Compositional Operational Semantics of a UML-Kernel-Model Language [1]

Harald Fecher, Marcel Kyas, Willem-Paul de Roever [2,3,4]

Institute for Computer Science and Applied Mathematics,
Christian-Albrechts-Universität zu Kiel, Germany

Frank S. de Boer [5]

CWI Amsterdam, The Netherlands

Abstract

We define a compositional operational semantics for state machines and their composition in UML. Each state machine describes the behavior of an object of a class. If a class of a newly generated object is active, a new activity group, which is a singly-threaded collection of objects, is generated. Communication of state machines between activity groups differs from the one inside an activity group. We introduce (i) two parallel combinators reflecting this difference, which return a SOS given that their arguments are SOS, (ii) an SOS for each state machine regarded in isolation.

Key words: structured operational semantics, UML, activity groups, compositionality, state machines, run-to-completion steps, deterministic passing of control

1 Introduction

UML [9] has become the standard modeling language in industry for object-oriented programming. The behavior of an object of a class in UML is described by a state machine. State machines have evolved from statecharts [5]

[1] 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)

[2] Email: hf@informatik.uni-kiel.de
[3] Email: mky@informatik.uni-kiel.de
[4] Email: wpr@informatik.uni-kiel.de
[5] Email: F.S.de.Boer@cwi.nl

and their object-oriented version [6]. Furthermore, a class can be defined as "active" or "passive". In some interpretations, e.g., in [3], objects of active classes define a set of objects having a single thread of control. Such sets are called "activity groups". If a class c (either active or passive) is associated with a passive class c' through a so-called "aggregation" association (depicted in the class diagram description), any object o associated with class c may create new "passive" objects of the passive class c'. Moreover, the new objects will belong to the activity group to which o belongs. However, if the associated class is active, creating an object of that (second) class also creates a new activity group to which the new object belongs. Activity groups allow better encapsulation and allow better control of the interaction between objects. For example, mutual exclusion of accessing data that are represented by different objects can be easily achieved.

One crucial property of an activity group is that it has exactly one thread of control. Communication between different activity groups leads to the synchronization of their threads of control. Because a thread is not bound to a single object, semantic complications are introduced. Similar complications have been observed in [1].

By introducing (i) an SOS for state machines in UML, and (ii) two different combinators reflecting the semantics of the parallel combinations of objects of the same activity group and of the parallel combinations of activity groups, we obtain a compositional SOS for state machines and their composition in UML. To our knowledge, this hierarchical combination of compositional semantics for UML-class diagrams combined with UML-state machines is the first one of its kind. Note that a compositional semantics is not straightforwardly defined whenever the assumption that a thread is bound to a single object is dropped, as is, e.g., the case in Java.

A UML semantics that takes class diagrams and state machines into account is given, e.g., by Reggio et al. [10], in terms of labeled transition systems, Große–Rhode [4], in terms of transformation systems, and by Kuske et al. [8], based on graph transformation. None of them consider the concept of activity groups. In [3] a non-compositional semantics of UML, which includes the concepts of activity groups, is given. In [7] a timed semantics for UML state machines with activity groups has been defined, which is not compositional. In both works, each step of an object o explicitly depends on the status of the active object of the activity group to which o belongs.

2 The Kernel Language

The kernel language of UML used in this paper is based on the language described in [3]. We further simplify this language in order to focus on the behavior of state machines and to avoid interference between method calls (which is not well understood [10]).

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. We do not consider generalization between classes.

The dynamic description of models is provided by state machines. We only consider flat state machines; that this is sufficient is a consequence of the run-to-completion-step assumption, as argued in [3]. A state machine is associated with an object and defines the object's behavior. Objects communicate by exchanging signals, which may be asynchronous or synchronous. In our paper as well as in UML versions 1.x, synchronous signals are called *operation calls* and asynchronous signals we simply call *signals*. 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 attribute value, call an operation, send a signal, and return a value from an operation call. Furthermore, we omit type information, because this is standard. The semantics can be straightforwardly extended to cover types.

We use the following notations: $\mathcal{P}_F(M)$ denotes the set of all finite subsets of M and $M_1 \rightharpoonup M_2$ denotes the set of all partial functions from M_1 to M_2. The domain of $f : M_1 \rightharpoonup M_2$ is the set $\{m_1 \mid f(m_1) \text{ is defined}\}$, and is denoted by $\mathrm{dom}(f)$. Moreover, $f(M_1')$ denotes the set $\{f(m_1) \mid m_1 \in M_1'\}$. By $f : M_1 \rightharpoonup_F M_2$ we express that f is a partial function with a finite domain, i.e., $f : M_1 \rightharpoonup M_2 \wedge |\mathrm{dom}(f)| < \infty$.

Suppose $w \in M^\star$, then $|w|$ denotes the length of w and $w[i]$ denotes the i-th element of w. The sequence concatenation of two strings $w, w' \in M^\star$ is denoted by $w \cdot w'$. The set $M^\sharp \subset M^\star$ denotes the set of all finite sequence of M where every element does not appear more than once. Suppose $f : M_1 \rightharpoonup M_2$, $\boldsymbol{m} \in M_1^\sharp$, and $\boldsymbol{m}' \in M_2^\star$ with $|\boldsymbol{m}| = |\boldsymbol{m}'|$. Then $f[\boldsymbol{m} \mapsto \boldsymbol{m}'] : M_1 \rightharpoonup M_2$ is given by

$$f[\boldsymbol{m} \mapsto \boldsymbol{m}'](x) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{m}'[i], & \text{if } x = \boldsymbol{m}[i] \\ f(x), & \text{otherwise .} \end{cases}$$

2.1 Static Information

Let $\mathcal{C}$ be a set of class names, with typical element c, and let $\mathcal{C}^{ac} \subseteq \mathcal{C}$ be the set of *active class* names, i.e., the creation of an object corresponding to an active class generates a new activity group. Let $\mathcal{A}$ be a set of *attribute* names [6], with typical element a. The set of all operations is denoted by $\mathcal{O}p$ (with typical element op), the set of all signals is denoted by $\mathcal{S}$ (with typical element s), and the set of all constructors is denoted by $\mathcal{C}r$ (with typical element cr), where $\mathcal{O}p$, $\mathcal{S}$, $\mathcal{C}r$, and $\mathcal{A}$ are pairwise disjoint. This signature of operations and signals is given by a function $\mathrm{ty} : (\mathcal{O}p \cup \mathcal{S}) \to \mathcal{A}^\sharp$. The signature function indicates how many arguments the operation/signals have and where the arguments are

[6] Attributes that correspond to objects are sometimes called references in the literature.

saved, i.e., where the arguments are stored explicitly in the object.

2.2 Action Language

An *action* is defined by

$$act \ ::= \ a := e \mid a := a'.op(\boldsymbol{e}) \mid a!s(\boldsymbol{e}) \mid self!s(\boldsymbol{e}) \mid ret(e) \mid a := new_c :: cr(\boldsymbol{e})$$

where $a, a' \in \mathcal{A}$, $op \in \mathcal{O}p$, $s \in \mathcal{S}$, $c \in \mathcal{C}$, $cr \in \mathcal{C}r$, and e is a *simple expression*, i.e., contains only *primitive functions* (elements of a set $\mathcal{F}$), attribute names (which are used for the corresponding attribute values), and the constant *self* for denoting 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 $self = a$. The set of the primitive functions $\mathcal{F}$ has to contain at least the typical boolean-operators ($\wedge$, $\vee$, $\neg$, ...), 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, EXP denotes the set of all simple expressions and $\boldsymbol{e}$ 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 Act.

The actions have the following intuitive meaning: $a := e$ assigns to a the value obtained from e. Action $a := a'.op(\boldsymbol{e})$ calls an operation op with arguments evaluated from $\boldsymbol{e}$ 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 $\boldsymbol{e}$, to the object that is stored in a is denoted by $a!s(\boldsymbol{e})$. Signals can also be sent to the object itself; this is denoted by $self!s(\boldsymbol{e})$. Operation calls of an object to that object itself always lead to deadlock, hence such an action is omitted. $ret(e)$ returns the value of e. Object creation is described by $a := new_c :: cr(\boldsymbol{e})$, where the new object is referred to by attribute a. Furthermore, the new object has to be of class c and is initialized via the constructor cr with the arguments obtained from $\boldsymbol{e}$. How the constructor cr behaves depends on the specification of class c.

2.3 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 and a guard b, written syntactically as $t[b]$. The set of all guards are denoted by *Guard* and the set of all triggered guards are denoted by *TrigGuard*.

Definition 2.1 A flat state machine St is a tuple (Q, T, ℓ_s) such that

- Q is a finite set of states
- $T \subseteq Q \times (TrigGuard \cup Guard) \times Act \times Q$ is a finite set of transitions and

- $\ell_s \in Q$ the initial state.

A transition with a guard means that the transition is enabled when the guard evaluates to true. A transition with a guarded trigger means that the transition is enabled when the guard evaluates to true and the trigger is available. When a transition is taken its associated action is executed.

2.4 Class

A class consists of a number of attributes, a list of operations and signals to which the class can react, a flat state machine in order to describe the dynamic behavior of the objects 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.[7]

Definition 2.2 A class is a tuple $(A, Op, S, St, Cr, I_{\mathcal{C}r})$ such that

- $A \in \mathcal{P}_F(\mathcal{A})$, $Op \in \mathcal{P}_F(\mathcal{O}p)$, $S \in \mathcal{P}_F(\mathcal{S})$, $Cr \in \mathcal{P}_F(\mathcal{C}r)$.
- Every attribute used by an operator or signal of the class is defined in the class, i.e., $\forall t \in Op \cup S : \forall i < |\mathrm{ty}(t)| : \mathrm{ty}(t)(i) \in A$.
- 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
- $I_{\mathcal{C}r}$ is a partial function from $\mathcal{C}r$ such that $\mathrm{dom}(I_{\mathcal{C}r}) = Cr$. The image of a constructor describes how new objects are generated depending on arguments. The description formalism (the range of $I_{\mathcal{C}r}$) is not further specified here.

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

In the following, we write $(A^c, Op^c, S^c, St^c, Cr^c, I^c_{\mathcal{C}r})$ for $I_{\mathcal{C}}(c)$, if $I_{\mathcal{C}}$ is clear from the context.

3 Semantical Concepts

3.1 Objects

Let $\mathcal{O}$ be an infinite set of *object names*, with typical element o. By V we denote the set of values. In particular, these consist at least of values true, false, the natural numbers, the set of object names $\mathcal{O}$, 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,

[7] Note that we omit "aggregation" operations in our syntax of class diagrams, since we assume that the behavior described by the state machines respects all associations (including aggregation) specified by the modeler.

if $o, o' \in \mathcal{O}$ then $o = o'$ means that o and o' are identical names (and not that the corresponding objects are considered to be equivalent in some sense). We introduce the following sets, which are later used for the definition of the transition labels:

$$\mathcal{O}p^{\boldsymbol{v}} \stackrel{\text{def}}{=} \{op(\boldsymbol{v}) \mid op \in \mathcal{O}p \wedge \boldsymbol{v} \in V^\star \wedge |\boldsymbol{v}| = |\text{ty}(op)|\}$$
$$\text{Ret}^v \stackrel{\text{def}}{=} \{ret(v) \mid v \in V\} \text{ and}$$
$$\mathcal{S}^{\boldsymbol{v}} \stackrel{\text{def}}{=} \{s(\boldsymbol{v}) \mid s \in \mathcal{S} \wedge \boldsymbol{v} \in V^\star \wedge |\boldsymbol{v}| = |\text{ty}(s)|\}.$$

The set of *object-action* sequences $\widetilde{Act}$ is defined by $\widetilde{Act} \stackrel{\text{def}}{=} (Act \cup \{\epsilon\} \cup \{a := rec(o) \mid a \in \mathcal{A} \wedge o \in \mathcal{O}\})^\star$. An element of $\widetilde{Act}$ is typically denoted by $\widetilde{act}$. The object-action sequence of an object describes what the object can do next. If it is not empty, the object cannot change its state machine position (in particular, it cannot react to signals and operation calls). The object-action $a := rec(o)$ indicates that the object has to wait for a return value from object with identity o. Moreover, the return value will be saved in attribute a.

Definition 3.1 An *object state* w.r.t. $I_{\mathcal{C}}$ is a tuple $(o, c, I, \ell, \widetilde{act}, \underline{o})$ where

- $o \in \mathcal{O}$ and $c \in \mathcal{C}$,
- $I : \mathcal{A} \rightharpoonup V$ with $\text{dom}(I) = A^c$,
- ℓ is a state machine state (its state machine position) from St^c,
- $\widetilde{act} \in \widetilde{Act}$ and $\underline{o} \in \mathcal{O} \cup \{\bot\}$, which indicates if and to whom a return value has to be sent.

Let $\boldsymbol{\mathcal{O}}_{I_{\mathcal{C}}}$ be the set of all object states w.r.t. $I_{\mathcal{C}}$, where index $I_{\mathcal{C}}$ is omitted if it is clear from the context. Elements of $\boldsymbol{\mathcal{O}}$ are usually denoted by $\hat{o}$.

In the following, we call object states just objects. Function $\pi^{ob}_{\mathcal{O}} : \hat{o} \to \mathcal{O}$ yields the corresponding object name of the objects, i.e., $\pi^{ob}_{\mathcal{O}}((o, c, I, \ell, \widetilde{act}, \underline{o})) = o$, and function $\pi^{ob}_{\mathcal{C}} : \hat{o} \to \mathcal{C}$ yields the corresponding class name of the objects, i.e., $\pi^{ob}_{\mathcal{C}}((o, c, I, \ell, \widetilde{act}, \underline{o})) = c$. If no confusion arises, we sometimes use the object's identity (its object name) when we talk about an object.

We introduce the notions of stability of objects, denoting that no internal execution can be made by the objects. This concept is used to ensure the *run-to-completion assumption* of UML, stating that an object may only react to a call if the processing of the previous call is fully completed. In order to define the stability of objects formally, 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., that $[\![_]\!]_{(_,_)} : \text{EXP} \times (\mathcal{A} \rightharpoonup V) \times \mathcal{O} \rightharpoonup V$ is given.

Definition 3.2 An object $\hat{o} = (o, c, I, \ell, \widetilde{act}, \underline{o})$ is *stable*, denoted stable$(\hat{o})$, if $\widetilde{act} = \epsilon$ (no actions are left to be executed) and $\underline{o} = \bot$ (no return value must be sent) and $\forall (\ell, [e], act, \ell') \in T^c : [\![e]\!]_{(I,o)} \neq \text{true}$ (no untriggered transition of

the state machine can be taken[8]).

3.2 Activity Groups

An activity group controls a set of objects with different identities, i.e., represent a parallel composition of objects; exactly one of its object corresponds to an active class. In particular, an activity group collects the incoming signals corresponding to its objects and determines which of its objects are allowed to execute, since no two objects of an activity group may execute at the same time. The incoming events are stored by our approach in an event queue using a first-in-first-out (FIFO) strategy. An object of an activity group should execute internally as long as possible before it can give up control (respecting the run-to-completion assumption), and therefore the activity group needs the information which object is currently running. Furthermore, we follow the approach that the caller gets control back immediately when the callee becomes stable (whenever the caller and callee belong to the same activity group). Therefore, some control-passing strategies have to be encoded in the activity group. This is done by a function, denoted by $\mathcal{M}$, from the object identities of the activity group to a sequence of object identities belonging to the activity group. The leftmost object of $\mathcal{M}(o)$ determines the object that will get control if o has gained control and o is stable (in this case we say that a *control move* takes place). Furthermore, the activity group is defined w.r.t. object names of the environment (i.e., those object names belonging to other activity groups). This is done in order to avoid the creation of an object with an identity that already exists in another activity group. Formally:

Definition 3.3 An *activity group (snapshot)* $G = (\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o)$ w.r.t. $\mathcal{O}' \subset \mathcal{O}$ is an element of $(\mathcal{O}' \rightharpoonup_F \mathbb{O}) \times (\mathcal{S}^{\boldsymbol{v}} \times \mathcal{O}')^\star \times (\mathcal{O}' \rightharpoonup_F \mathcal{O}'^\star) \times \mathcal{O}'$ such that:

- an object name can only be mapped to an object with this identity, i.e., $\forall o \in \mathrm{dom}(\boldsymbol{\sigma}) : o = \pi^{ob}_{\mathcal{O}}(\boldsymbol{\sigma}(o))$,
- there exists exactly one object that belongs to an active class, i.e., $|\{o \in \mathrm{dom}(\boldsymbol{\sigma}) \mid \pi^{ob}_{\mathcal{C}}(\boldsymbol{\sigma}(o)) \in \mathcal{C}^{ac}\}| = 1$,
- $\mathrm{dom}(\boldsymbol{\sigma}) = \mathrm{dom}(\mathcal{M})$ and $o \in \mathrm{dom}(\boldsymbol{\sigma})$.

The set of all activity groups w.r.t. $\mathcal{O}'$ is denoted by $\mathcal{G}_{\mathcal{O}'}$ and the set of all activity groups is denoted by $\mathcal{G}$.

An activity group $G = (\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o)$ is *stable*, denoted by stable(G), if no control move remains to be executed and all its objects are stable, i.e., $\mathcal{M}(o) = \epsilon \wedge \forall o' \in \mathrm{dom}(\boldsymbol{\sigma}) : \mathrm{stable}(\boldsymbol{\sigma}(o'))$. An activity group can be considered as a parallel operator of objects that contain control information concerning

[8] This interpretation is based on the semantics of UML 1.x. In UML 2.0, no transitions of the target state may be taken in the same run-to-completion step. Nevertheless, flat state machines of UML 2.0 can be embedded in our state machines by using only transitions with guarded triggers.

a single thread of control.

3.3 Systems

Finally, we consider a collection (a parallel composition) of activity groups. Let $\mathcal{O}^{out} \subset \mathcal{O}$ denote the object names of the environment, and let $(\mathcal{O}_i)^{i\in\mathbb{N}}$ be an infinite partition of $\mathcal{O} \setminus \mathcal{O}^{out}$ such that $\mathcal{O}_i$ is infinite for every $i \in \mathbb{N}$.

Definition 3.4 A *system* K is a finite collection (a parallel composition) of disjoint activity groups. More precisely, $\mathrm{K} : \mathbb{N} \rightharpoonup_F \mathcal{G}_{\mathcal{O}'}$ such that $|\mathrm{dom}(\mathrm{K})| < \infty$ and $\forall i \in \mathrm{dom}(\mathrm{K}) : \mathrm{K}(i)$ is an activity group w.r.t. $\mathcal{O} \setminus \mathcal{O}_i$.

4 Structural Operational Semantics

4.1 SOS of Objects

The operational semantics of an object w.r.t. $I_\mathcal{C}$ is given in terms of a transition system $(\boldsymbol{\mathcal{O}}, \mathcal{L}, \longrightarrow)$, with

$$\mathcal{L} = \{\tau\} \cup (\mathcal{O}p^{\boldsymbol{v}} \cup \mathrm{Ret}^v) \times \mathcal{O} \cup \mathcal{S}^{\boldsymbol{v}} \cup \mathcal{O} \times (\mathcal{O}p^{\boldsymbol{v}} \cup \mathrm{Ret}^v \cup \mathcal{S}^{\boldsymbol{v}}) \cup \boldsymbol{\mathcal{O}}.$$

Label τ denotes an internal move. Label $(op(\boldsymbol{v}), o')$ (respectively label $s(\boldsymbol{v})$, label $(rec(v), o')$) denotes that the object reacts to (receives) an operation call (respectively, reacts to a signal or upon receiving a return value) from the object (with identity) o'. The call of an operation (the sending of a signal or return value) from the object with arguments v to object o' is denoted by $(o', op(\boldsymbol{v}))$ (respectively, by $(o', s(\boldsymbol{v}))$ and $(o', ret(v))$). In order to enhance readability, we sometimes write $x.y$ for the tuple (x, y). Labels from $\boldsymbol{\mathcal{O}}$ indicate that the corresponding object is created [9] . Here, the activity group has to take care that only fresh object names are used. Furthermore, we assume to have an interpretation of how created objects are initialized, i.e., for all $c \in \mathrm{dom}(I_\mathcal{C})$, we have a function $\{_\}^c_{(_)} : V^\star \rightharpoonup \boldsymbol{\mathcal{O}}^c_{I_\mathcal{C}}$.

The transition rules of $\longrightarrow$ are given in Table 1. They reflect the UML semantics of flat state machines and are independent from the concept of activity groups. In the following, we comment on these rules. The first rule describes the acceptance of an operation call: the object has to be stable and the corresponding state machine has a transition that starts at the current state machine position and that has this operation call as its trigger $((\ell, op[e], act, \ell') \in T^c)$. Furthermore, the guard of this transition has to yield true, where the arguments of the operation call are used for the calculation. This is done using I', where the arguments are already stored in the corresponding attributes. The corresponding attributes are determined by the type

[9] In order to obtain also more compositionality for object creation, the object creation can be moved from the object to the activity group, respectively, to the environment. In this case all necessary information, like class name, creation function, and object name, must appear in the transition label for creation. We have moved object creation to objects in order to increase readability.

Let $\hat{o} = (o, c, I, \ell, \epsilon, \bot)$.

$$\frac{\text{stable}(\hat{o}) \quad (\ell, op[e], act, \ell') \in T^c \quad I' = I[\text{ty}(op) \mapsto \boldsymbol{v}] \quad \llbracket e \rrbracket_{(I',o)} = \text{true}}{(o, c, I, \ell, \epsilon, \bot) \xrightarrow{(op(\boldsymbol{v}), o')} (o, c, I', \ell', act, o')}$$

$$\frac{\text{stable}(\hat{o}) \quad (\ell, s[e], act, \ell') \in T^c \quad I' = I[\text{ty}(s) \mapsto \boldsymbol{v}] \quad \llbracket e \rrbracket_{(I',o)} = \text{true}}{(o, c, I, \ell, \epsilon, \bot) \xrightarrow{s(\boldsymbol{v})} (o, c, I', \ell', act, \bot)}$$

$$\frac{\text{stable}(\hat{o}) \quad \forall (\ell, s[e], act, \ell') \in T^c : \llbracket e \rrbracket_{(I[\text{ty}(s) \mapsto \boldsymbol{v}], o)} \neq \text{true}}{(o, c, I, \ell, \epsilon, \bot) \xrightarrow{s(\boldsymbol{v})} (o, c, I, \ell, \epsilon, \bot)}$$

$$\frac{(\ell, [e], act, \ell') \in T^c \quad \llbracket e \rrbracket_{(I,o)} = \text{true}}{(o, c, I, \ell, \epsilon, \underline{\mathrm{o}}) \xrightarrow{\tau} (o, c, I, \ell, act, \underline{\mathrm{o}})}$$

$$\frac{v = \llbracket e \rrbracket_{(I,o)}}{(o, c, I, \ell, (a := e), \underline{\mathrm{o}}) \xrightarrow{\tau} (o, c, I[a \mapsto v], \ell, \epsilon, \underline{\mathrm{o}})}$$

$$\frac{\boldsymbol{v} = \llbracket \boldsymbol{e} \rrbracket_{(I,o)} \quad o' = I(a')}{(o, c, I, \ell, (a := a'.op(\boldsymbol{e})), \underline{\mathrm{o}}) \xrightarrow{o'.op(\boldsymbol{v})} (o, c, I, \ell, (a := rec(o')), \underline{\mathrm{o}})}$$

$$\frac{\boldsymbol{v} = \llbracket \boldsymbol{e} \rrbracket_{(I,o)} \quad o' = I(a)}{(o, c, I, \ell, (a!s(\boldsymbol{e})), \underline{\mathrm{o}}) \xrightarrow{o'.s(\boldsymbol{v})} (o, c, I, \ell, \epsilon, \underline{\mathrm{o}})}$$

$$\frac{\boldsymbol{v} = \llbracket \boldsymbol{e} \rrbracket_{(I,o)}}{(o, c, I, \ell, (self!s(\boldsymbol{e})), \underline{\mathrm{o}}) \xrightarrow{o.s(\boldsymbol{v})} (o, c, I, \ell, \epsilon, \underline{\mathrm{o}})}$$

$$\frac{I' = I[a \mapsto v]}{(o, c, I, \ell, (a := rec(o')), \underline{\mathrm{o}}) \xrightarrow{(ret(v), o')} (o, c, I', \ell, \epsilon, \underline{\mathrm{o}})}$$

$$\frac{v = \llbracket e \rrbracket_{(I,o)}}{(o, c, I, \ell, ret(e), o') \xrightarrow{o'.ret(v)} (o, c, I, \ell, \epsilon, \bot)}$$

$$\frac{\boldsymbol{v} = \llbracket \boldsymbol{e} \rrbracket_{(I,o)} \quad \hat{o}' = \{cr\}^{c'}_{(\boldsymbol{v})} \quad I' = I[a \mapsto \pi^{ob}_{\mathcal{O}}(\hat{o}')]}{(o, c, I, \ell, (a := new_{c'} :: cr(\boldsymbol{e})), \underline{\mathrm{o}}) \xrightarrow{\hat{o}'} (o, c, I', \ell, \epsilon, \underline{\mathrm{o}})}$$

Table 1
Structural Operational Semantics of Objects

function. If all this is satisfied, the object can accept the operation, resulting in an object where the attribute values are updated and where the action of the state machine transition is stored in order to determine the next execution of the object. Furthermore, the new state machine position is used and the information to which the return value of this operation call has to be sent (the last component of the object) is stored.

The acceptance of signals is handled similarly, except that signals where no transition of the state machine can apply may be dropped, i.e., the object executes this signal without changing itself (in particular, the argument values of the signal are lost). State machine transitions without a trigger yield an internal execution of the object. The remaining rules concern the execution of the possible actions of the object. The return value will be immediately sent back, i.e., there is no waiting for the stability of the object. Consequently, a deadlock occurs if another return value should be sent back and the return value was already sent back or no operational call is executing.

The last rule considers the creation of objects. The created object is determined via the interpretation of the corresponding constructor taking the argument values into account ($\hat{o}' = \{\!\!\{cr\}\!\!\}^{c'}_{(\boldsymbol{v})}$). The reference to this created object is stored in the corresponding attribute ($I' = I[a \mapsto \pi^{ob}_{\mathcal{O}}(\hat{o}')]$). The label of the transition is the created object in order to give the responsibility of storing the created object to the activity group. Please note that the class and object names of an object remain unaffected by every $\longrightarrow$ step.

4.2 SOS of Activity Groups

The operational semantics of an activity group w.r.t. $\mathcal{O}'$ is given in terms of a transition system $(\mathcal{G}_{\mathcal{O}'}, \mathcal{L}^{\text{AcGr}}, \hookrightarrow)$, where

$$\begin{aligned}\mathcal{L}^{\text{AcGr}} = \{\tau\} \cup \overline{\mathcal{O}'} \times \Big(\big((\mathcal{O}p^{\boldsymbol{v}} \cup \text{Ret}^{v}) \times \mathcal{O}'\big) \cup \mathcal{S}^{\boldsymbol{v}}\Big) \cup \\ \Big(\big((\mathcal{O}p^{\boldsymbol{v}} \cup \text{Ret}^{v}) \times \overline{\mathcal{O}'}\big) \cup \mathcal{S}^{\boldsymbol{v}}\Big) \times \mathcal{O}' \cup \\ \{\hat{o} \in \boldsymbol{\mathcal{O}} \mid \pi^{ob}_{\mathcal{C}}(\hat{o}) \in \mathcal{C}^{ac} \wedge \pi^{ob}_{\mathcal{O}}(\hat{o}) \notin \mathcal{O}'\}.\end{aligned}$$

with $\overline{\mathcal{O}'} = \mathcal{O} \setminus \mathcal{O}'$.

The meaning of these labels is similar to the meaning of the labels of $\mathcal{L}$. A difference is that $\mathcal{L}^{\text{AcGr}}$ always contains the identity of the caller and callee (except in the case of asynchronous signals). For example, $o.(op(\boldsymbol{v}), o')$ denotes that object (with identity) o' does an operation call op in the object o, whereas $(op(\boldsymbol{v}), o).o'$ denotes that o' reacts to (receive) an operation call from object o. Furthermore, the possible created objects are restricted to those belonging to active classes, since the other objects will be added directly to the activity group and are not visible to the environment. Moreover, the identity of those visibly created objects have to be outside the scope of the activity group, in order to avoid that two objects exist with the same identity.

The transition rules of $\hookrightarrow$ are given in Table 2 and in Table 3, where o' getC $(\boldsymbol{\sigma}, \mathcal{M}, o)$ indicates that o' can *get control in* the activity group consisting of $(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o)$ for some $\boldsymbol{E}$. A nondeterministic control move is only possible if the object that had control last is stable and no control move is enforced. Formally:

$$o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o) \iff (o = o' \vee (\mathcal{M}(o) = \epsilon \wedge \text{stable}(\boldsymbol{\sigma}(o)))).$$

$$\frac{\boldsymbol{\sigma}(o') \xrightarrow{\tau} \hat{o}' \qquad o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o)}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\tau}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}'], \boldsymbol{E}, \mathcal{M}, o')}$$

$$\frac{\mathcal{M}(o) = o' \cdot \overrightarrow{o} \qquad \text{stable}(\boldsymbol{\sigma}(o))}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\tau}{\hookrightarrow} (\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}[o \mapsto \overrightarrow{o}], o')}$$

$$\frac{\boldsymbol{\sigma}(o') \xrightarrow{(op(\boldsymbol{v}),o'')} \hat{o}' \qquad \text{stable}((\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o)) \qquad o'' \notin \mathcal{O}'}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{(op(\boldsymbol{v}),o'').o'}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}'], \boldsymbol{E}, \mathcal{M}, o')}$$

$$\frac{\boldsymbol{\sigma}(o') \xrightarrow{o''.op(\boldsymbol{v})} \hat{o}' \qquad o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o) \qquad o'' \notin \mathcal{O}'}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{o''.(op(\boldsymbol{v}),o')}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}'], \boldsymbol{E}, \mathcal{M}, o')}$$

$$\frac{\boldsymbol{\sigma}(o') \xrightarrow{o''.op(\boldsymbol{v})} \hat{o}' \qquad o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o) \qquad \boldsymbol{\sigma}(o'') \xrightarrow{(op(\boldsymbol{v}),o')} \hat{o}''}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\tau}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}', o'' \mapsto \hat{o}''], \boldsymbol{E}, \mathcal{M}[o'' \mapsto o' \cdot \mathcal{M}(o'')], o'')}$$

$$\frac{\boldsymbol{\sigma}(o) \xrightarrow{(ret(v),o'')} \hat{o} \qquad o'' \notin \mathcal{O}'}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{(ret(v),o'').o}{\hookrightarrow} (\boldsymbol{\sigma}[o \mapsto \hat{o}], \boldsymbol{E}, \mathcal{M}, o)}$$

$$\frac{\boldsymbol{\sigma}(o') \xrightarrow{o''.ret(v)} \hat{o}' \qquad o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o) \qquad o'' \notin \mathcal{O}'}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{o''.(ret(v),o')}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}'], \boldsymbol{E}, \mathcal{M}, o')}$$

$$\frac{\boldsymbol{\sigma}(o') \xrightarrow{o''.ret(v)} \hat{o}' \qquad o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o) \qquad \boldsymbol{\sigma}(o'') \xrightarrow{(ret(v),o')} \hat{o}''}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\tau}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}', o'' \mapsto \hat{o}''], \boldsymbol{E}, \mathcal{M}, o')}$$

Table 2
Structural Operational Semantics of Activity Groups

In the following, we give some comments on the transition rules of $\hookrightarrow$. If an object that can get control executes an internal action, the whole activity group can execute an internal action, where the executed object is updated ($\boldsymbol{\sigma}[o' \mapsto \hat{o}']$) and remembered (it becomes the last component of the activity group). This is described in the first rule of Table 2, whereas the second rule describes a necessary control move (yields an internal action), which is only possible if the current object is stable.

The next block of rules of Table 2 describe operation-call handling. The first two rules describe the operations of accepting and calling to objects outside the activity group. An operation call from outside can only be accepted if the whole activity group is stable. The third rule considers the internal operation-call handling of activity groups: If an object that can get control makes an operation call inside the same activity group and the called object

$$\frac{o' \in \mathcal{O}'}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{s(\boldsymbol{v}).o'}{\hookrightarrow} (\boldsymbol{\sigma}, \boldsymbol{E} \cdot (s(\boldsymbol{v}).o'), \mathcal{M}, o)}$$

$$\frac{\boldsymbol{\sigma}(o') \overset{s(\boldsymbol{v})}{\longrightarrow} \hat{o}' \quad \boldsymbol{E} = (s(\boldsymbol{v}).o') \cdot \boldsymbol{E}' \quad \text{stable}((\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o))}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\tau}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}'], \boldsymbol{E}', \mathcal{M}, o')}$$

$$\frac{\boldsymbol{\sigma}(o') \overset{o''.s(\boldsymbol{v})}{\longrightarrow} \hat{o}' \quad o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o) \quad o'' \notin \mathcal{O}'}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{o''.s(\boldsymbol{v})}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}'], \boldsymbol{E}, \mathcal{M}, o')}$$

$$\frac{\boldsymbol{\sigma}(o') \overset{o''.s(\boldsymbol{v})}{\longrightarrow} \hat{o}' \quad o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o) \quad o'' \in \text{dom}(\boldsymbol{\sigma})}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\tau}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}'], \boldsymbol{E} \cdot (s(\boldsymbol{v}).o''), \mathcal{M}, o')}$$

$$\frac{\begin{array}{c}\boldsymbol{\sigma}(o') \overset{\hat{o}''}{\longrightarrow} \hat{o}' \quad o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o) \\ \pi^{ob}_{\mathcal{C}}(\hat{o}'') \notin \mathcal{C}^{ac} \quad o'' = \pi^{ob}_{\mathcal{O}}(\hat{o}'') \in \mathcal{O}' \setminus \text{dom}(\boldsymbol{\sigma})\end{array}}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\tau}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}', o'' \mapsto \hat{o}''], \boldsymbol{E}, \mathcal{M}[o'' \mapsto o'], o'')}$$

$$\frac{\boldsymbol{\sigma}(o') \overset{\hat{o}''}{\longrightarrow} \hat{o}' \quad o' \text{ getC } (\boldsymbol{\sigma}, \mathcal{M}, o) \quad \pi^{ob}_{\mathcal{C}}(\hat{o}'') \in \mathcal{C}^{ac} \quad \pi^{ob}_{\mathcal{O}}(\hat{o}'') \notin \mathcal{O}'}{(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\hat{o}''}{\hookrightarrow} (\boldsymbol{\sigma}[o' \mapsto \hat{o}'], \boldsymbol{E}, \mathcal{M}, o')}$$

Table 3
Structural Operational Semantics of Activity Groups (2)

can accept it (it has to be stable to do this), the activity group does an internal execution step. The obtained activity group updates the two objects and gives control to the called object. Furthermore, we enforce a control move back to the caller, when the callee becomes stable. This is done by adapting the control structure ($\mathcal{M}[o'' \mapsto o' \cdot \mathcal{M}(o'')]$). The last block of Table 2 considers the returning of values.

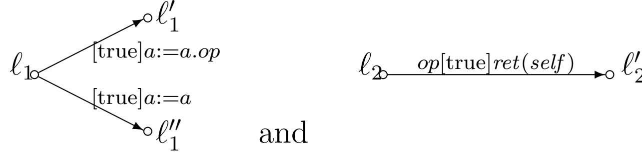
Signal handling is discussed in the first block of Table 3: Signal receiving and execution take place at different time points. When a signal is received (also from the same activity group), it is stored in the event queue. A signal can only be taken (and executed) from the event queue (via the first-in-first-out (FIFO) strategy), if the whole activity group is stable. The time point when this happens is not observable from outside, since it results in an internal action.

The last block of Table 3 considers the creation of objects: If the created object does not belong to an active class ($\pi^{ob}_{\mathcal{C}}(\hat{o}'') \notin \mathcal{C}^{ac}$) it is added to the activity group, gets control, and a control move back to the creator is enforced. Furthermore, it has to be ensured that the object name of the created object is fresh and belongs to the object name of the activity group $\pi^{ob}_{\mathcal{O}}(\hat{o}'') \in \mathcal{O}' \setminus \text{dom}(\boldsymbol{\sigma})$. In case the object belongs to an active class, the creation is observable in order to allow the system to generate a new activity group.

We mention the following observations:

- If the object that had control last is stable and no control move is enforced, control is passed nondeterministically among the willing objects (later we will see that such a situation cannot occur).
- Execution caused by external communication is only possible, if the whole activity group is stable.
- If an object makes an operation call to an object outside the activity group then the whole activity group is blocked, i.e., no control move or execution is possible, until the return value is obtained, in which case the caller can immediately continue its execution.
- If the object that has control has to send a return value without being called by an operation call, then the whole activity group is deadlocked (only signals can be received).
- The reception of a signal will never be blocked.

Example 4.1 Suppose $\hat{o}_1, \hat{o}_2 \in \boldsymbol{\mathcal{O}}$ such that $\hat{o}_1 = (o_1, c_1, \{a, o_2\}, \ell_1, \epsilon, \bot)$, $\hat{o}_2 = (o_2, c_2, \emptyset, \ell_2, \epsilon, \bot)$ with the corresponding state machines



The left state machine describes a nondeterministic choice, where in one case an operation call that does not have any arguments is made to the object stored in a. An operation execution op that sends back its identity is possible in the right hand side state machine. The right hand side state machine is stable, which is not the case for the one on the left hand side.

Some transition steps of the activity group $(\{\hat{o}_1, \hat{o}_2\}, \epsilon, id_\epsilon, o_1)$ w.r.t. $\hookrightarrow$ are presented in Figure 1, where $id_\epsilon = \{(o_1, \epsilon), (o_2, \epsilon)\}$. Note that we write for simplicity the function $\boldsymbol{\sigma}$ as the collections of its images, i.e., we abbreviate $\{(o_1, \hat{o}_1), (o_1, \hat{o}_2)\}$ by $\{\hat{o}_1, \hat{o}_2\}$.

Please note that it is possible to derive an activity group where more than one of its objects is unstable without waiting for a return value. This fact can arise if an object $\hat{o}$ makes an operation call to another object $\hat{o}'$ of its activity group. If $\hat{o}'$ sends the return value, $\hat{o}$ is no longer suspended (i.e., does not wait for the return value). By repeating this procedure ($\hat{o}'$ calls another object ...), it is possible that arbitrary many objects are unstable. Nevertheless, control passes deterministically inside an activity group, which is stated in Proposition 4.2. 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 4.2 *Suppose* $(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o)$ *is reachable from a stable activity group via* $\hookrightarrow$*, and the object name obtained through object creation is uniquely*

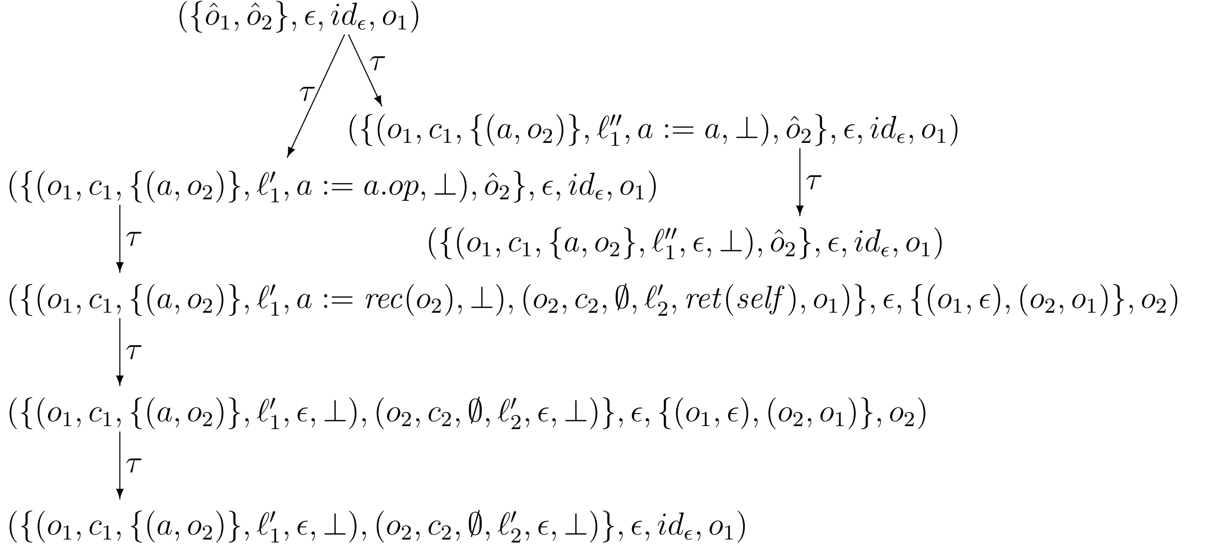


Fig. 1. Some $\hookrightarrow$ Derivations

determined. Then, for all $\gamma \in \mathcal{L}^{\mathrm{AcGr}}$, we have:

$$\left.\begin{array}{l}(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\gamma}{\hookrightarrow} (\boldsymbol{\sigma}', \boldsymbol{E}', \mathcal{M}', o') \wedge \\ (\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \overset{\gamma}{\hookrightarrow} (\boldsymbol{\sigma}'', \boldsymbol{E}'', \mathcal{M}'', o'')\end{array}\right\} \Rightarrow o' = o'' \wedge \boldsymbol{E}' = \boldsymbol{E}'' \wedge \mathcal{M}' = \mathcal{M}''.$$

Example 4.1 illustrates that it is possible that the control can be in different objects after two transition steps. This fact results from the nondeterministic behavior of state machines, i.e., from the nondeterministic behavior of the objects and not from its control structure (activity group).

The necessity that the range of $\mathcal{M}$ has to be $\mathcal{O}'^{\star}$ (instead of consisting of only single object names) is illustrated as follows: Suppose $\hat{o}_1, \hat{o}_2, \hat{o}_3$ are in the same activity group and $\hat{o}_1$ makes an operational call to $\hat{o}_2$. Object $\hat{o}_2$ sends the return value and thereafter makes an operational call to $\hat{o}_3$. Object $\hat{o}_3$ sends the return value, $\hat{o}_3$ does not get stable but $\hat{o}_2$ becomes stable by receiving the value. Then $\hat{o}_3$ makes an operational call to $\hat{o}_2$. Hence, $\hat{o}_2$ has to store the information that it will pass control back to $\hat{o}_3$ and when it gets control again it has to pass control to $\hat{o}_1$ (control has to pass from $\hat{o}_2$ to $\hat{o}_2$, then to $\hat{o}_2$ and then to $\hat{o}_1$).

4.3 SOS of Systems

The operational semantics of a system w.r.t. $\mathcal{O}^{out}$ is given in terms of a transition system $(\mathcal{K}, \mathcal{L}^{out}, \rightsquigarrow)$, where $\mathcal{L}^{out}$ is the set of all labels of $\mathcal{L}^{\mathrm{AcGr}}$ that correspond to communication with $\mathcal{O}^{out}$, i.e.,

$$\begin{aligned}\mathcal{L}^{out} = \{\tau\} \cup (\mathcal{O}^{out} \times (((\mathcal{O}p^{\boldsymbol{v}} \cup \mathrm{Ret}^{v}) \times \overline{\mathcal{O}^{out}}) \cup \mathcal{S}^{\boldsymbol{v}})) \cup \\ ((((\mathcal{O}p^{\boldsymbol{v}} \cup \mathrm{Ret}^{v}) \times \mathcal{O}^{out}) \cup \mathcal{S}^{\boldsymbol{v}}) \times \overline{\mathcal{O}^{out}}).\end{aligned}$$

with $\overline{\mathcal{O}^{out}} = \mathcal{O} \setminus \mathcal{O}^{out}$. The transition rules of $\rightsquigarrow$ are given in Table 4.

Let $\tilde{\gamma} \in ((\mathcal{O}p^{\boldsymbol{v}} \cup \mathrm{Ret}^{v}) \times \mathcal{O}) \cup \mathcal{S}^{\boldsymbol{v}}$ and $\gamma \in \mathcal{O}p^{\boldsymbol{v}} \cup \mathrm{Ret}^{v}$.

$$\frac{\mathrm{K}(i) \overset{\tau}{\hookrightarrow} G}{\mathrm{K} \overset{\tau}{\rightsquigarrow} \mathrm{K}[i \mapsto G]} \qquad \frac{\mathrm{K}(i) \overset{(\gamma,o').o}{\hookrightarrow} G \quad o' \in \mathcal{O}^{out}}{\mathrm{K} \overset{(\gamma,o').o}{\rightsquigarrow} \mathrm{K}[i \mapsto G]} \qquad \frac{\mathrm{K}(i) \overset{s(\boldsymbol{v}).o}{\hookrightarrow} G}{\mathrm{K} \overset{s(\boldsymbol{v}).o}{\rightsquigarrow} \mathrm{K}[i \mapsto G]}$$

$$\frac{\mathrm{K}(i) \overset{o.\tilde{\gamma}}{\hookrightarrow} G \quad o \in \mathcal{O}^{out}}{\mathrm{K} \overset{o.\tilde{\gamma}}{\rightsquigarrow} \mathrm{K}[i \mapsto G]} \qquad \frac{\mathrm{K}(i) \overset{o.\tilde{\gamma}}{\hookrightarrow} G \quad \mathrm{K}(j) \overset{\tilde{\gamma}.o}{\hookrightarrow} G' \quad i \neq j}{\mathrm{K} \overset{\tau}{\rightsquigarrow} \mathrm{K}[i \mapsto G, j \mapsto G']}$$

$$\frac{\mathrm{K}(i) \overset{\hat{o}'}{\hookrightarrow} G \quad j \notin \mathrm{dom}(\mathrm{K}) \quad o' = \pi^{ob}_{\mathcal{O}}(\hat{o}') \in \mathcal{O}_j}{\mathrm{K} \overset{\tau}{\rightsquigarrow} \mathrm{K}[i \mapsto G, j \mapsto (\{(o', \hat{o}')\}, \epsilon, \{(o', \epsilon)\}, o')]}$$

Table 4
Structural Operational Semantics of Systems

In the following, we comment on these rules. Internal execution of an activity group or communication between two activity groups leads to an internal execution. The creation of an object $\hat{o}'$ belonging to an active class, leads to the creation of a new activity group containing the single object $\hat{o}'$, as stated in the last rule. Note that this is only possible if the corresponding object name of $\hat{o}'$ belongs to an activity group that is not used so far ($o' \in \mathcal{O}_j \wedge j \notin \mathrm{dom}(\mathrm{K})$).

5 Conclusion

We have presented a semantics of UML considering class diagrams and (flat) state machines. In particular, we have defined the semantics of sets of objects (activity groups) via SOS using the semantics of its constituting objects; the semantics of an object is defined via SOS. A mechanism which handles the single-threaded passing of control inside an activity group is described. The semantics of parallel activity groups is also defined via SOS using the semantics of its constituting activity groups.

There exists no common agreement concerning the (formal) semantics of UML. Other possible semantics can, for example, differ from ours by considering a non-FIFO strategy for signal reception, control can pass more non-deterministically (for example, when the caller gets stable control does not have to pass directly back to the caller), sending back the return value at the point when the object gets stable (and not immediately). Our semantic decisions have the advantage that control passes deterministically inside activity groups.

Future work concerns the theoretical examination of parallel operators that have special control mechanisms associated with them. For example, is it possible to define transition rule classifications (similarly as for the existence

of transition systems, see, e.g., [2]) such that deterministic passing of control can be guaranteed?

References

[1] Ábrahám, E., F. S. de Boer, W.-P. de Roever and M. Steffen, *An assertion-based proof system for multithreaded Java*, Theoretical Computer Science **331** (2005), pp. 251–290.

[2] Aceto, L., W. Fokkink and C. Verhoef, *Structural operational semantics*, in: J. A. Bergstra, A. Ponse and S. A. Smolka, editors, *Handbook of Process Algebra*, North-Holland, 2001 pp. 197–292.

[3] Damm, W., B. Josko, A. Pnueli and A. Votintseva, *Understanding UML: A formal semantics of concurrency and communication in real-time UML*, in: F. de Boer, M. Bonsangue, S. Graf and W.-P. de Roever, editors, *FMCO 2002*, LNCS **2852** (2003), pp. 71–98.

[4] Große-Rhode, M., *Integrating semantics for object-oriented system models*, in: F. Orejas, P. G. Spirakis and J. van Leeuwen, editors, *ICALP 2001*, LNCS **2076** (2001), pp. 40–60.

[5] Harel, D., *Statecharts: A visual formalism for complex systems*, Science of Computer Programming **8** (1987), pp. 231–274.

[6] Harel, D. and E. Gery, *Executable object modeling with statecharts*, Computer **30** (1997), pp. 31–42.

[7] Hooman, J. and M. van der Zwaag, *A semantics of communicating reactive objects with timing*, Journal on Software Tools for Technology Transfer (2005), accepted for Publication.

[8] Kuske, S., M. Gogolla, R. Kollmann and H.-J. Kreowski, *An integrated semantics for UML class, object and state diagrams based on graph transformation*, in: M. Butler, L. Petre and K. Sere, editors, *IFM 2002*, LNCS **2335** (2002), pp. 11–28.

[9] Object Management Group, "UML 2.0 Superstructure Specification," (2004), (updated version). http://www.omg.org/cgi-bin/doc?ptc/2004-10-02.

[10] Reggio, G., E. Astesiano, C. Choppy and H. Hussmann, *Analysing UML active classes and associated state machines – a lightweight formal approach*, in: T. Maibaum, editor, *FASE 2000*, LNCS **1783** (2000), pp. 127–146.

A Proof Sketch of Proposition 4.2

The outline of the proof is as follows. First, we define a formula and prove that it is an invariant under every transition step. Thereafter, the invariant is used to conclude Proposition 4.2.

Before we present the invariant, we define function $\Delta : (\mathcal{O} \rightharpoonup_F \mathcal{O}^\star) \times \mathcal{O} \rightharpoonup (\mathcal{O} \rightharpoonup_F \mathcal{O}^\star) \times \mathcal{O}$ by $\Delta(\mathcal{M}, o) \stackrel{\text{def}}{=} \begin{cases} (\mathcal{M}[o \mapsto \overrightarrow{o}], o'), & \text{if } \mathcal{M}(o) = o' \cdot \overrightarrow{o} \text{ and} \\ \text{undefined}, & \text{otherwise .} \end{cases}$

Define the invariant $\Lambda \subseteq \mathcal{G}_{\mathcal{O}'}$ by

$$(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \in \Lambda \iff$$
$$\text{dom}(\boldsymbol{\sigma}) \supseteq \{\pi_2(\Delta^i(\mathcal{M}, o)) \mid i \in \mathbb{N}\} \supseteq \{o' \in \text{dom}(\boldsymbol{\sigma}) \mid \neg\text{stable}(\boldsymbol{\sigma}(o'))\}.$$

It is easily seen that every stable activity group satisfies Λ. We make the following observation: If $(\boldsymbol{\sigma}, \boldsymbol{E}, \mathcal{M}, o) \in \Lambda$ and $\mathcal{M}(o) = \epsilon$ then all its objects have to be stable since $\{\pi_2(\Delta^i(\boldsymbol{\sigma}, \mathcal{M}, o)) \mid i \in \mathbb{N}\} \supseteq \{o' \in \text{dom}(\boldsymbol{\sigma}) \mid \neg\text{stable}(\boldsymbol{\sigma}(o'))\}$ 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 o' getC $(\boldsymbol{\sigma}, \mathcal{M}, o)$ can only be used in the rules different from executing signals and operations if $o = o'$ holds. With these observations in mind it is straightforwardly checked that Λ is an invariant for all $\hookrightarrow$ rules. Proposition 4.2 can be straightforwardly concluded by using this invariant and by making a case analysis according to the transition labels considered.$\square$