Compositional Operational Semantics of an UML-Kernel-Model Language

Harald Fecher¹, Marcel Kyas¹, Frank S. de Boer², and Willem-Paul de Roever³

¹ Christian-Albrechts Universität zu Kiel, Germany
{hf,mky,wpr}@informatik.uni-kiel.de
² Centrum voor Wiskunde en Informatica (CWI), Netherlands
F.S.de.Boer@cwi.nl

Abstract. It is important to have a formal semantics for the object-oriented modelling language UML. Furthermore, it would be fine to have a compositional semantics in order to model open distributed systems and component-based systems. For this reason we present a compositional operational semantics for the state machines associated to a class diagram. We first describe the meaning of objects, then the meaning of sets of objects that contain only one thread of control (these sets are called activity groups) and finally, we give the meaning how those activity groups interact with each other. All these semantics are given in terms of labelled transition systems.

1 Introduction

UML [12,13] is a widely used diagrammatic language for modelling object-oriented systems. A formal semantics of UML is necessary:

– To enable the implementation of a simulator or compiler of the behavioral description of an UML model.
– To determine if an implementation is correct w.r.t. its specification.
– To prove properties of the model.
– To show (or falsify) that the concepts of UML lead to the expected behavior. For example, does the run-to-completion-step semantics [13] ensure reentrance freeness, or is the (necessary) failing of the synchronous hypothesis [1] adequately handled?
– To detect inconsistencies in UML’s concepts.

We concentrate on the class diagrams and state machines of UML. Class diagrams are used to specify the static structure of the objects and data types, whereas state machines are used to specify their dynamic behavior. UML state machines have evolved from statecharts [7] and their object-oriented version [8]. State machines support synchronous and asynchronous communication using a language inspired by synchronous data-flow languages to specify the reactions

* Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2)
to communication. The semantics of state machines is based on the run-to-completion step assumption: events are dispatched and processed by a state machine one at a time. An event can only be dispatched, if the processing of the previous event is fully completed. This assumption allows the translation of each state machine to a semantically equivalent flat state machine [3]. A flat state machine is a state machine without hierarchical states.

The run-to-completion step assumption was refined within the Omega project [14] in the following way:

**Active objects** An active object is like an event-driven task, which processes its incoming requests one at a time. It owns its own thread of control and is equipped with a dispatcher, which selects an event from its incoming queue and dispatches it to a state machine for processing.

**Passive objects** A passive object is only reacting to incoming requests on behalf of an active object. It does not have its own thread of control. It is owned by exactly one active object and only this active object may dispatch requests to it.

**Activity groups** A thread is identified with its active object and no object is shared by two threads of controls. This allows the grouping of objects into activity groups, an active object with its passive objects, which encapsulates the thread of control. It is like a process with its own private (structured) data, that is addressed by all the names of its constituent objects.

**Systems** A generally open system is build from interacting active objects.

**Coarse interleaving** We do not allow that a run-to-completion step is pre-empted, i.e., each run-to-completion step is atomic for a state machine.

We use a subset of UML’s action language consisting of assignments to update the instance variables of an object, emission of signals, calling of an operation, sending a reply to an operation call, and object creation.

Semantics w.r.t. the above approach [4, 9] were developed within the Omega Project. Unfortunately, they are not compositional. Having a compositional semantics is not only theoretically pleasing, but can also provide a sound basis for modelling open distributed systems and component-based systems. A compositional description of open distributed systems allows the definition of each part of the system without knowledge of the behavior of the other parts, and still can provide a semantics of a complete system only through the knowledge of the semantics of the parts.

As observed in [15] the interference between methods and reading or updating of attributes of another object with respect to the execution of other objects is not understood, because the notion of concurrency, atomicity of actions, and interleaving is not precisely defined. Therefore, we do not consider method calls and reads or updates of other object attributes in this paper. This topic will be addressed in future.

We define in a straightforward way an operational semantics in terms of labelled transition systems (LTS) for single objects. This semantics is directly derived from the state machine assigned to an object. We identify many semantic variation points while defining the semantics of an activity group. For
example, it is not obvious to define how and when control is passed between objects and how activity groups behave when an object returns a value. We give a compositional operational semantics in terms of LTS for activity groups. We argue that the semantical decisions we made are reasonable by showing that control inside activity group passes deterministically. Finally, we present a compositional semantics for collections of activity groups, again in terms of LTS. We omit type information, because it is standard and we focus on the behavioral semantics of state machines.

Our approach differs from most other approaches to the semantics of UML, see the conference proceedings [5] for examples, since our semantics integrates different modeling concepts: class diagrams and state machines.

In particular, in [10] the semantics of a single state machine is defined by translating it to hierarchical automata [11]. In [2] the semantics of state machines is defined by translation to abstract state machines [6]. Both semantics ignore class diagrams and object structures.

Most related to our approach is [15], where a semantics for state machines and class diagrams is given. The semantics in that paper does not consider the communication between different objects and is not compositional. Similar to our paper, it contains a list of problems in the definition of the semantics of state machines in UML 1.3.

All cited papers fail to define a compositional semantics.

The outline of our paper is as follows. In Section 2 we place our work into a scientific context. Section 3 introduce the sublanguage of UML considered by us. The semantics of the static description is given in Section 4, e.g. objects and activity groups are introduced. The operational semantics is defined in Section 5, where also the decisions points are worked out. The statement that control passes deterministically is also contained in Section 5.

2 Positioning of our Research

The highest level of specification for real-time embedded systems ever obtained is that for the synchronous languages ESTEREL, LUSTRE, and SIGNAL, which have basically the same underlying formal model. The proof of coincidence of their logical, operational and physical semantics, as given by Gerard Berry and his coworkers, represents one of the deepest results on formal semantics ever obtained, and represents the ultimate demonstration of the stability of the very concept underlying the semantics of synchronous languages.

These languages are usually characterized as satisfying Berry’s synchrony hypothesis: When compared with the rate of their external requests for communication, programs written in such languages react infinitely fast. Of course, this is an abstraction from the actual execution time of such programs. However, it can be argued that at the highest level of abstraction for real time this is the only hypothesis which makes sense.
In practice, careful mathematical analysis may be required to justify that programs written in these languages finish with reacting to external requests before the next such request arrives (as was the case when justifying that the software for the TGV, the French high-speed trains, could be modeled using the synchronous model above).

A well-known language which contains synchronous elements is David Harel’s Statecharts, the language interpreted by the StateMate tool. It is synchronous in that it models transitions as being infinitely fast, implying that all execution time is spent in the states of Statecharts. However, as its by now at least fifty possible semantics indicate, there is no agreed mathematical semantics for Statecharts as is the case for the truly synchronous languages above. So now we’re faced with the slightly unsatisfactory situation that its semantics is defined basically by means of a (justifiable) algorithm implemented by the Statemate tool.

Does this imply that Statecharts are “worse” than the purely synchronous languages above? No, Statemate has been designed with a particular application in mind, basically, specifying embedded software for aircraft. And it is clear that such software cannot modeled purely synchronously, because the sheer length of communication lines inside aircraft induces communication delays which cannot be modeled that way. This has led to the introduction of GALS, i.e., models combining synchrony for local levels of computation with asynchrony for characterizing more global interactions.

To the best of our knowledge, notwithstanding important progress made by research groups s.a. those of Albert Benveniste and Paul Caspi, no mathematically satisfactory solutions have been published for combining synchrony and asynchrony which answer the comprehensive needs for modeling software for, e.g., the Airbus line of aircraft.

So we’re faced here with at least two levels of abstraction—of purely synchronous languages and that of the more implementation-oriented partially asynchronous language of Statecharts—and have to face the problem of finding consistent and useful ways of combining synchrony with asynchrony.

UML-charts add a third level of “concretisation” to this spectrum: The hypothesis that transitions between states take no time for their execution is given up, as a result of which external requests may have to be buffered. For calculating reactions upon external requests may now take an nonnegligable amount of time which is larger than that between subsequent external requests. Instead, the familiar notion of a “run-to-completion” semantics is used to express which (sequences of) steps should be regarded as “atomic”.

The present paper constitutes a Gedankenexperiment investigating whether clean mathematical concepts for this new problem area exist, and what such concepts might look like.

Clearly the situation here is worse than for Statecharts for which at least a functioning semantics exists (implemented in the StateMate tool and published by Harel and Naamad). For, in case of UML, no such universally accepted yardstick exists at all, whereas the UML documentation is certainly not precise enough to characterize a clearly delineated semantics. Hence, our study can never
be more than an experiment, be it that we subjugate ourselves to the yardsticks of (1) a compositional description using (2) a commonly accepted mathematical canon.

3 The Kernel Language

The kernel language of UML used in this paper is based on the language described in [4] and was developed in the context of the Omega project. We further simplify this language in order to focus on the behavior of state machines and to avoid the interference between method calls.

We only allow classes which declare attributes, signal receptions (which declare the kind of signals an instance of the class is willing to receive), and one-to-one associations between classes. Generalization between classes is not relevant for this paper, since we do not include methods, and state machines are not inherited.

The dynamic description of a model is provided by state machines. We only consider flat state machines. A state machine is associated to an object and defines the object’s behavior. Objects communicate by exchanging of signals, which may be asynchronous or synchronous. As in UML 1.x we call synchronous signals \textit{operation calls}. Operation calls return a value.

We use a simple expression language based on OCL to describe guards and values, and introduce a simple action language consisting of actions to create new objects, update an attributes value, call an operation, send an asynchronous signal, and return a value from an operation call.

In order to formalize the notions, we introduce the following notations: $\mathcal{P}(M)$ denotes the powerset of $M$, $\mathcal{P}_F(M)$ denotes the set of all finite subsets of $M$ and $M_1 \rightarrow M_2$ denotes the set of all partial functions from $M_1$ to $M_2$. The domain of $f : M_1 \rightarrow M_2$ is the set $\{m_1 \mid f(m_1) \text{ is defined}\}$, and is denoted by $\text{dom}(f)$. Moreover, $f(M'_1)$ denotes the set $\{f(m_1) \mid m_1 \in M'_1\}$. We write $f(m_1) = f'(m_3)$, where $f' : M_3 \rightarrow M_2$, to denote that $f(m_1)$ is defined if and only if $f(m_3)$ is defined and if they are defined they have to be equal. By $f : M_1 \rightarrow^*_\_ M_2$ we denote that $f$ is a partial function with a finite domain, i.e., $f : M_1 \rightarrow M_2 \land |\text{dom}(f)| < \infty$.

Suppose $w \in M^*$ then $|w|$ denotes the length of $w$. Furthermore, $w[i]$ denotes the $i$-th element of $w$ and $\pi$ denotes the set of all elements of $M$ that appears in $w$. The sequence concatenation of two strings $w, w' \in M^*$ is denoted by $w \cdot w'$. The sequence concatenation of two sets $W,W' \subseteq M^*$ is given by $W \cdot W' = \{w \cdot w' \mid w \in W \land w' \in W'\}$. The set $M^f \subset M^*$ denotes the set of all finite sequence of $M$ where every element does not appear more than once.

Suppose $f : M \rightarrow M'$ and $m \in M^f$ and $m' \in M^*$ with $|m| = |m'|$. Then $f[m \mapsto m'] : M \rightarrow M'$ is given by

$$f[m \mapsto m'](x) = \begin{cases} m'[i] & \text{if } x = m[i] \\ f(x) & \text{otherwise} \end{cases}.$$
3.1 Static Information

Let \( C \) be a set of class names, with typical element \( c \), and let \( C^{\text{ac}} \subseteq C \) be the set of class-names which generates new activity groups. Elements of \( C^{\text{ac}} \) are called active classes.

Let \( A \) be a set of attributes names\(^3\), with typical element \( a \). The set of all operations is denoted by \( O_p \) (with typical element \( op \)), the set of all signals is denoted by \( S \) (with typical element \( s \)) and the set of all constructors is denoted by \( C_r \) (with typical element \( cr \)), where \( O_p, S, C_r \) and \( A \) are pairwise disjoint.

The signature of operations and signals is given by a function \( \text{ty} : (O_p \cup S) \rightarrow A \). The attributes indicate where the arguments are saved, i.e., where the arguments are stored explicitly in the object. We do not have call by value.

3.2 Action Language

An action is given by

\[
\text{act} ::= a := e \mid a := a'.op(e) \mid a!s(e) \mid \text{ret}(e) \mid a := \text{new}_c :: cr(e)
\]

where \( a, a' \in A \), \( op \in O_p \), \( s \in S \), \( c \in C \), \( cr \in C_r \) and \( e \) is a simple expression, i.e., it contains only primitive functions (elements of a set \( F \)), attribute-names (which are used for the corresponding attribute values) and the constant \( \text{self} \) (which means the object itself). The set of the primitive functions \( F \) has to contain at least the typical boolean-operators (\( \land, \lor, \neg, ... \)), and the typical integer-operators like addition and the identity relation. We assume that every primitive function can be effectively calculated in its interpretation. By our definition, expressions only depend on the local information, i.e., an object cannot obtain information from other objects via expressions. Furthermore, \( e \) denotes a finite sequence of simple expressions. We assume that every action is well typed in the sense that the number of the arguments matches the specification. The set of all actions is denoted by \( \text{Act} \).

The actions have the following intuitive meaning: \( a := e \) assigns to \( a \) the value obtained from \( e \). Action \( a := a'.op(e) \) calls an operation \( op \) with arguments evaluated from \( e \) in the object that is stored in \( a' \) and stores the return value of this call in \( a \). A signal call with arguments evaluated from \( e \) to the object that is stored in \( a \) is denoted by \( a!s(e) \). Signals can also be send to the object itself, which is denoted by \( \text{self!s}(e) \). Operation calls to the object itself always lead to deadlock, hence such an action is omitted. \( \text{ret}(e) \) returns the value of \( e \). Object creation is described by \( a := \text{new}_c :: cr(e) \), where the new object is referred to by attribute \( a \). Furthermore, the new object has to be of class \( c \) and it is initialized via the constructor \( cr \) with the arguments obtained from \( e \). How the constructor \( cr \) behaves depends on the specification of class \( c \). The constant \( \text{self} \) in an expression denotes the identity of the object. It is used to formalize statements such as that attribute \( a \) refers to the object itself, which is specified by the expression \( \text{self} = a \).

\(^3\) Attributes that corresponds to objects are sometimes called references in the literature.
3.3 Flat State Machines

A *guard* is a simple boolean expression and a *guarded trigger* is a conjunction of a trigger event (operator- or signal-name) $t$ an a guard $b$, written syntactically as $t[b]$. The set of all guards are denoted by $\text{Guard}$ and the set of all triggered guards are denoted by $\text{TrigGuard}$.

**Definition 1.** A flat state machine $St$ is a tuple $(Q,T,\ell_s)$ such that

- $Q$ is a finite set of states
- $T \subseteq Q \times (\text{TrigGuard} \cup \text{Guard}) \times \text{Act} \times Q$ is a finite set of transitions and
- $\ell_s \in Q$ the initial state.

A transition with a guard means that the transition can be taken if the guard evaluates to true. A transition with a guarded trigger means that the transition can be taken if the guard evaluates to true and the trigger is available. When a transition is taken its action will be executed.

3.4 Class

A class consists of a number of attributes, a list of operation and signals to which the class can react, a flat state machine in order to describe the dynamic behavior of the object corresponding to the class, and a list of create-methods together with their code in order to describe how new objects of this class are generated.

**Definition 2.** A class is a tuple $(A,Op,S,St,Cr,I_{Cr})$ such that

- $A \in \mathcal{P}_F(A)$, $Op \in \mathcal{P}_F(Op)$, $S \in \mathcal{P}_F(S)$, $Cr \in \mathcal{P}_F(Cr)$.
- Every attribute used by an operator or signal of the class has to be defined in the class, i.e., $\forall t \in Op \cup S : \text{ty}(t) \subseteq A$.
- $St$ is a flat state machine that uses only the names that are specified in the class.
- There is at least one constructor, i.e., $|Cr| \geq 1$ and
- $I_{Cr}$ is a partial function from $Cr$ such that $\text{dom}(I_{Cr}) = Cr$. The image of a constructor describes how new objects are generated depending on arguments (for example, by assigning a value to each attribute of $A$). The description formalism (the range of $I_{Cr}$) can be given differently, which is not further specified here.

The set of all classes are denoted by $\mathcal{C}$. A class interpretation $I_C$ is a function from $\mathcal{C}$ to $\mathcal{C}$.

In the following, we write $(A^c,Op^c,S^c,St^c,Cr^c,I_{Cr}^c)$ for $I_C(c)$, if $I_C$ is clear from the context.
4 The System

4.1 Objects

For every $c \in \mathcal{C}$ let $\mathcal{O}^c$ be an infinite set of object names that are pairwise disjoint. Define $\mathcal{O} = \bigcup_{c \in \mathcal{C}} \mathcal{O}^c$, with typical element $o$, and $\mathcal{O}^{ac} = \bigcup_{c \in \mathcal{C}\setminus\mathcal{C}^a} \mathcal{O}^c$, with typical element $\hat{o}$. Elements of $\mathcal{O}^{ac}$ are called active object names. Let $\pi^{ob}_{ac} : \mathcal{O} \to \mathcal{O}^{ac}$ be a function such that $\pi^{ob}_{ac}(\hat{o}) = \hat{o}$ and $\{o \in \mathcal{O}^c \mid \pi^{ob}_{ac}(o) = \hat{o}\}$ is infinite, for every $\hat{o} \in \mathcal{O}^{ac}, c \in \mathcal{C}\setminus\mathcal{C}^{ac}$. Function $\pi^{ob}_{ac}$ is used to determine to which activity group (defined later) the object belongs. Define $\pi^{ob}_{cl} : \mathcal{O} \to \mathcal{C}$ such that $o \in \mathcal{O}_{\pi^{ob}_{cl}(o)}$, for every $o \in \mathcal{O}$. Function $\pi^{ob}_{cl}$ determines to which class the object-name corresponds.

By $V$ we denote the set of values. In particular they consist at least of values true, false, the natural numbers, the set of object names and a fresh symbol nil for the general object. We assume that the set of primitive functions $\mathcal{F}$ can be effectively calculated in their interpretation, i.e., in $V$. In particular, if $o, o' \in \mathcal{O}^c$ then $o = o'$ means that they are identical names (and not that the objects are considered to be equivalent in some sense).

The set of object-action sequences $\tilde{\mathcal{A}}$ is defined by $\tilde{\mathcal{A}} = (\mathcal{A} \cup \{a := \text{rec} \mid a \in \mathcal{A}\})^*$. An element of $\tilde{\mathcal{A}}$ is typically denoted by $\tilde{\mathcal{a}}$. The object-action sequence of an object describes what the object can do next. If it is empty the object can change its state machine position (for example can react to signals and operation calls), otherwise, the object has to execute its next object action. The object action $a := \text{rec}$ indicates that the object has to wait for a return value, which it will save in attribute $a$.

Definition 3. An object of class $c$ w.r.t. $I_C$ is a tuple $(I, \ell, \tilde{\mathcal{a}})$ where

- $I : \mathcal{A} \to V$ with $\text{dom}(I) = \mathcal{A}^c$,
- $\ell$ is a state machine position from $St^c$ and
- $\tilde{\mathcal{a}} \in \tilde{\mathcal{A}}$.

Let $\mathcal{O}^c_{I_C}$ be the set of all objects of class $c$ w.r.t. $I_C$, where index $I_C$ and $c$ are omitted if this is clear from the context. Elements of $\mathcal{O}$ are usually denoted by $\hat{o}$.

In particular, objects do not know who communicates with them, e.g. which object called the operation. In other words, an object does not know, for example, to whom it shall send the return value. This task and other control tasks is carried out by the activity group to which the object belongs. Another fact is that an object does not know its identity, i.e., which object name it has. This is again controlled by its activity group.

4.2 Activity Groups

An activity group controls a set of objects and gives them identities. This is usually done by a partial function $\sigma : \mathcal{O} \to \mathcal{F} \mathcal{O}$, where the domain of $\sigma$ determines
the object names that belong to the activity group. Furthermore, an activity
group contains exactly one active object, consequently, the creation of an active
object generates a new activity group. Moreover, an activity group:

- Has to restore the signals, since signals communicate asynchronously. A
  queue of signal events (typically denoted by $E$) is used for this storage.
  A signal event consists of a signal name together with an object name indi-
cating to whom the signal was send. It is necessary to store the order of the
received signals, since we follow a first-in-first-out (FIFO) strategy.
- Has to determine to whom the return value will be sent. Therefore, the call-
ers of the objects of the activity groups have to be stored. This is done by using
a function from the object names of the activity group to $\mathcal{O}^*$, where the
leftmost object determine the last object that made an operation call to the
 corresponding object. Such a kind of function is usually denoted by $R$.
- Has to remember the object that has control, i.e., the object that preformed
  the last action. This is necessary, since we have a run-to-completion-step seman-
tics, hence the control (the object which executes) can only be trans-
ferred if the object that has control becomes stable in some sense (stability
is formally defined in Subsection 5.3).
- Has to determine control transfer constraints. In our approach, if an opera-
tion call inside the same activity group takes place, the caller has to get back
control as soon as the callee becomes ‘stable’. Since a lot of further operation
calls can happen after the returning of the value and reaching stability, we
have to store this transfer condition for every object of the activity group.
This is again done by using a function from the object names of the activity
group to $\mathcal{O}^*$. Such a kind of function is usually denoted by $Z$. The leftmost
object of $Z(o)$ determines the object that will get control if $o$ has control
and is ‘stable’.

This is formalized as follows:

**Definition 4.** An activity group $G = (\sigma, E, R, Z, o)$ is an element of
$(\mathcal{O} \rightarrow_F \mathcal{O}) \times (S^v \times \mathcal{O})^* \times (\mathcal{O} \rightarrow_F \mathcal{O}^*) \times (\mathcal{O} \rightarrow_F \mathcal{O}^*) \times \mathcal{O}$ such that:

- $\text{dom}(\sigma) = \text{dom}(R) = \text{dom}(Z)$,
- $o \in \text{dom}(\sigma)$, and
- there exists $\dot{o} \in \mathcal{O}^{ac}$ with $\dot{o} \in \text{dom}(\sigma)$ and $\forall o' \in \text{dom}(\sigma) : \dot{o} = \pi_{ac}(\dot{o}')$. This
unique object name is denoted by $\pi_{ac}^G(\sigma)$.

The set of all activity groups is denoted by $\mathcal{G}$. The domain of an activity group
is given by $\text{dom}((\sigma, E, R, Z, o)) = \text{dom}(\sigma)$.

If no confusion arises, we sometimes use the object’s identity (e.g., $o$) when
we talk about an object (i.e., when we talk, in fact, about $\sigma(o)$).

### 4.3 Systems

Finally, we consider a collection of activity groups:
Definition 5. A system $K$ is a finite collection of activity groups that correspond to different active objects, i.e., $K : \mathcal{O}^{ac} \to \mathcal{G}$ such that $K(\dot{o}) = (\sigma, z, o') \Rightarrow \dot{o} \in \text{dom}(\sigma)$.

The set of all systems is denoted by $K$. The completed domain of $K$ is given by $\text{dom}(K) = \bigcup_{\dot{o} \in \mathcal{O}^{ac}} \text{dom}(K(\dot{o}))$.

5 Structural Operational Semantics

In order to give a compositional semantics, we first describe the semantics of objects with a labelled transition system (LTS). Then these LTSs are used to define the semantics of activity groups also in terms of LTSs. Finally the semantics of a system is given by an LTS, which depends on the LTSs of activity groups. The labels of these three LTSs consist of the execution of signals/operation calls, sending of signals, calling of operations, receiving/sending of return values, and the creation of new objects.

5.1 SOS of Objects

In order to define the operational semantics, we assume that we have a function that calculates the value of an expression w.r.t. the information about local attributes and the interpretation of self, i.e., let $[\_]: \text{EXP} \times (\mathcal{A} \to V) \times \mathcal{O} \to V$ be given. Note that nontermination coincides with false, because we only test guards for true.

The operational semantics of an object is given in terms of a transition system $(\mathcal{O}_o, \mathcal{L}, \rightarrow)$, where $o \in \mathcal{O}^c$ indicates the name of the object (which is used to determine the value of self) and

$$\mathcal{L} = \{\tau\} \cup \mathcal{O}^p \cup \mathcal{S}^v \cup (\mathcal{O} \times (\mathcal{O}^p \cup \mathcal{S}^v)) \cup \text{Ret}^v \cup \{\text{rec}(v) | v \in V\} \cup (\mathcal{C}r^v \times \mathcal{O})$$

where $\mathcal{O}^p = \{\text{op}(v) | op \in \mathcal{O} \land v \in V^* \land |v| = |\text{ty}(s)|\}$, $\text{Ret}^v = \{\text{ret}(v) | v \in V\}$, $\mathcal{S}^v = \{s(v) | s \in S \land v \in V^* \land |v| = |\text{ty}(s)|\}$, and $\mathcal{C}r^v = \{\text{cr}(v) | cr \in \mathcal{C} \land v \in V^*\}$.

Furthermore, we often write $o.y$ instead of the tuple $(o,y)$. Label $\tau$ denotes an internal move. $\text{op}(v)$ ($s(v)$, $\text{rec}(v)$) denotes that the object reacts to an operation call (respectively, to a signal or to receiving a return value). In these cases, the information who is sending is not transferred in the transition label. The activity group has the responsibility that the return value is given back to the callee and not to another object. This encapsulation has an advantage for the modelling of privacy, since objects can only know other objects if this information is explicitly transferred in an argument or by a return value. The call of an operation (the sending of a signal) from the object with arguments $v$ to object (with identity) $o$ is denoted by $o.\text{op}(v)$ (respectively, by $o.s(v)$). The sending of a return value is denoted by $\text{ret}(v)$. Label $\text{cr}(v).o$ denotes that an object with identity $o$ is created using the constructor $\text{cr}(v)$. Here again, the activity group (better, the environment) has to take care that only fresh object names are used. The object
that generates a new object has to know the new object’s identity, since this identity has to be given to an attribute of the generating object.

The transition rules of $\rightarrow_o$ are given in Table 1. In the following, we give some comments on the transition rules of $\rightarrow_o$. A signal or operation can only be executed if the object is in a state where there is an outgoing transition with this trigger and its guard evaluating to true. Whether such a transition exists depends also on the arguments of the signal sent (respectively, on the arguments of the operation call). The arguments are stored in the attributes that are determined by the type function. A signal can be dropped whenever there is no transition with the above described conditions. In this case, the argument values are lost. In particular, we do not forbid a signal or operation execution on the object level if there is a transition with an untriggered guard that evaluates to true. The activity group of the object has to guarantee that this situation will not appear, if so desired. The other transition rules of $\rightarrow_o$ are self-explanatory.

### 5.2 Semantic Decisions

Giving an operational semantics to objects is more or less straight forward. Unfortunately, this is not the case for activity groups. It is general knowledge that a run-to-completion step semantics has to be used, in the sense that activity
groups can only react to external signals or operation calls when the activity group is stable, i.e., it has completed its run. More precisely, an activity group is stable if all its objects are stable, i.e., none of these objects can execute further actions (can only react to signals and operation calls) and none of them have to wait for a return value (in this case the object is called suspended). The formal definition of stability is given in Subsection 5.3. Unfortunately, there is no common agreement how control (the object that may execute next) inside activity groups has to be handled:

- Suppose object (with identity) \( o \) that has control makes an operational call to another object \( o' \) of the same activity group. Then it is clear that the control will immediately pass to \( o' \). But to whom is control passed if \( o' \) gives up control? Will it pass nondeterministically to any non-stable object (of the activity group) or will it pass directly back to \( o \)? We take the later approach.

- Consider the same situation as described above. Then it is not clear when \( o' \) gives up control. Is this the case when \( o' \) becomes stable or when \( o' \) sends back the return value (getting control later in order to finish its executions)? We follow the former approach that control can only be given up if the object is stable (or suspended).

- Suppose object \( o \) makes an operational call to object \( o' \) of a different activity group. Then \( o \) is suspended and still has the control of its activity group. Is it now possible that \( o \) gives up control or does the activity group stay idle until \( o \) receives its return value? The later strategy is used in our approach.

- Who gets control when object \( o \) creates a new passive object \( o' \), which is put into the same activity group? Is there any control movement condition when the object that got control becomes stable? We decided that the newly created object \( o' \) gets control and that it will pass control back to \( o \) immediately after it becomes stable.

Other uncertainties exist concerning the returning of values. Suppose an operational call was made to object \( o \):

- If more than one \( \text{ret} \)-action is executed before \( o \) becomes stable, which value will be sent back? We chose that the first return value will be sent back.

- At which point is the return value sent back, immediately when the object executes the \( \text{ret} \)-action (this only makes sense if the first value is sent back) or when the object becomes stable? We decide that the return value is immediately sent back when the object execute the \( \text{ret} \)-action. In this case, the receiver is not blocked anymore (if it is in a different activity group) and may continue to execute.

- What happens if \( o \) becomes stable without sending a return value at all? Does it sends any value back, does it change the control, or does it lead to a deadlock? We decided that this situation leads to a deadlock.

Now suppose object \( o \) is executed and no operation call to \( o \) took place. What happens if \( o \) wants to execute a \( \text{ret} \)-action? We decided to result in a deadlock, instead of skipping the action.
5.3 SOS of Activity Groups

Firstly, we give a formal definition of stability. In order to determine if an object is stable, we have to know its identity, since guards have to be evaluated.

**Definition 6.** An object \( \hat{o} = (I, \ell, \tilde{\text{act}}) \) is stable w.r.t. its identity \( o \), denoted \( \text{stable}(\hat{o}, o) \), if \( \tilde{\text{act}} = \epsilon \) and \( \hat{o} \xrightarrow{\tau} o \) (which means that no untriggered transition can be taken).

An activity group \( G = (\sigma, E, R, Z, o) \) is stable, denoted by \( \text{stable}(G) \), if no control move\(^4\) has to be done and all its object are stable and do not have to reply to an operational call, i.e. \( Z(o) = \epsilon \land \forall o' \in \text{dom}(\sigma) : \text{stable}(\sigma(o'), o') \land R(o') = \epsilon \).

Furthermore, we assume that we have an interpretation of how created objects are initialized, i.e., for all \( c \in \text{dom}(I_C) \), we have a function \( \{ \}^{c} : V^* \rightarrow O^c I_C \).

The operational semantics of an activity group is given in terms of a transition system \( (G, \mathcal{L}_{\text{AcGr}}, \leftarrow) \), where

\[
\mathcal{L}_{\text{AcGr}} = \{ \tau \} \cup (O \times (((Op^v \cup Ret^v) \times O) \cup S^v)) \cup (((Op^v \cup Ret^v) \times O) \cup S^v) \cup (C^{v}_r \times O^{ac}).
\]

The meaning of these labels is similar to the meaning of the labels of \( \mathcal{L} \). A difference is that \( \mathcal{L}_{\text{AcGr}} \) also contains the object-name of the sending object in the case of operation calls and sending/receiving return values. For example, \( o.(op(v), o') \) denotes that object (with identity) \( o' \) makes an operation call \( op \) in the object \( o \). Whereas \( (op(v), o').o \) denotes that \( o \) reacts to an operation call from object \( o' \). Furthermore, labels w.r.t. the creation of objects are restricted to active objects, since the other objects are generated inside the activity group and are not visible for the environment.

The transition rules of \( \leftarrow \) are given in Table 2, where \( \Phi \subseteq (O \rightarrow_v O^*) \times O \times O \) is defined (for simplification of the transition rules) by

\[
\Phi(\sigma, R, Z, o, o') \iff (o = o' \lor (Z(o) = \epsilon \land R(o) = \epsilon \land \text{stable}(\sigma(o), o))).
\]

\( \Phi(\sigma, R, Z, o, o') \) indicates that \( o' \) can get control if the activity group is equal to \( (\sigma, E, R, Z, o) \) for some \( E \). In the following, we give some comments on the transition rules of \( \leftarrow \). Consider the case that object \( o \) has control and is stable. If \( o \) has to reply to an operation call, the activity group deadlocks. Otherwise, the control is given back to the callee if it is present \( (Z(o) \neq \epsilon) \). If none of the above situations hold, the control is given to any object of the activity group that wants to execute something. External communication is only possible, if the whole activity group is stable. Operation calls inside the activity group are invisible for the environment (\( \tau \) is observed). If an internal operation call takes place, control is immediately passed to the called object. Furthermore, it is implicitly stored in \( R \) that the control has to move back immediately when the called object becomes stable (and has already sent back the return value).

---

\(^4\) A control move means that the activity group gives the control to another object without executing anything.
\[
\begin{align*}
\sigma(o') & \xrightarrow{\sigma(o') = o'} E(\sigma, R, Z, o, o') \\
\text{stable}(\sigma(o), o) & \xrightarrow{\text{stable}(\sigma(o), o) \in \text{dom}(\sigma)} (\sigma, E, R, Z, o) \xrightarrow{\sigma(o') \mapsto o'} (\sigma, E, R, Z, o) \\
R(o') & = \epsilon \quad o' \in \text{dom}(\sigma) \\
\end{align*}
\]

\[
\begin{align*}
\sigma(o') & \xrightarrow{\sigma(o') = o'} E(\sigma, R, Z, o, o') \\
\text{stable}(\sigma(o), o) & \xrightarrow{\text{stable}(\sigma(o), o) \in \text{dom}(\sigma)} (\sigma, E, R, Z, o) \xrightarrow{\sigma(o') \mapsto o'} (\sigma, E, R, Z, o) \\
R(o') & = \epsilon \quad o' \in \text{dom}(\sigma) \\
\end{align*}
\]

\[
\begin{align*}
\sigma(o') & \xrightarrow{\sigma(o') = o'} E(\sigma, R, Z, o, o') \\
\text{stable}(\sigma(o), o) & \xrightarrow{\text{stable}(\sigma(o), o) \in \text{dom}(\sigma)} (\sigma, E, R, Z, o) \xrightarrow{\sigma(o') \mapsto o'} (\sigma, E, R, Z, o) \\
R(o') & = \epsilon \quad o' \in \text{dom}(\sigma) \\
\end{align*}
\]

\[
\begin{align*}
\sigma(o') & \xrightarrow{\sigma(o') = o'} E(\sigma, R, Z, o, o') \\
\text{stable}(\sigma(o), o) & \xrightarrow{\text{stable}(\sigma(o), o) \in \text{dom}(\sigma)} (\sigma, E, R, Z, o) \xrightarrow{\sigma(o') \mapsto o'} (\sigma, E, R, Z, o) \\
R(o') & = \epsilon \quad o' \in \text{dom}(\sigma) \\
\end{align*}
\]

\[
\begin{align*}
\sigma(o') & \xrightarrow{\sigma(o') = o'} E(\sigma, R, Z, o, o') \\
\text{stable}(\sigma(o), o) & \xrightarrow{\text{stable}(\sigma(o), o) \in \text{dom}(\sigma)} (\sigma, E, R, Z, o) \xrightarrow{\sigma(o') \mapsto o'} (\sigma, E, R, Z, o) \\
R(o') & = \epsilon \quad o' \in \text{dom}(\sigma) \\
\end{align*}
\]

Table 2. Structural Operational Semantics of Activity Groups
note that the sending of the return value and giving back control can take place at different time points (the called object has to get stable before giving back control). If an object made an operation call to an object outside the activity group than it can continue its execution immediately after obtaining the return value. An object that has to send a return value without being called by an operation call results in a deadlock. When a return value is received, we do not check if we get the value from the object that we called. This has the advantage to be more flexible w.r.t. mobility extensions.

The reception of a signal will never be blocked. In particular, it will be stored in $E$. The signals will be considered if the activity group becomes stable, where the earliest received signals will be considered first (FIFO-approach). The sending of a signal to a non-existing object results in a deadlock. We can show by an invariant that this never happens. The creation of a passive object, which is done w.r.t. $\{\}_\tau$, leads to an extension of the activity group by this object. Furthermore, the activity group checks if the corresponding object name is valid: it belongs to the activity group identity $(\pi_{ob}(o''') = \pi_{ac}(\sigma))$ and it is fresh $(o'' \notin \text{dom}(\sigma))$. The task to create an active object will be passed to the environment.

Example 1. Suppose $\hat{o}_1, \hat{o}_2 \in \mathcal{O}$ such that $\hat{o}_1 = \{(a, o_2), \ell_1, \epsilon\}, \hat{o}_2 = (\emptyset, \ell_2, \epsilon)$,

$$\ell_1 \xrightarrow{\text{true}\cdot \text{a} := a} \ell_1'$$

$$\ell_2 \xrightarrow{o\cdot \text{true}} \ell_2'$$

and

Some transition steps of the activity group $\left(\{(a, o_2), \{\epsilon, \ell_1, id, id, o_1\}\right)$ w.r.t. $\leftarrow$ are presented in Figure 1, where $id_e = \{(a, \epsilon), (o_2, \epsilon)\}$.

Please note that it is possible to derive an activity group where more than one of its objects is unstable and not suspended. This fact can arise if an object $o$ makes an operation call to another object $o'$ of its activity group. If $o'$ sends the return value, $o$ is no longer suspended. Furthermore, it is possible that both
objects \((o \text{ and } o')\) are unstable. Nevertheless, control passes deterministically inside an activity group, which is stated in Proposition 1. This proposition only holds, because guards of the state machines depend only on local information (on the value of the object’s attributes) and because the attributes of an object can only be changed by the object itself.

**Proposition 1.** Suppose \((\sigma, E, R, Z, o)\) is reachable from a stable activity group via \(\rightarrow\), and the object-name obtained through object creation is uniquely determined. Then for all \(\gamma \in L^{AcGr}\) we have:

\[
\begin{align*}
(\sigma, E, R, Z, o) &\xrightarrow{\gamma} (\sigma', E', R', Z', o') \land \\
(\sigma, E, R, Z, o) &\xrightarrow{\gamma} (\sigma'', E'', R'', Z'', o'') \implies o' = o''.
\end{align*}
\]

**Proof.** The proof is given in the appendix. \(\square\)

This proposition is a hint that our semantical descisions are reasonable. Example 1 illustrates that it is possible that the control can be on different objects after two transition steps. This fact is based on the nondeterministical behavior of state machines.

In order to argue that the range of \(Z\) has to be \(O\)\(^*\), we modify Example 1 in the following way: Assume that state \(\ell_1'\) can accept an operation call \(op\) (and has no guarded transition) and state \(\ell_2'\) has a further trivially guarded transition that makes an operation call \(op\) to \(o_1\). Furthermore, we assume that \(o_1\) got control from an operation call of another object \(o_3\) of the same activity group, and \(o_1\) already answer the call. In this case, the considered activity group is equivalent to the activity group of Example 1, except that the control structure is \(\{(o_1, o_3), (o_2, \epsilon)\}\) instead of \(id\). Moreover, a transition sequence is possible that results in the control structure \(\{(o_1, o_2 \cdot o_3), (o_2, o_1)\}\).

### 5.4 SOS of Systems

The operational semantics of a system is given w.r.t. the object names of the environment \(\hat{O} \subseteq O^{ac}\). That this is necessary is explained as follows. Suppose we communicate with the environment via object name \(o\). Later we can generate a new object with the same object name \(o\). Hence two objects with the same identity exist (one inside the system and one in the environment). This leads to confusion, since communication to \(o\) of the environment can take place to the new created object with identity \(o\). Therefore, only objects that do not belong to the environment (are not elements of \(\hat{O}\)) may be created. The operational semantics of a system w.r.t. \(\hat{O}\) is given in terms of a transition system \((K, L^{\hat{O}}, \xrightarrow{\hat{O}})\), where \(L^{\hat{O}}\) is the set of all labels of \(L^{AcGr}\) that correspond to communication with \(\hat{O}\), i.e.,

\[
L^{\hat{O}} = \{\tau\} \cup (O^{out} \times (((Op^{v} \cup Ret^{v}) \times \hat{O}^{out}) \cup S^{v})) \cup \\
(((Op^{v} \cup Ret^{v}) \times O^{out}) \cup S^{v}) \times \hat{O}^{out}.
\]
Let \( \tilde{\gamma} \in ((\text{Op}^\text{ac} \cup \text{Ret}) \times \mathcal{O}) \cup \mathcal{S}^\text{ac} \) and \( \gamma \in \text{Op}^\text{ac} \cup \text{Ret}^\text{ac} \).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( K(\dot{\phi}) \xrightarrow{\gamma} G )</td>
<td>( K(\dot{\phi}) \xrightarrow{\gamma} G ) ( \pi_{\text{ac}}^\text{ac}(\dot{o}) \notin \hat{O} )</td>
</tr>
<tr>
<td>( K \xrightarrow{\sim}{\hat{O}} K(\dot{o} \mapsto G) )</td>
<td>( K \xrightarrow{\sim}{\hat{O}} K(\dot{o} \mapsto G) ) ( \pi_{\text{ac}}^\text{ac}(\dot{o}) \notin \hat{O} )</td>
</tr>
<tr>
<td>( K(\dot{o}) \xrightarrow{\gamma} G )</td>
<td>( K(\dot{o}) \xrightarrow{\gamma} G ) ( \pi_{\text{ac}}^\text{ac}(\dot{o}) \notin \hat{O} )</td>
</tr>
<tr>
<td>( K \xrightarrow{\gamma}{\hat{O}} K(\dot{o} \mapsto G) )</td>
<td>( K \xrightarrow{\gamma}{\hat{O}} K(\dot{o} \mapsto G) ) ( \pi_{\text{ac}}^\text{ac}(\dot{o}) \notin \hat{O} )</td>
</tr>
<tr>
<td>( K(\dot{\phi}) \xrightarrow{\gamma} G )</td>
<td>( K(\dot{\phi}) \xrightarrow{\gamma} G ) ( \pi_{\text{ac}}^\text{ac}(\dot{o}) \notin \hat{O} )</td>
</tr>
<tr>
<td>( \overline{\mathcal{O}} \cap \mathcal{O} )</td>
<td>( \overline{\mathcal{O}} \cap \mathcal{O} )</td>
</tr>
</tbody>
</table>

Table 3. Structural Operational Semantics of Systems

where \( \mathcal{O}^\text{out} = \{ o \in \mathcal{O} \mid \pi_{\text{ac}}^\text{ac}(o) \in \hat{O} \} \) and \( \overline{\mathcal{O}}^\text{out} = \mathcal{O} \setminus \mathcal{O}^\text{out} \).

The transition rules of \( \sim{\hat{O}} \) are given in Table 3. In the following, we give some comments on the transition rules of \( \sim{\hat{O}} \). Communication via an object name that exists in the environment and in the system leads to a deadlock. It is an invariant that no object name of the environment may be in the system. The constraint \( \pi_{\text{ac}}^\text{ac}(o) \neq \dot{o} \) is only included in some transition rules for readability, since it holds implicitly there.

References


A Proof of Proposition 1

The outline of the proof is the following. First, we define a formula and prove that it is an invariant under every transition step. Thereafter, the invariant is used to conclude Proposition 1.

Before we present the invariant, we define function \( \Delta : (O \rightarrow F \times O) \times O \rightarrow (O \rightarrow F \times O) \times O \) by

\[
\Delta(Z, o) = \begin{cases} 
(Z[o \mapsto o], o') & \text{if } Z(o) = o' \cdot o \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

Furthermore, let \( \delta : (O \rightarrow F \times O) \times (O \rightarrow F \times O) \rightarrow (O \rightarrow F \times O) \) be given by

\[
\delta(R, Z)(o) = \begin{cases} 
o' \cdot Z(o) & \text{if } R(o) = o' \cdot o \land o' \in \text{dom}(Z) \\
Z(o) & \text{otherwise}
\end{cases}
\]

Define the invariant \( \Lambda \subseteq G \) by \( (\sigma, E, R, Z, o) \in \Lambda \iff \text{dom}(\sigma) \supseteq \{ \Delta^i(\delta(R, Z), o) \mid i \in \mathbb{N} \} \supseteq \{ o' \in \text{dom}(\sigma) \mid \neg \text{stable}(\sigma(o'), o') \} \land \forall o' \in \text{dom}(\sigma) : R(o') \in \mathcal{O} \cup \{ \epsilon \} \).

It is easily seen that every stable activity group satisfies \( \Lambda \). Furthermore, it is straightforwardly checked that \( \Lambda \) is an invariant for all \( \rightarrow \rightarrow \) rules, if the following observation is used: If \( (\sigma, E, R, Z, o) \in \Lambda \) and \( Z(o) = \epsilon \) then all its objects have to be stable since \( \{ \Delta^i(\delta(R, Z), o) \mid i \in \mathbb{N} \} \supseteq \{ o' \in \text{dom}(\sigma) \mid \neg \text{stable}(\sigma(o'), o') \} \) holds. Note that stable objects can only react to signals and operational calls, i.e., they cannot execute other events. Hence, all transition rules that contain the expression \( \Phi(\sigma, R, Z, o, o') \) can only be used in these rules if \( o = o' \) holds.

With this observation in mind Proposition 1 is straightforwardly checked by making a case analysis on the considered transition label. \( \square \)