Theoretical Computer Science*, 34:83-133, 1984.
16. A. Jeffrey and J. Rathke. A Fully Abstract May Testing Semantics for Concurrent Objects. In *Proc. of the 17th LICS*, pp. 101-112. IEEE Computer Society Press, 2002.
17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.
18. Object Management Group, *UML 2.0 Superstructure (Final Adopted specification)*. Document – ptc/03-08-02, August 2004.
19. G. Övergaard Formal Specification of Object-Oriented Meta-Modelling. In *Proc. FASE 2000*, vol. 1783 of LNCS, Springer 2000.
20. B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
21. A. Snyder. Encapsulation and inheritance in object-oriented programming. In *Proc. OOPSLA*, pp. 38–45, SIGPLAN Notices 21:11, 1986.
22. C. Szyperski, D. Gruntz and S. Murer *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.