Mobile UML Statecharts with Localities*

Diego Latella[1], Mieke Massink[1], Hubert Baumeister[2], and Martin Wirsing[2]

[1] CNR, Istituto di Scienza e Tecnologie dell'Informazione,
Via Moruzzi 1, I56124 Pisa, Italy
{Diego.Latella, Mieke.Massink}@isti.cnr.it
[2] LMU, Institut für Informatik, Oettingenstr. 67, D-80538 München, Germany
{Hubert.Baumeister, Martin.Wirsing}@ifi.lmu.de

Abstract. In this paper an extension of a behavioural subset of UML statecharts for mobile computations is proposed. We study collections of UML objects whose behaviour is given by statecharts. Each object resides in a given place, and a collection of such places forms a network. Objects are aware of the *localities* of other objects, i.e. the logical names of the places where the latter reside, but not of the physical name of such places. In addition to their usual capabilities, such as sending messages etc., objects can move between places and create and destroy places, which may result in a deep reconfiguration of the network. A formal semantics is presented for this mobility extension which builds upon a core semantics definition of statecharts without mobility which we have used successfully in several contexts in the past years. An example of a model of a network service which exploits mobility for resource usage balance is provided using the proposed extension of UML statecharts.

1 Introduction

Mobility plays a major role in the programming of nowadays network-based services. The Unified Modelling Language (UML) is the de facto standard graphical modelling language for object-oriented software and systems [17]. It has been specifically designed for visualising, specifying, constructing, and documenting several aspects of—or views on—systems. In this paper we focus on a behavioural subset of UML statecharts (UMLSCs) and in particular on a powerful extension of this notation in order to deal with a notion of mobility which is sometimes referred to as *mobile computation* and requires computing elements to be able to migrate from one node to another within a network [4], as opposed to *mobile computing*, where the focus is instead on dynamic communication structures. We address mobile computing in the context of UMLSCs in a companion paper [12].

In this paper we assume that a system is modelled as a *dynamic* collection of (cooperating, autonomous) *objects*. In order to express mobile computations we assume that each object is located at exactly one network node, or *place* as

* This work has been carried out in the context of Project EU-IST IST-2001-32747 Architectures for Mobility (AGILE).

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 34–58, 2004.

we shall call it in the sequel. A *network* is a collection of places. Note that for simplicity a network has a flat hierarchy similar to the approach used in KLAIM [5], that is, places cannot contain places.

The attributes of an object may contain values of basic datatypes, like integer, boolean e.t.c., references to other objects, and references to network nodes. The behaviour of an object is given by a UMLSC. Objects can move among places and objects and places can be created or destroyed.

Objects are not aware of the physical names of places; they make reference only to logical names, also called *localities*, which play the same role as symbolic addresses in the Internet. Consequently, each place is also equipped with an *allocation environment* which maps localities to the physical names of places. The architecture of the network is dynamic and is implicitly defined by the information encapsulated in the allocation environments of all the places belonging to the network; such information collectively defines the set of places each place is "in touch with". The choice of objects being unaware of place physical names has been inspired by the work on KLAIM [5].

Example. Consider a simplified model of a resource based compute server. Depending on the resources needed for a computation, the computation should be performed at different network places or being split among different network places. Figure 1 shows the class diagram using the stereotype ≪mobile≫ introduced in [2] to indicate classes whose objects can move between places.

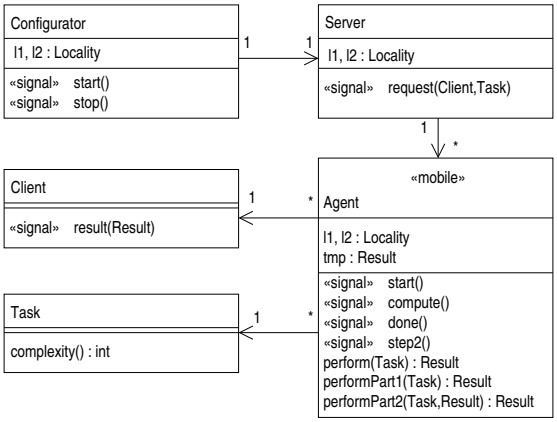

Fig. 1. Class diagram for a simple compute server

The behaviour of the configurator, server, and agents is given by the statecharts in Figs. 2, 3, and 4. The configurator first creates two network places where the computation should take place (cf. Fig. 2). During the creation of the network places also two new localities are created that refer to these places and which are stored in attributes l_1 and l_2. The configurator then exports the place where the configurator is located (given by variable atLoc) to the newly

created places, i.e. the locality home at the new places refers to the place of the configurator. Next, the server is created and initialised with the two localities l_1 and l_2.

When the server receives a request by a client to perform a task, an agent is created to fulfil the request (cf. Fig. 3). The agent moves to the different network places depending on the complexity of the computation task (cf. Fig. 4). In case of a simple task, the agent stays at the place of the server, computes the result of the task, and sends the result back to client. In case of a more complex task, the agent first moves to the place referred to by l_1, performs the computation there, and then sends the result back to the client who stayed at the place referred to by locality home. In the most complex case, the agent first performs part of the computation at the place referred to by locality l_1 and then the second part of the computation at locality l_2 before sending the result to the client at locality home. After the agent has done its work he destroys itself.

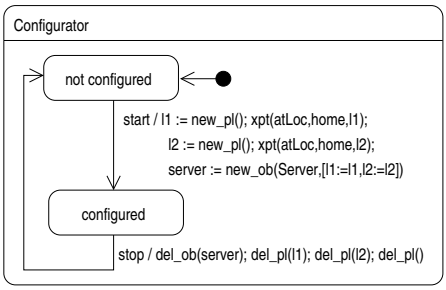

Fig. 2. Statechart for class Configurator of Fig. 1

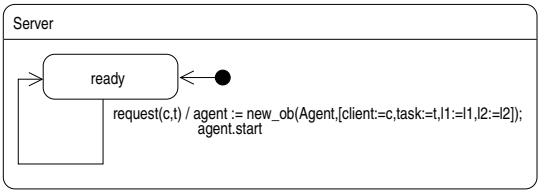

Fig. 3. Statechart for class Server of Fig. 1

In the present paper, we propose a formal operational semantics for the mobility extension of UMLSCs briefly discussed above. The operational semantics is built upon previous work of Latella et al. on formalising the semantics of UMLSCs [11, 6], which in turn was inspired by the work of Mikk [16] on Harel statecharts.

Several other proposals for formal semantics of UMLSCs can be found in the literature, e.g. [19, 3, 14, 13, 18]. None of these approaches deals with mobility; we refer to [6] for a comparison with our previous work. An approach similar to ours is [10] in which ambients [4] are added to Interacting State Machines

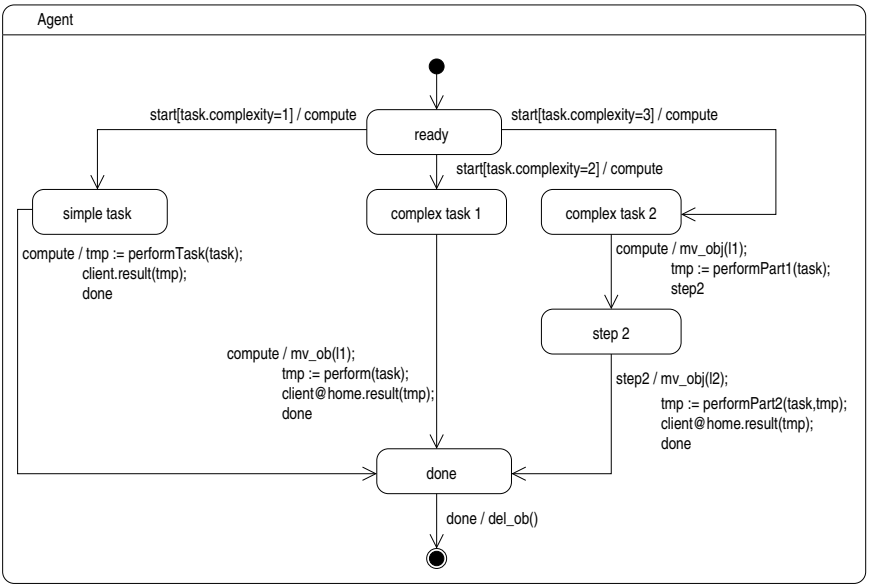

Fig. 4. Statechart for class Agent of Fig. 1

(ISMs). The differences are that this approach is not intended to model UMLSCs and therefore does not deal directly with features of UMLSCs, like composite states and concurrent substates, and that ISMs have a hierarchical structure of places while our place structure is flat. Another approach is [8] where UML state machines with mobility are translated to MTLA formulas [15] to study refinement of state machines.

Several proposals for extending the UML with mobility and/or agent notions are present in the literature, e.g. [2, 9, 7, 1]. In [2] a proposal for extending UML activity diagrams with mobility notions is presented while in [9] mobility in UML sequence diagrams is addressed. In [7] an extension of UML in order to describe agent interaction is proposed which does not address mobility explicitly. Mobile agents in UML are the focus of [1]. All the above mentioned proposals deal with notational extensions of UML performed mainly by means of UML extension mechanisms (stereotypes, tagged-values, etc.) and do not address issues of formal semantics. We are not aware of any proposal for a formal semantics of objects, object management, mobility and network configuration in the context of UML statecharts.

The paper is organised as follows: Section 2 briefly addresses the general framework of our approach, including our modelling assumptions and an informal introduction of the basic notions of hierarchical automata for UMLSCs. Section 3 describes the kind of actions which can label UMLSCs transitions and the notion of network specification. The formal semantics is given in Sect. 4 and conclusions are drawn in Sect. 5. Finally, for the interested reader, the appendix contains the formal semantics of hierarchical automata.

2 Basic Framework

In this section we set the basic framework of our work. In particular we present a brief description of the modelling assumptions on which we base our approach and an informal introduction to the basic notions of the computational framework of hierarchical automata, which we use as an abstract syntax for the definition of UMLSCs semantics. All technical details on hierarchical automata for UMLSCs can be found in [11, 6] and in the appendix.

2.1 Object Model

In this paper we assume that a system is modelled as a *dynamic* collection of (cooperating, autonomous) *objects* and that the behaviour of each object is specified by a UMLSC (more than one single object may have its behaviour specified by the same UMLSC). More precisely, we assume that a networked system is specified by a *static* collection $\{SC_1, \ldots, SC_c\}$ of UMLSCs. In order to express mobile computations, we explicitly model network nodes, or *places* as we shall call them in the sequel, and we define a *network* as a collection of places. The set of places of a network may dynamically change during the evolution of the network.

Each place has a set of objects residing therein and is uniquely identified within the network by its *physical name*. A physical name can be thought of as an IP address in the context of the Internet. The objects residing within a place are uniquely identified by their *object names* within that place. The set of objects residing in a given place may dynamically change during the evolution of the network.

Objects cannot directly refer to the places' physical names; instead, they use logical names, called *localities*, which are mapped to physical names by the *allocation environment* of the place where the object resides. In addition, each object may have a private allocation environment. The network structure is not given explicitly, but implicitly by the information encapsulated in the allocation environments, which defines the set of places a place is "in touch with".

The above assumptions are quite realistic and rather common in networked systems, in particular unawareness of physical names of places. In this choice we have been inspired by the work on KLAIM [5] of which our model also shares the flat structure of places. Other proposals, like for instance [4], assume a hierarchical structure. We choose a flat structure for the sake of simplicity.

2.2 Hierarchical Automata

As briefly mentioned in Sect. 2.1, in our approach, a networked system is specified by a finite collection of UMLSCs $\{SC_1, \ldots, SC_c\}$. We use hierarchical automata (HAs) [16] as an abstract syntax for UMLSCs. HAs for UMLSCs have been introduced in previous work of co-authors of the present paper ([11, 6]). In this section we recall, informally, only the main notions which are necessary for the understanding of the paper. The reader interested in the detailed formal definitions concerning UMLSCs, like the definition of the behavioural semantics, is

referred to the appendix. Such definition is essentially an orthogonal extension of the original formalisation of UMLSCs semantics of [11, 6], where mobility was *not* addressed.

Informally, a HA is composed of a collection of simple sequential automata related by a *refinement function* which imposes on the HA the hierarchical state nesting-structure of the associated statechart. Inter-level transitions are encoded by means of proper annotations in transition labels.

The operational semantics of HAs, which we shall define later on in the present paper, makes use of what we call the *Core Semantics* of HAs. The Core Semantics of a HA H characterises the relation $H :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_L \mathcal{C}'$ with the following intended meaning: whenever the current *state configuration* of H is $\mathcal{C}$, its current variables/values binding is *store* β, and *event* ev is fed to H's *state-machine*, the *transitions* in L may fire bringing H to configuration $\mathcal{C}'$; Ac is the sequence of *actions* to be executed—actually an interleaving of the action sequences labelling the transitions in L—and ξ records the binding of the parameters occurring in the *triggers* of such transitions with the corresponding values in ev.

Thus the role of the Core Semantics is the characterisation of the set of transitions to be fired, their related actions, and the resulting configuration. All issues of (action) ordering, concurrency, and non-determinism within single statecharts are dealt with by the Core Semantics. Although essential for the definition of the formal semantics, all the above issues are technically quite orthogonal to mobility and dynamic network/object management.

Another issue which deserves to be briefly addressed here is the way in which we deal with the so called *input-queue* of UMLSCs, i.e. their "external environment". In the standard definition of UML statecharts semantics [17], a *scheduler* is in charge of selecting an event from the input-queue of an object, feeding it into the associated state-machine, and letting such a machine produce a STEP transition. Such a STEP transition corresponds to the firing of a maximal set of enabled non-conflicting transitions of the statechart associated to the object, provided that certain transition priority constraints are not violated. After such transitions are fired and when the execution of all the actions labelling them is completed, the STEP itself is completed and the scheduler can choose another event from a queue and start the next cycle. While in classical statecharts the external environment is modelled by a set, in the definition of UML statecharts, the nature of the input-queue of a statechart is not specified; in particular, the management policy of such a queue is not defined. In our overall approach to UMLSCs semantics definition, we choose *not* to fix any particular semantics, such as set, or multi-set or FIFO-queue etc., but to model the input queue in a policy-independent way, freely using a notion of abstract data types. In the following we assume that for set D, Θ_D denotes the class of all structures of a certain kind (like FIFO queues, or multi-sets, or sets) over D and we assume to have basic operations for manipulating such structures. In particular, in the present paper, we let $\mathsf{Add}\ d\ \mathcal{D}$ denote the structure obtained inserting element d in structure $\mathcal{D}$ and the predicate $(\mathsf{Sel}\ \mathcal{D}\ d\ \mathcal{D}')$ state that $\mathcal{D}'$ is the structure

resulting from selecting d from $\mathcal{D}$; of course, the selection policy depends on the choice for the particular semantics. We assume that if $\mathcal{D}$ is the empty structure, denoted by $\langle\rangle$, then $(\mathsf{Sel}\ \mathcal{D}\ d\ \mathcal{D}')$ yields FALSE for all d and $\mathcal{D}'$. We shall often speak of the *input queue*, or simply *queue*, by that meaning a structure in Θ_D, abstracting from the particular choice for the semantics of Θ_D.

We shall refer to the set $\{H_1, \ldots, H_c\}$ of HAs associated to $SC_1, \ldots, SC_c$. Conf_H will denote the set of all state configurations of HA H and we shall assume that for every set $\{H_1, \ldots, H_c\}$ of HAs, there exists a distinguished element $\mathcal{C}_{\mathsf{err}}$ such that $\mathcal{C}_{\mathsf{err}} \notin \bigcup_{j=1}^{c} \mathrm{Conf}_{H_j}$.

3 Action Language and Network Specifications

In this section we introduce the syntax and informal semantics of HAs transition *actions* and *triggers*. Moreover, we formalise the notion of *network specification*.

3.1 Actions

The action side of transition t, i.e. $AC\ t$, is an *action*, and its abstract syntax is shown in Fig. 5. In our extension we will deal with place physical names, place logical names, object names, method names, and with variables[1]; moreover, parameters may occur in method activations. Consequently proper countable, mutually disjoint sets—$\mathcal{Z}_P, \mathcal{Z}_L, \mathcal{Z}_O, \mathcal{Z}_M, \mathsf{Var}$, and Par, respectively—are introduced for them. Moreover we assume that object names form a $\mathcal{Z}_P$-indexed family of disjoint sets.

$$
\begin{array}{rl}
Ac ::= & var := exp \mid obj@loc_1.meth(exp) \mid \mathsf{mv\_ob}(obj@loc_1, loc_2) \mid \mathsf{mv\_cl}(obj@loc_1, loc_2) \\
\mid & var := \mathsf{new\_ob}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)@loc_1 \mid var := \mathsf{new\_cl}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)@loc_1 \\
\mid & \mathsf{del\_ob}(obj@loc_1) \mid var := \mathsf{new\_pl}() \mid \mathsf{del\_pl}(loc) \mid \mathsf{xpt}(loc_1, loc_2, loc_3) \mid Ac; Ac
\end{array}
$$

where $var \in \mathsf{Var}$, $exp \in \mathsf{Var} \cup \mathsf{Par} \cup \mathcal{Z}_L$, $obj \in \mathsf{Var} \cup \mathsf{Par}$, $loc, loc_i \in \mathsf{Var} \cup \mathsf{Par} \cup \mathcal{Z}_L$ for $i \in \{1, 2, 3\}$, $meth \in \mathcal{Z}_M$, and Var *(variable identifiers)* including self and atLoc, Par *(parameter identifiers)*, $\mathcal{Z}_L$ *(place logical names—i.e. localities)* including here, and $\mathcal{Z}_M$ *(method names)* are countable, mutually disjoint sets.

Fig. 5. Abstract syntax of actions

In the following we informally describe the meaning of the various actions, together with simple static semantics constraints. The formal definition of the action semantics will be addressed later on in the paper while we refrain from giving a formal definition of the static semantics since the static semantics is not very relevant for the purpose of the present paper. Consequently we assume that all variables used in a statechart are declared in the associated object definition and that all actions are type-correct.

[1] We will use "variable" and "attribute" as synonym in this paper.

By $var := exp$, variable var is assigned the value of exp in the current store of the object where the action is executed. Only attribute names of the current object are allowed as variables; i.e. variables can not refer to attributes of different objects. Of course different objects (or even HAs) can use the same variable name which will be bound to possibly different values by different local stores. Reserved, read-only, variables self and atLoc are always bound to the name of the object in the store of that object, and, similarly, the reserved locality here is always bound to the physical place name in the allocation environment of that place.

Action $obj@loc_1.meth(exp)$ sends an *asynchronous message meth* with (optional parameter-)value exp to object obj residing in (the place referred to by) locality loc_1; the following short-hands are provided: $obj.meth(exp)$ is used instead of $obj@\mathsf{atLoc}.meth(exp)$ and $meth(exp)$ instead of $\mathsf{self}.meth(exp)$.

Action $\mathsf{mv\_ob}(obj@loc_1, loc_2)$ makes object obj migrate from locality loc_1 to locality loc_2—notice that the moved object can also be the object executing the action, in which case the short-hand $\mathsf{mv\_ob}(loc_2)$ can be used.

Action $var := \mathsf{new\_ob}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)@loc_1$ creates a new object the behaviour of which is determined by the HA referred to by H_0. The name of the newly created object will be bound to var in the store of the object executing the action, i.e. the *creator*. The initial configuration $\mathcal{C}_0$ must belong to Conf_{H_0}; for the specification of the initial store β_0 we use the notation $[var_1 := exp_1, \ldots, var_n := exp_n]$, where $exp_1, \ldots exp_n \in \mathsf{Var} \cup \mathsf{Par} \cup \mathcal{Z}_L$, and $var_1, \ldots, var_n \in \mathsf{Var} \setminus \{\mathsf{self}, \mathsf{atLoc}\}$ are attribute names of the created object.[2] The initial configuration, store, and input-queue $\mathcal{C}_0, \beta_0, \mathcal{E}_0$ are optional. If absent, the initial configuration indicated in the definition of the HA referred to by H_0 is used for $\mathcal{C}_0$ while $\mathcal{E}_0$ is empty and β_0 binds only self to the name of the created object and atLoc to the locality where it resides. The newly created object will (initially) reside in locality loc_1. If loc is atLoc or here, the short-hand $var := \mathsf{new\_ob}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)$ can be used.

The references of an object to localities are normally resolved via the allocation environment of the place where the object resides when the action is executed which uses such references. Moreover, a closure-like version of object creation (new_cl) and moving (mv_cl) actions is provided; the allocation environment of the place where the *creator* resides will be inherited by the created object as its private allocation environment; similarly, the allocation environment of the place where the moved object was residing will be inherited by the moved object and extended with its private allocation environment, if any.

By $\mathsf{del\_ob}(obj@loc_1)$ an object kills object obj residing in locality loc_1—notice that the object destroyed can also be the object executing the action, in which case the shorthand $\mathsf{del\_ob}()$ can be used.

[2] Notice that in the specification of β_0 the creator object will access attributes of the created object (i.e. $var_1, \ldots, var_n$); this is the only exception to the local variables rule mentioned above.

An object can create a new place by means of $var := \mathsf{new\_pl}()$. Variable identifier var will be bound—in the store of the current object—to the new locality, which in turn will be bound—in the current allocation environment—to the physical name of the newly created object.

Similarly place loc_1 is destroyed by executing $\mathsf{del\_pl}(loc_1)$; the place will be removed from the network and all the information it contains (objects residing therein as well as the allocation environment) is lost. The short-hand $\mathsf{del\_pl}()$ destroys the place where the executing object resides.

By $\mathsf{xpt}(loc_1, loc_2, loc_3)$ locality loc_2 can be exported to the place referred to by locality loc_3 and get bound, in its allocation environment, to the place physical name which loc_1 is bound to in the allocation environment of the residence place of the executing object.

Finally, the sequential composition of action(s) Ac_1 with action(s) Ac_2 is denoted by $Ac_1; Ac_2$

3.2 Triggers

The *trigger EV t* of transition t must be a method name ($meth \in \mathcal{Z}_M$) or a method name with one parameter ($meth(x)$ with $x \in \mathsf{Par}$). The trigger has the usual pattern-matching semantics; the parameter is bound to the input value when the transition is selected for being fired. It is worth pointing out here that the UML requires the scope of a parameter to be confined to the *single* transition where it occurs as part of the trigger. The restriction to just one parameter is made only for the sake of notational simplicity.

3.3 Transition Labels and Network Specifications

The concrete syntax for the complete label of a transition t, at the UMLSC level, will be $EV\ t\ [G\ t]/AC\ t$, where the guard $[G\ t]$ is optional. The treatment of all optional parts of actions as well as short-hands is assumed to be dealt with at the static analysis level. We can now formally define *network specifications*:

Definition 1 (Network Specification). *A network specification is composed of a set* $\{H_1, \ldots, H_c\}$ *of HAs and an initialisation command* ($\mathsf{INIT}\ Ac$) *where* Ac *is a (possibly compound) action, as specified in Fig 5.*

An example of a network specification is given by the statecharts for Configurator, Server, and Agent of Figs. 2, 3, and 4 from the example in the introduction. The initialisation actions in this example are:

$$\mathsf{INIT}(l := \mathsf{new\_pl}();\ c := \mathsf{new\_ob}(\mathsf{Configurator})@l;\ c.start).$$

4 Network Semantics

The operational semantics associates a transition system to a network specification. The states of such a transition system correspond to distinct states of the

network. In this section we define the semantics formally. We start with the formal definition of *stores* and *allocation environments*. Then we define *Network-* (resp. *Place-*, *Object-*) *States* and the transition relation. The definition of the latter makes use of a function for the semantic interpretation of the actions labeling statechart transitions; the remainder of the present section is devoted to the formal definition of such a function.[3]

Definition 2 (Stores). *A* store β *is a function* $\beta : \mathsf{Var} \cup \mathsf{Par} \to \mathcal{Z}_L \cup \mathcal{Z}_O \cup \{\mathsf{unbound}\}$*, where* $\mathsf{unbound} \notin \mathcal{Z}_L \cup \mathcal{Z}_O$ *is a distinguished value.*

As usual $\beta\ x = \mathsf{unbound}$ means that x is not bound by β to any value. The *empty store, unit store* and *store extension* operators ($[\,]$, $[x \mapsto n]$, and $\triangleleft$ respectively) are defined in the usual way:

$$
\begin{array}{lll}
[\,]\ x & \triangleq & \mathsf{unbound}, \\
 & & \textbf{for all}\ \ x \in \mathsf{Var} \cup \mathsf{Par} \\
[x \mapsto n]\ x' & \triangleq & \textbf{if}\ \ x = x' \textbf{then}\ \ n\ \textbf{else}\ \mathsf{unbound}, \\
 & & \textbf{for all}\ \ x, x' \in \mathsf{Var} \cup \mathsf{Par}, n \in \mathcal{Z}_L \cup \mathcal{Z}_O \\
(\beta_1 \triangleleft \beta_2)\ x & \triangleq & \textbf{if}\ \ \beta_2\ x \neq \mathsf{unbound}\ \textbf{then}\ \beta_2\ x\ \textbf{else}\ \beta_1\ x, \\
 & & \textbf{for all}\ \ \textit{stores}\ \beta_1, \beta_2, x \in \mathsf{Var} \cup \mathsf{Par}
\end{array}
$$

Allocation environments map localities to place physical names:

Definition 3 (Allocation Environments). *An* allocation environment γ *is a function* $\gamma : \mathcal{Z}_L \cup \{\mathsf{unbound}\} \to \mathcal{Z}_P \cup \{\mathsf{unbound}\}$*. We require that* $\gamma\ \mathsf{unbound} = \mathsf{unbound}$ *for every allocation environment* γ*.*

As for stores, we will let $[\,]$, respectively $[l \mapsto p]$, denote the empty, respectively unit, allocation environment, and $\gamma_1 \triangleleft \gamma_2$ denote the extension of γ_1 with γ_2, with a little bit of overloading in the notation. For each object, reserved read-only variables $\mathsf{self}, \mathsf{atLoc} \in \mathsf{Var}$ will be bound respectively to the name of the object and to the distinguished element $\mathsf{here} \in \mathcal{Z}_L$, in its current store. Similarly, for each place, here will be bound to the place physical name in its current allocation environment. Finally $\mathsf{hereafter} \in \mathcal{Z}_P$ is a distinguished place name conventionally used in the definition of the semantics of object/place destruction.

The operational semantics defines how a network may evolve as a consequence of firing transitions of the statecharts associated to the objects residing in the

[3] In the following we shall freely use a functional-like notation in our definitions where currying will be often used in function application, i.e. $f\ a_1\ a_2 \ldots\ a_n$ will be used instead of $f(a_1, a_2, \ldots, a_n)$ and function application will be considered left-associative; for function $f : X \to Y$ and $Z \subseteq X$, $f\ Z \triangleq \{y \in Y \mid \exists x \in Z.\ y = fx\}$, $dom\ f$ and $rng\ f$ denote the *domain* and *range* of f and $f_{|Z}$ is the restriction of f to Z; in particular, $f \setminus z$ stands for $f_{|(dom\ f) \setminus \{z\}}$; for distinct $x_1, \ldots, x_n$, $f[y_1/x_1, \ldots, y_n/x_n]$ is the function which on x_j yields y_j and on any other $x' \notin \{x_1, \ldots, x_n\}$ yields $f\ x'$.

places of the network itself. We remind here that, in our approach, the primitive computational elements are the objects and that their behaviour is specified by statecharts. The evolution of objects is modelled by their internal state together with their "physical" position in the network. More specifically, at each stage of the global computation, each object resides in a specific place of the network and its internal state is composed by its current *configuration*—drawn from those of the statechart specifying its behaviour, its current local *store*—where the current values of its attributes (variables) are maintained, the current value of its *input queue*, and the current *private allocation environment*.

The evolution of the network can thus be modelled by means of a transition system where each state corresponds to a network state and each transition corresponds to a change of network state operated by firing the transitions of a STEP of the statechart of an object. In order to make the above notions more precise we need the definition below, where $\mathcal{E} \in \Theta_{\mathcal{Z}_M \cup (\mathcal{Z}_M \times (\mathcal{Z}_L \cup \mathcal{Z}_O))}$, i.e. the elements of input queues are method invocations, with possibly one parameter. For $(m, n) \in \mathcal{Z}_M \times (\mathcal{Z}_L \cup \mathcal{Z}_O)$, we use the more common, "constructor-like", syntax $m(n)$.

Definition 4 (Network, Place and Object States). *A* network state N *is a finite set of place states. A* place state P *of a network state* N *is a triple* $(p, \gamma, \mu) \in N$ *where* $p \in \mathcal{Z}_P \setminus \{\mathsf{hereafter}\}$ *is the physical name for* P*, and is required to be unique net-wide;* γ *is the allocation environment of* P *and* μ *is the finite set of object states of* P*. An object state* O *of place state* $P = (p, \gamma, \mu)$ *of network state* N *models the state of an object and is a 6-tuple* $(o, H, \lambda, \mathcal{C}, \beta, \mathcal{E})$ *where* $o \in \mathcal{Z}_O^p$ *is the name of the object and is required to be unique network-wide;* H *is the reference to the HA which specifies the behaviour of* o *and* λ *(respectively* $\mathcal{C}, \beta, \mathcal{E}$*) is the current private allocation environment (respectively configuration, store, input queue) of* o*.*

It is worth pointing out here that as an obvious consequence of place name uniqueness within a network state N the latter can be used and manipulated as a (finite domain) function on $\mathcal{Z}_P$ such that $p \in dom\ N$ and $N\ p = (\gamma, \mu)$ if, and only if $(p, \gamma, \mu) \in N$. Similarly, also μ can be used and manipulated as a (finite domain) function on $\mathcal{Z}_O^p$ with $o \in dom\ \mu$ and $\mu\ o = (H, \lambda, \mathcal{C}, \beta, \mathcal{E})$ if, and only if $(o, H, \lambda, \mathcal{C}, \beta, \mathcal{E}) \in \mu$. In the sequel, we assume $\{\mathsf{unbound}, \mathsf{hereafter}\} \cap (dom\ N) = \emptyset$ for all network states N and $\mathsf{unbound} \notin (dom\ \mu)$ for all place states (p, γ, μ).

The operational semantics associates a transition system $(S, \longrightarrow, S_0)$ to a network specification. S is the set of states of such transition system, which are network states, defined as above. A distinguished network state corresponds to the initial state S_0. Conventionally, such a state consists of the single place $(\mathsf{init\_pl}, [\,], \{(\mathsf{init\_o}, \mathsf{INITHA}, [\,], \{1\}, [\,], \langle \mathsf{init\_ev} \rangle)\})$ where INITHA is a conventional HA composed of a single state—1— which is source and target of a single transition labelled by $\mathsf{init\_ev}/as$ where as is the argument of the INIT initialisation command.[4] The transition relation $\longrightarrow$ is defined by means of a logical deduc-

[4] It is assumed that $\mathsf{init\_o}$ and $\mathsf{init\_ev}$ do not occur in any network specification.

tion system, and the definition is given in two stages: the *Top Rule* and the *Core Semantics*. The Top Rule is shown in Fig. 6 and in turn uses the Core Semantics, which has been introduced in Sect. 2. The Top Rule stipulates that in order for the network to evolve from network state N to state N' there must exist an object o (2nd premise) in a place p of the current network state (1st premise), the statechart of which—H—can perform a STEP from the (non-error) configuration (3rd premise) $\mathcal{C}$ to $\mathcal{C}'$ (5th premise) when event ev is selected from its input queue (5th premise). The STEP transition generated by the Core Semantics is labelled by the set L of the HA transitions which are fired in the STEP, the pair (ev, β), where β is the current store of o, and the pair (Ac, ξ) where Ac is a sequence of actions—the actions of the transitions in L—and ξ is the set of bindings of the parameters occurring in the triggers of such transitions to the input event ev. Ac is a *symbolic* representation of transition actions; thus we need to interpret them. This is achieved by means of the interpretation function $\mathcal{N}$ which actually computes the new network state N' (6th premise). N' is computed by applying $\mathcal{N}[\![\mathsf{Ac}]\!]$ to a network state which is the same as N except that the new configuration $\mathcal{C}'$ and the remaining input queue $\mathcal{E}'$ are recorded for object o. Notice that the store of o is not updated (yet) since the new store will be (part of) the result of the execution of Ac.

$$\frac{\begin{array}{l}(p, \gamma, \mu) \in N \\ (o, H, \lambda, \mathcal{C}, \beta, \mathcal{E}) \in \mu \\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}} \\ \mathsf{Sel}\ \mathcal{E}\ \ ev\ \mathcal{E}' \\ H :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_L \mathcal{C}' \\ N' = \mathcal{N}[\![\mathsf{Ac}]\!]\ (N[(\gamma, \mu[(H, \lambda, \mathcal{C}', \beta, \mathcal{E}']/o)/p], p)\ o\ \xi\end{array}}{N \longrightarrow N'}$$

Fig. 6. Transition relation definition (top rule)

In the following we define the action interpretation function $\mathcal{N}$. Actually $\mathcal{N}$ simply returns the first element of the pair resulting from the application of function $\mathcal{I}$ to the same arguments.

$$\mathcal{N}[\![Ac]\!]\ (N, p)\ o\ \xi \ \stackrel{\Delta}{=}\ N' \text{ where } (N', p') = \mathcal{I}[\![Ac]\!]\ (N, p)\ o\ \xi$$

The definition of $\mathcal{I}$ uses the following auxiliary functions.

$$\begin{array}{ll}\mathsf{ERR}\ N\ p\ o & \stackrel{\Delta}{=}\ N[(\gamma, \mu[(nil, [], \mathcal{C}_{\mathsf{err}}, [], \langle\rangle)/o])/p], \\ & \textbf{for all}\ \ net.\ states\ N, p \in dom\ N, o \in \mathcal{Z}_O\end{array}$$

$\mathsf{ERR}\ N\ p\ o$ is the network state which differs from N only because the *erratic* object state $(o, nil, [], \mathcal{C}_{\mathsf{err}}, [], \langle\rangle)$ is present in place state p.

Function $\mathcal{V}$ is defined in the usual way.

$$\mathcal{V}[\![exp]\!]\ \beta \quad \stackrel{\Delta}{=} \quad \begin{array}{l} \textbf{if}\ \ exp \in \mathsf{Var} \cup \mathsf{Par} \\ \textbf{then}\ \ \beta\ exp \\ \textbf{else}\ \ exp, \\ \textbf{for all}\ \ stores\ \beta, \\ \qquad\qquad exp \in \mathsf{Var} \cup \mathsf{Par} \cup \mathcal{Z}_L \end{array}$$

$\mathsf{new}\ \mathcal{Z}_P \quad \stackrel{\Delta}{=}$ *a* fresh new *place physical name* $p \in \mathcal{Z}_P \setminus \{\mathsf{init\_pl}, \mathsf{hereafter}\}$ *different from any place physical name already generated.*

$\mathsf{new}\ \mathcal{Z}_L \quad \stackrel{\Delta}{=}$ *a* fresh new *locality name* $l \in \mathcal{Z}_L \setminus \{\mathsf{here}\}$ *different from any locality textually occurring in the network specification or already generated.*

$\mathsf{new}\ \mathcal{Z}_O^p \quad \stackrel{\Delta}{=}$ *a* fresh new *object name* $o \in \mathcal{Z}_O^p \setminus \{\mathsf{init\_o}\}$ *textually different from any object name occurring in the network specification or already generated.*

For action Ac, $\mathcal{I}[\![\mathsf{Ac}]\!]$ is a function which takes a pair (N, p)—where N is a network state and $p \in \mathcal{Z}_P$—an object name $o \in \mathcal{Z}_O$ and a parameter binding ξ. $\mathcal{I}[\![\mathsf{Ac}]\!]\ (N, p)\ o\ \xi$ is a pair (N', p') where N' is the network state resulting from the execution of the actions Ac and $p' \in \mathcal{Z}_P$. $\mathcal{I}[\![\mathsf{Ac}]\!]$ is defined by induction on the structure of Ac. In all cases it is first of all required that $p \in dom\ N$, i.e. $\exists \gamma, \mu.\ (\gamma, \mu) = N\ p$ and that o is not erratic, with $o \in dom\ \mu$.

In the following, we list all cases of the definition, together with a short informal explanation, when necessary.

$$\begin{array}{l} \mathcal{I}[\![var := exp]\!]\ (N, p)\ o\ \xi \quad \stackrel{\Delta}{=} \\ \quad \textbf{if}\ \gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, n \\ \quad \textbf{exist such that} \\ \quad (\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}, \\ \quad n = \mathcal{V}[\![exp]\!]\ (\beta \triangleleft \xi),\ n \neq \mathsf{unbound} \\ \quad \textbf{then}\ N[(\gamma, \mu[(H, \lambda, \mathcal{C}, \beta[n/var], \mathcal{E})/o])/p], p) \\ \quad \textbf{else}\ (\mathsf{ERR}\ N\ p\ o, p) \end{array}$$

Access to an uninitialised variable exp (i.e. $\mathcal{V}[\![exp]\!]\ (\beta \triangleleft \xi) = \mathsf{unbound}$) in an assignment action $var := exp$ brings o to the erratic state, otherwise var is bound in the store to the value n of exp. Notice that expressions are evaluated—using function $\mathcal{V}$—in the current store *temporarily extended* with the parameter bindings.

$$\begin{array}{l} \mathcal{I}[\![obj@loc.meth(exp)]\!]\ (N, p)\ o\ \xi \quad \stackrel{\Delta}{=} \\ \quad \textbf{if}\ \gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l, o', p', \gamma', \mu', H', \lambda', \mathcal{C}', \beta', \mathcal{E}', m, n \\ \quad \textbf{exist such that} \\ \quad (\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}, \end{array}$$

$o' = \mathcal{V}[\![obj]\!]\ (\beta \triangleleft \xi),\ l = \mathcal{V}[\![loc]\!]\ (\beta \triangleleft \xi),\ p' = (\gamma \triangleleft \lambda)\ l,$
$(\gamma', \mu') = N\ p',\ (H', \lambda', \mathcal{C}', \beta', \mathcal{E}') = \mu'\ o',$
$n = \mathcal{V}[\![exp]\!]\ (\beta \triangleleft \xi)$
then $(N[(\gamma', \mu'[(H', \lambda', \mathcal{C}', \beta', \mathsf{Add}\ meth(n)\ \mathcal{E}')/o'])/p'], p)$
else $(\mathsf{ERR}\ N\ p\ o, p)$

In the execution of sending an asynchronous call $meth(exp)$ to object $obj@loc$, performed by object o residing in place p of N, it is required that locality $\mathcal{V}[\![loc]\!]\ (\beta \triangleleft \xi) = l$ is bound, in the local allocation environment γ (possibly extended with the private allocation environment of o if any) to the physical name p' of a place where an object named $\mathcal{V}[\![obj]\!]\ (\beta \triangleleft \xi) = o'$ exists. In this case, $meth(\mathcal{V}[\![exp]\!]\ (\beta \triangleleft \xi))$ is added in the input queue of o'; otherwise o ends up in the erratic state.

$\mathcal{I}[\![\mathsf{mv\_ob}(obj@loc_1, loc_2)]\!]\ (N, p)\ o\ \xi \ \stackrel{\Delta}{=}$
if $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l_1, l_2, p_1, p_2, \gamma_1, \gamma_2, \mu_1, \mu_2, o', p'$
exist such that
$(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$
$l_1 = \mathcal{V}[\![loc_1]\!]\ (\beta \triangleleft \xi),\ p_1 = (\gamma \triangleleft \lambda)\ l_1,\ (\gamma_1, \mu_1) = N\ p_1,$
$l_2 = \mathcal{V}[\![loc_2]\!]\ (\beta \triangleleft \xi),\ p_2 = (\gamma \triangleleft \lambda)\ l_2,\ (\gamma_2, \mu_2) = N\ p_2,$
$o' = \mathcal{V}[\![obj]\!]\ (\beta \triangleleft \xi),\ o' \in (dom\ \mu_1),$
$p' =$ **if** $p = p_1\ , o = o'$ **then** p_2 **else** p
then $(N[(\gamma_1, \mu_1 \setminus o')/p_1, (\gamma_2, \mu_2[\mu_1\ o'/o'])/p_2], p')$
else $(\mathsf{ERR}\ N\ p\ o, p)$

$\mathcal{I}[\![\mathsf{mv\_cl}(obj@loc_1, loc_2)]\!]\ (N, p)\ o\ \xi \ \stackrel{\Delta}{=}$
if $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l_1, l_2, p_1, p_2, \gamma_1, \gamma_2, \mu_1, \mu_2, \mu_2' o', p',$
$H', \lambda', \mathcal{C}', \beta', \mathcal{E}'$
exist such that
$(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$
$l_1 = \mathcal{V}[\![loc_1]\!]\ (\beta \triangleleft \xi),\ p_1 = (\gamma \triangleleft \lambda)\ l_1,\ (\gamma_1, \mu_1) = N\ p_1,$
$l_2 = \mathcal{V}[\![loc_2]\!]\ (\beta \triangleleft \xi),\ p_2 = (\gamma \triangleleft \lambda)\ l_2,\ (\gamma_2, \mu_2) = N\ p_2,$
$o' = \mathcal{V}[\![obj]\!](\beta \triangleleft \xi),\ o' \in (dom\ \mu_1),$
$(H', \lambda', \mathcal{C}', \beta', \mathcal{E}') = \mu'\ o',$
$\mu_2' = \mu_2[(H', \gamma_1 \triangleleft \lambda', \mathcal{C}', \beta', \mathcal{E}')/o']$
then $(N[(\gamma_1, \mu_1 \setminus o')/p_1, (\gamma_2, \mu_2')/p_2], p')$
else $(\mathsf{ERR}\ N\ p\ o, p)$

The successful execution of $\mathsf{mv\_ob}(obj@loc_1, loc_2)$ requires that the object denoted by obj resides in the place p_1 referred to by loc_1 *and* that there is no object with the same name in the place p_2 referred to by loc_2. If this is the case, this object is removed from the residence place p_1 and added to the set of objects of p_2, i.e. is moved from p_1 to p_2. In the case of $\mathsf{mv\_cl}$, furthermore the current allocation environment γ_1 of p_1 is added to the (possibly empty) private allocation environment of the object. Notice that actions $\mathsf{mv\_ob}$ and $\mathsf{mv\_cl}$ can be applied also by o to itself, when obj evaluates to o and loc_1 refers to p. In this

case the residence place of o changes to the value of loc_2. We keep track of the residence of o as it results from the execution of Ac in the second element of the result of $\mathcal{I}[\![\mathsf{Ac}]\!]\ (N,p)\ o\ \xi$. The reader is invited to check how this information is dealt with in the definition of $\mathcal{I}[\![Ac_1; Ac_2]\!]$.

$$
\begin{array}{l}
\mathcal{I}[\![var := \mathsf{new\_ob}(H_0,\mathcal{C}_0,\beta_0,\mathcal{E}_0)@loc]\!]\ (N,p)\ o\ \xi \ \stackrel{\Delta}{=} \\
\quad \textbf{if } \gamma,\mu,H,\lambda,\mathcal{C},\beta,\mathcal{E},l,o',p',\gamma',\mu' O,O' \\
\quad \textbf{exist such that} \\
\quad (\gamma,\mu) = N\ p,\ (H,\lambda,\mathcal{C},\beta,\mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}} \\
\quad l = \mathcal{V}[\![loc]\!]\ (\beta \triangleleft \xi),\ p' = (\gamma \triangleleft \lambda)\ l,\ (\gamma',\mu') = N\ p', \\
\quad o' = \mathsf{new}\ \mathcal{Z}_O^{p'}, \\
\quad O = (H,\lambda,\mathcal{C},\beta[o'/var],\mathcal{E}), \\
\quad O' = (H_0,[],\mathcal{C}_0,\beta_0[o'/\mathsf{self},\mathsf{here}/\mathsf{atLoc}],\mathcal{E}_0) \\
\quad \textbf{then if } p = p' \\
\qquad\quad \textbf{then } (N[(\gamma,\mu[O/o,O'/o'])/p],p) \\
\qquad\quad \textbf{else } (N[(\gamma,\mu[O/o])/p,(\gamma',\mu[O'/o'])/p'],p) \\
\quad \textbf{else } (\mathsf{ERR}\ N\ p\ o,p)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I}[\![var := \mathsf{new\_cl}(H_0,\mathcal{C}_0,\beta_0,\mathcal{E}_0)@loc]\!]\ (N,p)\ o\ \xi \ \stackrel{\Delta}{=} \\
\quad \textbf{if } \gamma,\mu,H,\lambda,\mathcal{C},\beta,\mathcal{E},l,o',p',\gamma',\mu',O,O' \\
\quad \textbf{exist such that} \\
\quad (\gamma,\mu) = N\ p,\ (H,\lambda,\mathcal{C},\beta,\mathcal{E}) = \mu\ o,\ \ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}, \\
\quad l = \mathcal{V}[\![loc]\!]\ (\beta \triangleleft \xi),\ p' = (\gamma \triangleleft \lambda)\ l,\ (\gamma',\mu') = N\ p', \\
\quad o' = \mathsf{new}\ \mathcal{Z}_O^{p'}, \\
\quad O = (H,\lambda,\mathcal{C},\beta[o'/var],\mathcal{E}), \\
\quad O' = (H_0,\gamma,\mathcal{C}_0,\beta_0[o'/\mathsf{self},\mathsf{here}/\mathsf{atLoc}],\mathcal{E}_0) \\
\quad \textbf{then if } p = p' \\
\qquad\quad \textbf{then } (N[(\gamma,\mu[O/o,O'/o'])/p],p) \\
\qquad\quad \textbf{else } (N[(\gamma,\mu[O/o])/p,(\gamma',\mu[O'/o'])/p'],p) \\
\quad \textbf{else } (\mathsf{ERR}\ N\ p\ o,p)
\end{array}
$$

The creation $\mathsf{new\_ob}$ of a new object in an *existing* place referred to by loc requires the modification of the store of the creator object o in order to bind variable var to the name o' of the newly created object. Moreover the new object is placed in the place referred to by loc. Notice that the initial store of the new object binds atLoc to here and self to o'. In the case of $\mathsf{new\_cl}$, the private allocation environment of the newly created object is initialised to the allocation environment of the place where the creator resides.

$$
\begin{array}{l}
\mathcal{I}[\![\mathsf{del\_ob}(obj@loc)]\!]\ (N,p)\ o\ \xi \ \stackrel{\Delta}{=} \\
\quad \textbf{if } \gamma,\mu,H,\lambda,\mathcal{C},\beta,\mathcal{E},l_1,p_1,\gamma_1,\mu_1,o',p' \\
\quad \textbf{exist such that} \\
\quad (\gamma,\mu) = N\ p,\ (H,\lambda,\mathcal{C},\beta,\mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}, \\
\quad l_1 = \mathcal{V}[\![loc]\!]\ (\beta \triangleleft \xi),\ p_1 = (\gamma \triangleleft \lambda)\ l_1,\ (\gamma_1,\mu_1) = N\ p_1, \\
\quad o' = \mathcal{V}[\![obj]\!]\ (\beta \triangleleft \xi), \\
\quad p' = \textbf{if}\ \ p = p_1,\ o = o'\ \textbf{then}\ \ \mathsf{hereafter}\ \textbf{else}\ \ p
\end{array}
$$

$\textbf{then}\ (N[(\gamma, \mu[(H, \lambda, \mathcal{C}, \beta \setminus obj, \mathcal{E})/o])/p, (\gamma_1, \mu_1 \setminus o')/p_1], p')$
$\textbf{else}\ (\mathsf{ERR}\ N\ p\ o, p)$

An existing object *obj* residing in a place referred to by *loc* is destroyed by executing $\mathsf{del\_ob}(obj@loc)$. The result will be that the object will be removed from the set of objects of *loc* and variable *obj* will be unbound after $\mathsf{del\_ob}$ will have been executed. Notice that if $obj@loc$ is exactly the object which is executing the action then its residence place changes to $\mathsf{hereafter}$ (we remind the reader that $\mathsf{hereafter} \in (dom\ N)$ for *no* network state N). Notice moreover that the semantics is undefined if action $\mathsf{del\_ob}$ occurs in Ac but *not* as its *last* element.

$\mathcal{I}[\![var := \mathsf{new\_pl}()]\!]\ (N, p)\ o\ \xi \ \stackrel{\Delta}{=}$
$\textbf{if}\ \gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, O, l, p'$
$\textbf{exist such that}$
$(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$
$l = \mathsf{new}\ \mathcal{Z}_L,\ p' = \mathsf{new}\ \mathcal{Z}_P,$
$O = (H, \lambda, \mathcal{C}, \beta[l/var], \mathcal{E})$
$\textbf{then}\ (N[(\gamma[p'/l], \mu[O/o])/p, ([\mathsf{here} \mapsto p'], \emptyset)/p'], p)$
$\textbf{else}\ (\mathsf{ERR}\ N\ p\ o, p)$

The successful execution of $var := \mathsf{new\_pl}()$ creates a new place where here is bound to its physical name by its (otherwise empty) allocation environment. Moreover, a new locality is bound to such physical name in the allocation environment of the place where the executing object resides and such locality is bound to variable *var* in the store of the object.

$\mathcal{I}[\![\mathsf{del\_pl}(loc)]\!]\ (N, p)\ o\ \xi \ \stackrel{\Delta}{=}$
$\textbf{if}\ \gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l, p', p''$
$\textbf{exist such that}$
$(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}$
$l = \mathcal{V}[\![loc]\!]\ (\beta \triangleleft \xi),\ p'' = (\gamma \triangleleft \lambda),$
$p' = \textbf{if}\ \ p = p''\ \textbf{then}\ \ \mathsf{hereafter}\ \textbf{else}\ \ p$
$\textbf{then}\ ((N[(\gamma \setminus l, \mu[(H, \lambda, \mathcal{C}, \beta \setminus loc, \mathcal{E})/o])/p]) \setminus p'', p')$
$\textbf{else}\ (\mathsf{ERR}\ N\ p\ o, p)$

The interpretation of place destruction ($\mathsf{del\_pl}$) should be clear to the reader; the only exception is when the object executing it destroys the place where it resides. Notice that the semantics is undefined if action $\mathsf{del\_pl}$ occurs in Ac but *not* as its *last* element.

$\mathcal{I}[\![\mathsf{xpt}(loc_1, loc_2, loc_3]\!]\ (N, p)\ o\ \xi \ \stackrel{\Delta}{=}$
$\textbf{if}\ \gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l_1, l_2, l_3, p_1, p_3, \gamma_3, \mu_3$
$\textbf{exist such that}$
$(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}$
$l_1 = \mathcal{V}[\![loc_1]\!]\ (\beta \triangleleft \xi),\ p_1 = (\gamma \triangleleft \lambda)\ l_1,$
$l_2 = \mathcal{V}[\![loc_2]\!]\ (\beta \triangleleft \xi),$

$l_3 = \mathcal{V}[\![loc_3]\!]\ (\beta \triangleleft \xi),\ p_3 = (\gamma \triangleleft \lambda)\ l_3,$
$(\gamma_3, \mu_3) = N\ p_3,\ l_2 \notin dom\ \gamma_3$
then $(N[(\gamma_3[p_1/l_2], \mu_3)/p_3], p)$
else $(\mathsf{ERR}\ N\ p\ o, p)$

$\mathcal{I}[\![Ac_1; Ac_2]\!]\ (N, p)\ o\ \xi \ \stackrel{\Delta}{=}$
$\mathcal{I}[\![Ac_2]\!]\ (\mathcal{I}[\![Ac_1]\!]\ (N, p)\ o\ \xi)\ o\ \xi$

The semantics of xpt and sequentialization is self-explanatory.

5 Conclusions

In this paper, UML statecharts have been extended with a notion of mobility. In particular *mobile computation*, where computational units migrate from one node to another within a network has been considered, as opposed to *mobile computing*, which addresses dynamic communication structures [12].

A formal operational semantics for the extended notation has been provided which covers all major aspects of UML statecharts—like state hierarchy, inter-level transitions, a parametric treatment of transition priority and input queue, intra- and inter- statechart concurrency, and run-to-completion. Furthermore, it includes dynamic object management, i.e. object creation and object destruction, for objects (the behaviour of which is) specified by statecharts; finally notions specific to dynamic network management are addressed: network places, network architecture management, and mobility (i.e. object migration).

An example of a model of a network service which exploits mobility for resource usage balance has been provided using our mobile extension of UMLSCs.

We are not aware of any proposal in the literature which combines all the above mentioned issues in a single formal framework which is moreover completely compatible and consistent with other "views" and extensions of UML statecharts, like testing theories, stochastic behaviour modelling, and analysis and LTL/BTL model-checking.[5]

The space of network places is *flat*, which is similar to that of KLAIM [5]. Other proposals, like for instance [4, 10], assume a hierarchical structure for the place space. We chose a flat approach mainly for the sake of simplicity. A hierarchical place structure would open the way toward mobility of *places* and, since mobility is part of computation and objects are the computational units, this would bring to the blurring of the conceptual difference between objects and places. In addition, the conceptual difference between physical names and localities would need to be revisited. We leave all the above issues for further study.

The impact of our extension on the UML meta model—e.g. what additional meta classes are needed and whether these can be introduced by stereotypes or not—is also left for future study.

[5] Papers describing the above mentioned issues can be found at: http://fmt.isti.cnr.it.

Another issue for further study is the integration of the mobile computing approach proposed in [12] with the mobile computation one proposed in the present paper, as well as the interplay between mobile computing and mobile computation in a framework where also place mobility is considered, as briefly mentioned above. Finally, we are interested in developing useful theories for the extension we proposed in the present paper, like, e.g. access control and security, in a similar way as in [5].

References

1. B. Bauer, J. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Computer Science*, pages 91–103. Springer-Verlag, 2001.
2. H. Baumeister, N. Koch, P. Kosiuczenko, and M. Wirsing. Extending Activity Diagrams to Model Mobile Systems. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World.*, volume 2591 of *Lecture Notes in Computer Science*, pages 278–293. Springer-Verlag, 2002.
3. J. Broersen and R. Wieringa. Interpreting UML-statecharts in a modal μ-calculus. Unpublished manuscript, 1997.
4. L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *FoSSaCS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–145. Springer-Verlag, 1998.
5. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–329, 1998.
6. S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML Statechart Diagrams kernel and its extension to Multicharts and Branching Time Model Checking. *The Journal of Logic and Algebraic Programming. Elsevier Science*, 51(1):43–75, 2002.
7. C. Klein, A. Rausch, M. Sihling, and Z. Wen. Extension of the Unified Modeling Language for mobile agents. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 8. Idea Group Publishing, Hershey, PA and London, 2001.
8. Alexander Knapp, Stephan Merz, and Martin Wirsing. On refinement of mobile UML state machines, 2004. to appear in Proc. AMAST 2004.
9. P. Kosiuczenko. Sequence Diagrams for Mobility. In J. Krogstie, editor, *MobIMod 2002*, volume XXXX of *Lecture Notes in Computer Science*. Springer-Verlag, 2003. (To appear).
10. A. Kuhn and von Oheimb D. Interacting state machines for mobility. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 698–718. Springer-Verlag, 2003.
11. D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, pages 331–347. Kluwer Academic Publishers, 1999. ISBN 0-7923-8429-6.

12. D. Latella and M. Massink. On mobility extensions of UML Statecharts; a pragmatic approach. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 199–213. Springer-Verlag, 2003.
13. J. Lilius and I. Paltor Porres. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard.*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445. Springer-Verlag, 1999.
14. J. Lilius and I. Paltor Porres. The semantics of UML state machines. Technical Report 273, Turku Centre for Computer Science, 1999.
15. S. Merz, M. Wirsing, and J. Zappe. A Spatio-Temporal Logic for the Specification and Refinement of Mobile Systems. In M. Pezzé, editor, *Fundamental Approaches to Software Engineering (FASE 2003)*, volume 2621 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
16. E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *Third Asian Computing Science Conference. Advances in Computing Sience - ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196. Springer-Verlag, 1997.
17. Object Management Group, Inc. *OMG Unified Modeling Language Specification - version 1.5*, 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.
18. M. von der Beeck. A structured operational semantics for UML-statecharts. *Software Systems Modeling. Springer*, (1):130–141, 2002.
19. R. Wieringa and J. Broersen. A minimal transition system semantics for lightweight class and behavior diagrams. In M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors, *Proceedings of the ICSE98 Workshop on Precise Semantics for Software Modeling techniques*, 1998.

A Hierarchical Automata

In this section we informally recall some basic notions related to UMLSCs. They are treated in depth in [11, 6]. We use hierarchical automata (HAs) [16] as the abstract syntax for UMLSCs. HAs are composed of simple sequential automata related by a *refinement function*. In [11] an algorithm for mapping a UMLSC to a HA is given. Here we just recall the main ingredients of this mapping, by means of a simple example. Consider the UMLSC of Fig. 7 (). Its HA is shown top on the bottom of the figure. Roughly speaking, each OR-state of the UMLSC is mapped into a sequential automaton of the HA while basic and AND-states are mapped into states of the sequential automaton corresponding to the OR-state immediately containing them. Moreover, a refinement function maps each state in the HA corresponding to an AND-state into the set of the sequential automata corresponding to its component OR-states. In our example (Fig. 7, bottom), OR-states $s0, s4, s5$ and $s7$ are mapped to sequential automata $A0, A1, A2$ and $A3$, while state $s1$ of $A0$, corresponding to AND-state $s1$ of our UMLSC, is refined into $\{A1, A2\}$. Non-interlevel transitions are represented in the obvious way: for instance transition $t8$ of the HA represents the transition from state $s8$ to state $s9$ of the UMLSC. The labels of transitions are collected in Table 1; for example the *trigger* of $t8$, namely $EV\ t8$, is $e2$ while its associated *action*, namely $AC\ t8$

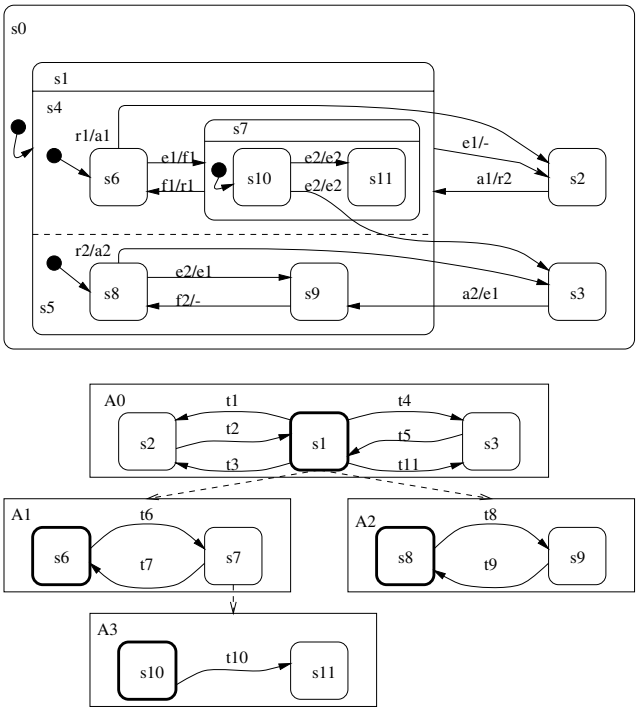

Fig. 7. A UMLSC and its HA

Table 1. Transition labels for the HA of Fig. 7

t	$SR\ t$	$EV\ t$	$AC\ t$	$TD\ t$
$t1$	$\{s6\}$	$r1$	$a1$	$\emptyset$
$t2$	$\emptyset$	$a1$	$r2$	$\{s6, s8\}$
$t3$	$\emptyset$	$e1$	ϵ	$\emptyset$
$t4$	$\{s8\}$	$r2$	$a2$	$\emptyset$
$t5$	$\emptyset$	$a2$	$e1$	$\{s6, s9\}$
$t6$	$\emptyset$	$e1$	$f1$	$\{s10\}$
$t7$	$\emptyset$	$f1$	$r1$	$\emptyset$
$t8$	$\emptyset$	$e2$	$e1$	$\emptyset$
$t9$	$\emptyset$	$f2$	ϵ	$\emptyset$
$t10$	$\emptyset$	$e2$	$e2$	$\emptyset$
$t11$	$\{s10\}$	$e2$	$e2$	$\emptyset$

consists in $e1$. Label $e2$ can model the activation of a method of an object the behaviour of which is modeled by the statechart and, respectively, $e1$ can be the invocation of a method, which takes place if and when $t8$ is fired.

An interlevel transition is represented as a transition t departing from (the HA state corresponding to) its highest source and pointing to (the HA state corresponding to) its highest target. The set of the other sources, resp., targets,

are recorded in the *source restriction*—$SR\ t$, resp. *target determinator* $TD\ t$, of t. So, for instance, $SR\ t1 = \{s6\}$ means that a necessary condition for $t1$ to be enabled is that the current state configuration contains not only $s1$ (the source of $t1$), but *also* $s6$. Similarly, when firing $t2$ the new state configuration will contain $s6$ and $s8$, besides $s1$. Finally, each transition has a guard $G\ t$, not shown in this example.

Transitions originating from the same state are said to be in *conflict*. The notion of *conflict* between transitions is to be extended in order to deal with state hierarchy and a priority notion between conflicting transition is defined. When transitions t and t' are in conflict we write $t\#t'$. Intuitively transitions coming from deeper states have higher priority. For the purposes of the present paper it is sufficient to say that priorities form a partial order. We let πt denote the priority of transition t and $\pi t \sqsubseteq \pi t'$ mean that t has lower priority than (the same priority as) t'.

In the sequel we will be concerned only with HAs. In particular, for a given network specification, we shall make reference to the set $\{H_1, \ldots, H_c\}$ of the HAs associated to the UMLSCs $SC_1, \ldots, SC_c$ used in the specification.

A.1 Basic Definitions

The first notion we need to define is that of (sequential) automaton:[6]

Definition 5 (Sequential Automata). *A sequential automaton A is a 4-tuple $(\sigma_A, s^0_A, \lambda_A, \delta_A)$ where σ_A is a finite set of* states *with $s^0_A \in \sigma_A$ the* initial state

[6] In the following we shall freely use a functional-like notation in our definitions where: (i) currying will often be used in function application, i.e. $f\ a_1\ a_2 \ldots\ a_n$ will be used instead of $f(a_1, a_2, \ldots, a_n)$ and function application will be considered left-associative; (ii) for function $f : X \to Y$ and $Z \subseteq X$, $f\ Z \stackrel{\Delta}{=} \{y \in Y \mid \exists x \in Z.\ y = fx\}$, $dom\ f$ and $rng\ f$ denote the *domain* and *range* of f and $f_{|Z}$ is the restriction of f to Z; in particular, $f \backslash z$ stands for $f_{|(dom\ f) \backslash \{z\}}$; for distinct $x_1, \ldots, x_n$, $f[y_1/x_1, \ldots, y_n/x_n]$ is the function which on x_j yields y_j and on any other $x' \notin \{x_1, \ldots, x_n\}$ yields $f\ x'$; for functions f and g such that for all $x \in (dom\ f \cap dom\ g)$ $f\ x = g\ x$ holds we will often let $f \cup g$ denote the function which yields $f\ x$ if $x \in dom\ f$ and $g\ x$ if $x \in dom\ g$ and we extend the notation to n functions in the obvious way. By $\exists_1 x.\ P\ x$ we mean "there exists a unique x such that $P\ x$". Finally, for set D, we let D^* denote the set of finite sequences over D. The empty sequence will be denoted by ϵ and, for $d \in D$, with a bit of overloading, we will often use d also for the unit sequence containing only d; the concatenation of sequence x with sequence y will be indicated by xy. For sequences x, y and z we let predicate $\mathsf{mrg}\ x\ y\ z$ hold if, and only if z is a non-deterministic merge (or interleaving) of x and y, that is z is a permutation of xy such that the occurrence order in x (respectively y) of the elements of x (respectively y) is preserved in z; a possible definition for mrg is $\mathsf{mrg}\ x\ y\ z \stackrel{\Delta}{=} \exists w \in (D \times \{1,2\})^*.\ pr1\ w = z \wedge pr1\ (only\ 1\ w) = x \wedge pr1\ (only\ 2\ w) = y$ where $pr1\ \epsilon \stackrel{\Delta}{=} \epsilon$, $pr1\ (e,j)l \stackrel{\Delta}{=} e(pr1\ l)$, $only\ j\ \epsilon \stackrel{\Delta}{=} \epsilon$, and $only\ j\ (e,j')l \stackrel{\Delta}{=} (\mathbf{if}\ \ j = j'\ \mathbf{then}\ \ (e,j)\ \mathbf{else}\ \epsilon)(only\ j\ l)$; the extension of mrg to n sequences, $\mathsf{mrg}^n_{j=1}\ x_j\ z$, is defined in the obvious way.

λ_A *is a finite set of* transition labels *and* $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ *is the* transition relation.

We assume that all transitions are uniquely identifiable. This can be easily achieved by just assigning them arbitrary unique names, as we shall do throughout this paper. For sequential automaton A let functions $SRC, TGT : \delta_A \to \sigma_A$ be defined as $SRC(s, l, s') = s$ and $TGT(s, l, s') = s'$.

HAs are defined as follows:

Definition 6 (Hierarchical Automata). *A HA H is a 3-tuple (F, E, ρ) where F is a finite set of sequential automata with mutually disjoint sets of states, i.e. $\forall A_1, A_2 \in F.\ \sigma_{A_1} \cap \sigma_{A_2} = \emptyset$ and E is a finite set of transition labels; the* refinement function *$\rho : \bigcup_{A \in F} \sigma_A \to 2^F$ imposes a tree structure to F, i.e. (i) there exists a unique root automaton $A_{root} \in F$ such that $A_{root} \notin \bigcup rng\ \rho$, (ii) every non-root automaton has exactly one ancestor state: $\bigcup rng\ \rho = F \setminus \{A_{root}\}$ and $\forall A \in F \setminus \{A_{root}\}.\ \exists_1 s \in \bigcup_{A' \in F \setminus \{A\}} \sigma_{A'}.\ A \in (\rho\ s)$ and (iii) there are no cycles: $\forall S \subseteq \bigcup_{A \in F} \sigma_A.\ \exists s \in S.\ S \cap \bigcup_{A \in \rho s} \sigma_A = \emptyset$.*

We say that a state s for which $\rho\ s = \emptyset$ holds is a *basic* state. Every sequential automaton $A \in F$ characterises a HA in its turn: intuitively, such a HA is composed by all those sequential automata which lay below A, including A itself, and has a refinement function ρ_A which is a restriction of ρ:

Definition 7. *For $A \in F$ the* automata *and* states under *A are defined respectively as*
$\mathcal{A}\ A \stackrel{\Delta}{=} \{A\} \cup (\bigcup_{A' \in \left(\bigcup_{s \in \sigma_A} (\rho_A s)\right)} (\mathcal{A}\ A'))$, $\mathcal{S}\ A \stackrel{\Delta}{=} \bigcup_{A' \in \mathcal{A}\ A} \sigma_{A'}$

The definition of sub-hierarchical automaton follows:

Definition 8 (Sub-hierarchical Automata). *For $A \in F$, (F_A, E, ρ_A), where $F_A \stackrel{\Delta}{=} (\mathcal{A}\ A)$, and $\rho_A \stackrel{\Delta}{=} \rho_{|(\mathcal{S}\ A)}$, is the HA characterised by A.*

In the sequel for $A \in F$ we shall refer to A both as a sequential automaton and as the sub-hierarchical automaton of H it characterises, the role being clear from the context. H will be identified with A_{root}. Sequential automata will be considered a degenerate case of HAs. A central role in UMLSCs is played by *(state) configurations*, defined as follows:

Definition 9 (Configurations). *A* configuration *of HA $H = (F, E, \rho)$ is a set $\mathcal{C} \subseteq (\mathcal{S}\ H)$ such that (i) $\exists_1 s \in \sigma_{A_{root}}.\ s \in \mathcal{C}$ and (ii) $\forall s, A.\ s \in \mathcal{C} \wedge A \in \rho\ s \Rightarrow \exists_1 s' \in \sigma_A.\ s' \in \mathcal{C}$*

A *configuration* is a global state of a HA, composed of local states of component sequential automata. For $A \in F$ the set of all configurations of A is denoted by Conf_A. Moreover we will assume that for every set $\{H_1, \ldots, H_c\}$ of HAs, there exists a distinguished element $\mathcal{C}_{\mathsf{err}}$ such that $\mathcal{C}_{\mathsf{err}} \notin \bigcup_{j=1}^{c} \mathrm{Conf}_{H_j}$

Progress rule

$$\frac{\begin{array}{l} t \in \mathsf{LE}_A\ \mathcal{C}\ \beta\ ev \\ \nexists t' \in T \cup \mathsf{E}_A\ \mathcal{C}\ \beta\ ev.\ \pi t \sqsubset \pi t' \end{array}}{A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(AC\ t,\mathsf{bnd}\ ev\ (EV\ t))}_{\{t\}} DST\ t}$$

Stuttering Rule

$$\frac{\begin{array}{l} \{s\} = \mathcal{C} \cap \sigma_A \\ \rho_A\ s = \emptyset \\ \forall t \in \mathsf{LE}_A\ \mathcal{C}\ \beta\ ev.\ \exists t' \in T.\ \pi t \sqsubset \pi t' \end{array}}{A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\epsilon,[\,])}_{\emptyset} \{s\}}$$

Composition Rule

$$\frac{\begin{array}{l} \{s\} = \mathcal{C} \cap \sigma_A \\ \rho_A\ s = \{A_1, \ldots, A_n\} \neq \emptyset \\ \left(\bigwedge_{j=1}^{n} A_j \uparrow T \cup \mathsf{LE}_A\ \mathcal{C}\ \beta\ ev :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac}_j,\xi_j)}_{L_j} \mathcal{C}_j\right) \\ \mathsf{mrg}_{j=1}^{n} \mathsf{Ac}_j\ \mathsf{Ac} \wedge \xi = \bigcup_{j=1}^{n} \xi_j \wedge L = \bigcup_{j=1}^{n} L_j \\ L = \emptyset \Rightarrow (\forall t \in \mathsf{LE}_A\ \mathcal{C}\ \beta\ ev.\ \exists t' \in T.\ \pi t \sqsubset \pi t') \end{array}}{A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_{L} \{s\} \cup \bigcup_{j=1}^{n} \mathcal{C}_j}$$

Fig. 8. Rules of the Core Semantics

A.2 Core Semantics Definition

The Core Semantics definition is given in Fig. 8

As mentioned before, the Core Semantics definition is very similar to the one we have used in previous work of ours. Here we give a very brief description with emphasis on those aspects relevant for the purposes of the present paper and we refer the reader interested in more details to [11, 6]. Intuitively, $A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_{L} \mathcal{C}'$ models labelled transitions of the HA A, and L is the set containing the transitions of the sequential automata of A which are selected to fire. We call $\xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_{L}$ the *STEP*-transition relation in order to avoid confusion with transitions of sequential automata. When confusion may arise, we call the latter *sequential* transitions. T is a set of sequential transitions. It represents a constraint on each of the transitions fired in the step, namely that it must not be the case that there is a transition in T with a higher priority. So, informally, $A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_{L} \mathcal{C}'$ should be read as (an object the behaviour of which is specified by HA) "A, on configuration $\mathcal{C}$, provided with input event (i.e. method call) ev and the store of which is β can perform L moving to configuration $\mathcal{C}'$, when required to perform transitions with priorities not smaller than any in T; Ac is the sequence of actions to be executed as result of firing the transitions in L and ξ binds the value carried by ev, if any, to proper parameters occurring in the triggers of transitions in L". Set T will be used to record the transitions a certain automaton can do when considering its sub-automata. More specifically, for sequential automaton A, T will accumulate all transitions which are enabled in the ancestors of A. The Core Semantics definition makes use of the auxiliary functions defined in Fig. 9. $\mathsf{LE}_A\ \mathcal{C}\ \beta\ ev$

is the set of all the *enabled local* transitions of A in $\mathcal{C}, \beta$, with ev[7]. Similarly, the set of all *enabled* transitions of A—considered as an HA, i.e. including the transitions of descendants of A—in $\mathcal{C}, \beta$, with ev, is $\mathsf{E}_A\ \mathcal{C}\ \beta\ ev$.

$$
\begin{array}{l}
\mathsf{LE}_A\ \mathcal{C}\ \beta\ ev \ \stackrel{\Delta}{=} \\
\quad \{t \in \delta_A \mid \{(SRC\ t)\} \cup (SR\ t) \subseteq \mathcal{C}, \\
\qquad\qquad \mathsf{match}\ ev\ (EV\ t), \\
\qquad\qquad (\mathcal{C}, \beta \triangleleft (\mathsf{bnd}\ ev\ (EV\ t)), ev) \models (G\ t)\} \\
\quad \textbf{for all} \quad HAs\ H = (F, E, \rho),\ A \in F,\ \mathcal{C} \in \mathrm{Conf}_H, \\
\qquad\qquad stores\ \beta,\ input\ events\ ev
\end{array}
$$

$$
\begin{array}{l}
\mathsf{E}_A\ \mathcal{C}\ \beta\ ev \ \stackrel{\Delta}{=} \\
\quad \bigcup_{A' \in (\mathcal{A}\ A)} \mathsf{LE}_{A'}\ \mathcal{C}\ \beta\ ev
\end{array}
$$

where:

$$
\begin{array}{ll}
\mathsf{bnd}\ m\ m' & \stackrel{\Delta}{=} \\
\quad [\,] & \\
\mathsf{bnd}\ m(n)\ m'(x) & \stackrel{\Delta}{=} \\
\quad \textbf{if}\ \mathsf{match}\ m(n)\ m'(x)\ \textbf{then}\ [x \mapsto n]\ \textbf{else}\ [\,] & \\
\quad \textbf{for all}\ \ n \in \mathcal{Z}_L \cup \mathcal{Z}_O, m, m' \in \mathcal{Z}_M, x \in \mathsf{Par} &
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{match}\ m\ m' & \stackrel{\Delta}{=} \\
\quad (m = m') & \\
\mathsf{match}\ m(n)\ m'(x) & \stackrel{\Delta}{=} \\
\quad (m = m'), \mathrm{Type}[\![n]\!]_H = \mathrm{Type}[\![x]\!]_H & \\
\quad \textbf{for all}\ \ n \in \mathcal{Z}_L \cup \mathcal{Z}_O, m, m' \in \mathcal{Z}_M, x \in \mathsf{Par} &
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{Type}[\![exp]\!]_H & \stackrel{\Delta}{=}\ \ the\ type\ of\ expression\ exp\ in\ the\ context \\
& \quad of\ (the\ variables/constants\ declaration \\
& \quad of\ the\ class\ associated\ to)\ HA\ H.
\end{array}
$$

Fig. 9. Auxiliary functions for the Core Semantics

In the Core Semantics, the Progress Rule establishes that if there is a transition t of A enabled by event ev in the current configuration $\mathcal{C}$ and store β and the priority of such a transition is "high enough" then the transition fires and a new configuration is reached accordingly. The action to be (eventually) executed is $AC\ t$ and the parameter binding is generated in the obvious way by means of function bnd. The Composition Rule stipulates how automaton A delegates the execution of transitions to its sub-automata (3rd premise) and these transitions are propagated upward. Notice that for all v, i, j, $\xi_i\ v \neq \mathsf{unbound}$ and

[7] $(\mathcal{C}, \beta, ev) \models g$ means that guard g is true for configuration $\mathcal{C}$, store β and input event ev. Its formalisation is immaterial for the purposes of the present paper. The definition of $\mathrm{Type}[\![e]\!]_H$ is part of UML static semantics and here we assume it given.

$\xi_j\ v \neq \mathsf{unbound}$ implies $\xi_i\ v = \xi_j\ v = ev$. Moreover, different orderings of actions due to different interleavings of the firing of the transitions in L are captured by means of predicate mrg (4th premise). Finally, if there is no transition of A enabled with "high enough" priority and moreover no sub-automata exist to which the execution of transitions can be delegated, then A has to "stutter", as enforced by the Stuttering Rule. Notice that stuttering of sub-automata is propagated upwards by the Composition Rule *only* if no local transition can be fired either (last premise of Composition Rule). In the operational semantics definition of Fig. 6, the simplified notation $H :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_L \mathcal{C}'$ has been used which stands for $H \uparrow \emptyset :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_L \mathcal{C}'$.

The following theorem links our semantics to the general requirements set by the official semantics of UML:

Theorem 1. *Given HA* $H = (F, E, \rho)$ *for all* $A \in F, ev \in E, T, L, \mathcal{C}, \beta, \mathsf{Ac}$ *the following holds:* $A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_L \mathcal{C}'$ *for some* $\mathcal{C}', \xi$ *iff* L *is a maximal set, under set inclusion, which satisfies all the following properties: (i)* L *is conflict-free, i.e.* $\forall t, t' \in L.\ \neg t \# t'$*; (ii) all transitions in* L *are enabled, i.e.* $L \subseteq \mathsf{E}_A\ \mathcal{C}\ \beta\ ev$ *; (iii) there is no transition outside* L *which is enabled and which has higher priority than a transition in* L*, i.e.* $\forall t \in L.\ \nexists t' \in \mathsf{E}_A\ \mathcal{C}\ \beta\ ev.\ \pi t \sqsubset \pi t'$*; and (iv) all transitions in* L *respect* T*, i.e.* $\forall t \in L.\ \nexists t' \in T.\ \pi t \sqsubset \pi t'$.

Proof. The proof can be carried out in a similar way as for the main theorem of [6], by structural induction for the direct implication and by derivation induction for the reverse implication. □