System Model for UML
The Interactions Case

María Victoria Cengarle

Institut für Informatik, Technische Universität München
85748, Garching bei München, Boltzmannstr. 3, Germany
cengarle@in.tum.de

Abstract. Previous works define the notion of system model, which provides a formal basis for the definition of the semantics of a distributed object-oriented modelling language. This article addresses the UML 2.0 interactions and their meaning in terms of a system model. One denotational and two operational approaches are discussed. These are based on existing proposals for the semantics of interactions, but in principle alternative proposals can also be employed.

Keywords. UML, interactions, denotational semantics, operational semantics, system model

1 Introduction

The *Methods for Modelling Software Systems (MMOSS)* seminar which took place in Dagstuhl by the end of August'06 addressed, among others, the issue of "right" level of abstraction for the task of modelling complex software systems. Abstraction is indeed an indispensable tool for managing complexity. In many cases, however,

> formal proofs of the adequacy of such abstractions are infeasible because they somehow rely on the availability of a non-existing model of the "full" system

as stated in the motivation for the seminar. While convincing although informal arguments cannot be ruled out, we can and should nevertheless try to enlarge the applicability of formal methods.

The present article presents work in progress within the realm of the Unified Modeling Language (UML). In a previous work the concept of system model was defined, i. e., a precise notion of timed state transition system tailored for the definition of a formal semantics of any distributed object-oriented modelling language; see [1,2,3]. The claim is that system models allow the definition of an all-encompassing semantics of UML. The challenge now, thus, consists in either mapping any specification written in UML onto an adequate system model, or giving accurate criteria to decide if a system model satisfies a given UML specification. General ideas and directions for work in the context of UML 2.0 interaction diagrams are presented below.

The overall goal, therefore, is to provide means for the formal proof or adequacy of UML interactions on the basis of a system model for the "full" system. (That is, and quoting the motivation of the seminar, to address complexity.)

This contribution presents some hints for mapping the UML interactions (see [4,5]) onto the semantic framework of system models. The article is organised as follows. In Sect. 2 the concept of system model is briefly presented, in Sect. 3 the same is done for the UML interactions. Sect. 4 addresses the central concern, and proposes different possibilities: a denotational (Sect. 4.1) and an operational approach (Sect. 4.2), the latter subdivided into a global and a local variant. Finally Sect. 5 draws some conclusions and hints at a few directions for future work. The Appendix presents a summary of the local operational semantics for UML interactions, still unpublished and on which the local operational approach of Sect. 4.2 can be based.

2 Preliminaries

In the present section the concept of system model is briefly introduced; see e. g. [6,7]. Roughly speaking, a system model is a timed state transition system whose states are composed of three parts, namely a data store, a control store, and a message pool. Timed state transition systems are a special kind of transition systems.

2.1 Ingredients

A system model for an object-oriented specification language defines, on the one hand,

- a static part, basically values and store,
- a dynamic part, basically threads and control, and
- a finite number of message pools (in the simplest case, queues of incoming messages like method invocations and signals).

These ingredients are combined into states of a timed state transition system, and it "only" remains to define the transition function that suits the specification language of interest.

The static part defines universe(s) of basic values unified in UVAL, interpretation mechanisms for sets, relations, records, etc., as well as for object identifiers and classes, and a universe ULOC of locations.

A store is a partial function

$$\mathsf{DataStore} : \mathsf{ULOC} \hookrightarrow \mathsf{UVAL}$$

that maps locations onto values, with $\mathrm{dom}(\mathsf{DataStore}) \subseteq \mathsf{ULOC}$ the set of locations assigned by $\mathsf{DataStore}$. $\mathsf{DATASTORE}$ denotes the universe of all possible data stores.

The dynamic part defines a universe $\mathsf{UTHREAD}$ of threads and the universe of execution frames as

$$\mathsf{UFRAME} = \mathsf{UOID} \times \mathsf{UMETH} \times \mathsf{UVAL} \times \mathsf{UVAL} \times \mathsf{UPC} \times \mathsf{UOID}.$$

That is, a frame contains the identifier of the executing object, the method invoked, arguments and return values packed in a record, values of the local variables likewise packed in a record, the program counter, and the identifier of the caller object. To each thread corresponds then a stack of frames.

CONTROLSTORE denotes the universe of all possible control states. A control state contains information about all the threads being executed. There is more than one way to collect this information. The "thread centric view" simply establishes

$$\mathsf{ControlStore} : \mathsf{UTHREAD} \hookrightarrow \mathsf{UFRAME}$$

whereas the "object centric view" defines

$$\mathsf{ControlStore} : \mathsf{UOID} \times \wp(\mathsf{UTHREAD}) \hookrightarrow \mathsf{UFRAME}.$$

These both views are isomorphic. For the purposes of this work, the choice is irrelevant.

UEVENT denotes the universe of possible events. The events in an object are retrieved via the function $\mathsf{events} : \mathsf{UOID} \to \wp(\mathsf{UEVENT})$. Messages, collected in the universe UMSG, define special kinds of events: for each $m \in \mathsf{UMSG}$ we have events $\mathsf{SendEvent}(m)$ and $\mathsf{ReceiveEvent}(m)$, and

$$\begin{aligned}
&\mathsf{sender}, \mathsf{receiver} : \mathsf{UMSG} \to \mathsf{UOID},\\
&\mathsf{sender}(m) = oid \iff \mathsf{SendEvent}(m) \in \mathsf{events}(oid), \text{ and}\\
&\mathsf{receiver}(m) = oid \iff \mathsf{ReceiveEvent}(m) \in \mathsf{events}(oid).
\end{aligned}$$

Moreover, the message signature of an object, i. e., the messages that the object may receive or send, is provided by

$$\mathsf{msgIn}, \mathsf{msgOut} : \mathsf{UOID} \to \wp(\mathsf{UMSG}).$$

An event store contains all the events associated with an object, i. e., an event store is a function

$$\mathsf{EventStore} : \mathsf{UOID} \to \mathsf{Buffer}(\mathsf{UEVENT}),$$

where a buffer is a general structure to store and handle events, possibly deal with priorities, etc. The universe EVENTSTORE collects all such event stores.

2.2 TSTS

A timed state transition system is a tuple $(\mathsf{STATE}, \Delta, \mathsf{Input}, \mathsf{Output}, \mathsf{Init})$ with

$$\begin{aligned}
&\mathsf{STATE} \text{ a set of states,}\\
&\mathsf{Input} \text{ and } \mathsf{Output} \text{ the input and output channel sets, respectively,}\\
&\mathsf{Init} \subseteq \mathsf{STATE}, \text{ and}\\
&\Delta : (\mathsf{STATE} \times \mathsf{T}(\mathsf{Input})) \to \wp(\mathsf{STATE} \times \mathsf{T}(\mathsf{Output}))
\end{aligned}$$

where $\mathsf{T}(C)$ is the set of possible channel traces for the channel set C:

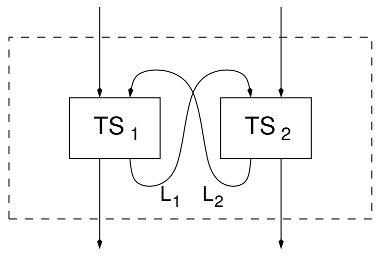


Fig. 1. Composition of TSTS's TS_1 and TS_2

$$\mathsf{T}(C) = C \rightarrow \mathsf{UMSG}^*$$

i. e., if $x \in \mathsf{T}(C)$ and $c \in C$, then by $x.c$ we denote the finite sequence of messages $x(c) \in \mathsf{UMSG}^*$.

Timed state transition systems behave as a Moore machine,[1] that is, the output depends on the state and not on the input:

$$(\sigma', y) \in \Delta(\sigma, x) \Rightarrow \forall x'.\exists \sigma''.(\sigma'', y) \in \Delta(\sigma, x')$$

The state transition function is furthermore total:

$$\Delta(\sigma, i) \neq \emptyset \quad \text{for any } s \in \mathsf{STATE}, i \in \mathsf{T}(\mathsf{Input})$$

Composition

Two timed state transition systems (see Fig. 1)

$$(\mathsf{STATE}_k, \Delta_k, \mathsf{Input}_k, \mathsf{Output}_k, \mathsf{Init}_k) \;\; (k = 1, 2)$$
$$\text{with } \mathsf{Output}_1 \cap \mathsf{Output}_2 = \emptyset$$

can be composed into

$$(\mathsf{STATE}, \Delta, \mathsf{Input}, \mathsf{Output}, \mathsf{Init})$$

where

$$\begin{array}{ll} \mathsf{STATE} = \mathsf{STATE}_1 \times \mathsf{STATE}_2 & L_1 = \mathsf{Output}_1 \cap \mathsf{Input}_2 \\ \mathsf{Input} = (\mathsf{Input}_1 \cup \mathsf{Input}_2) \setminus L & L_2 = \mathsf{Output}_2 \cap \mathsf{Input}_1 \\ \mathsf{Output} = (\mathsf{Output}_1 \cup \mathsf{Output}_2) \setminus L & L = L_1 \cup L_2 \end{array}$$

$$\begin{aligned} \Delta((s_1, s_2), x) = \{ \; & ((s_1', s_2'), y) : \exists z \in \mathsf{T}(L \cup \mathsf{Input} \cup \mathsf{Output}) \; . \\ & z\,|\,\mathsf{Input} = x \;\wedge\; (s_1', z\,|\,\mathsf{Output}_1) \in \Delta_1(s_1, x\,|\,\mathsf{Input}_1) \;\wedge \\ & z\,|\,\mathsf{Output} = y \;\wedge\; (s_2', z\,|\,\mathsf{Output}_2) \in \Delta_2(s_2, x\,|\,\mathsf{Input}_2) \;\; \} \end{aligned}$$

The composed state transition function Δ is denoted by $\Delta_1 \otimes \Delta_2$.

[1] A Moore machine is a finite state machine that produces an output for each state.

Interface abstraction

Given a timed state transition system $(\mathsf{STATE}, \varDelta, \mathsf{Input}, \mathsf{Output}, \mathsf{Init})$, the state transition function induces an interface function

$$\mathsf{B}[\varDelta] : \mathsf{STATE} \to (\overrightarrow{\mathsf{Input}} \to \wp(\overrightarrow{\mathsf{Output}}))$$

where $\overrightarrow{C}$ is the set of all valuations of channel C by streams:

$$\overrightarrow{C} = C \to (\mathsf{UMSG}^*)^\infty$$

i. e., if $y \in \overrightarrow{C}$ and $c \in C$, then $y(c) \in (\mathsf{UMSG}^*)^\infty$. The interface function $\mathsf{B}[\varDelta]$ abstracts away from the local encapsulated state and is defined by

$$\begin{aligned}\mathsf{B}[\varDelta](\sigma)(x\hat{\ }y) = \{\ x'\hat{\ }y' :\ & \exists\sigma' \in \mathsf{STATE}\ . \\ & (\sigma', x') \in \varDelta(\sigma, x) \land y' \in \mathsf{B}[\varDelta](\sigma')(y)\ \}\end{aligned}$$

where $\hat{\ }$ prepends a trace to a channel valuation:

$$\_\hat{\ }\_ : \mathsf{T}(C) \times \overrightarrow{C} \to \overrightarrow{C}$$

such that if $x \in \mathsf{T}(C)$, $y \in \overrightarrow{C}$ and $c \in C$, i. e., $x(c) \in \mathsf{UMSG}^*$ and $y(c) \in (\mathsf{UMSG}^*)^\infty$, then $x\hat{\ }y(c) \overset{\text{def}}{=} x(c)\hat{\ }y(c) \in (\mathsf{UMSG}^*)^\infty$ is the concatenation of $x.c$ and $y(c)$.

An interface function returns an I/O-behaviour. Two such behaviours

$$\begin{aligned}&F_k : \overrightarrow{\mathsf{Input}_k} \to \wp(\overrightarrow{\mathsf{Output}_k})\ (k = 1, 2)\\ &\text{with } \mathsf{Output}_1 \cap \mathsf{Output}_2 = \emptyset\end{aligned}$$

can be composed into $F : \overrightarrow{\mathsf{Input}} \to \wp(\overrightarrow{\mathsf{Output}})$ defined by

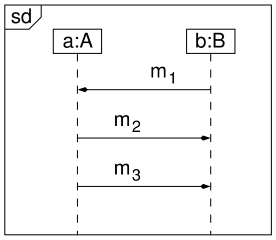
$$\begin{aligned}(F_1 \otimes F_2)(x) = \{\ y|\mathsf{Output} :\ & y \in \mathsf{Input}_1 \cup \mathsf{Input}_2 \cup \mathsf{Output}_1 \cup \mathsf{Output}_2 \\ & \land\ y|\mathsf{Input} = x|\mathsf{Input} \\ & \land\ y|\mathsf{Output}_1 \in F_1(y|\mathsf{Input}_1) \\ & \land\ y|\mathsf{Output}_2 \in F_2(y|\mathsf{Input}_2) \qquad \}\end{aligned}$$

$$\begin{aligned}\text{where}\quad \mathsf{Input} &= (\mathsf{Input}_1 \cup \mathsf{Input}_2) \setminus (\mathsf{Output}_1 \cup \mathsf{Output}_2)\\ \mathsf{Output} &= (\mathsf{Output}_1 \cup \mathsf{Output}_2) \setminus (\mathsf{Input}_1 \cup \mathsf{Input}_2)\end{aligned}$$

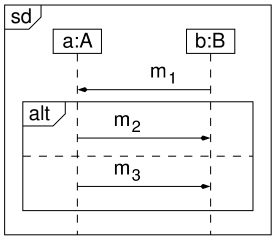
Proposition 1. *Interface abstraction and composition commute: given timed state transition systems* $(\mathsf{STATE}_k, \varDelta_k, \mathsf{Input}_k, \mathsf{Output}_k, \mathsf{Init}_k)$ *$(k = 1, 2)$,*
given states $\sigma_1 \in \mathsf{STATE}_1$ and $\sigma_2 \in \mathsf{STATE}_2$,

$$\mathsf{B}[\varDelta_1 \otimes \varDelta_2]((\sigma_1, \sigma_2)) = \mathsf{B}[\varDelta_1](\sigma_1) \otimes \mathsf{B}[\varDelta_2](\sigma_2)$$

The proof is done by induction on the time intervals; a proof for a variation of the present approach can be found in [8].



(a) Basic UML 2.0 interaction diagram



(b) UML 2.0 interaction diagram with disjunction

Fig. 2. Sample UML 2.0 interactions

2.3 System models

A system model is any timed state transition system with

$$\mathsf{STATE} \in \mathsf{DATASTORE} \times \mathsf{CONTROLSTORE} \times \mathsf{EVENTSTORE}$$

Given a system model $SM = (\Sigma_{SM}, \Delta_{SM}, \mathsf{Input}_{SM}, \mathsf{Output}_{SM}, \mathsf{Init}_{SM})$, a system run of SM is a finite or infinite sequence $\sigma_0 \cdot \sigma_1 \cdot \sigma_2 \cdot \ldots$ of Σ_{SM}-states such that $\sigma_0 \in \mathsf{Init}_{SM}$ and $(\sigma_{i+1}, \_) \in \Delta_{SM}(\sigma_i, \_)$ for all i.

To define a semantics for any object-oriented, distributed specification language (as e. g. UML) is the art of defining the transition function Δ depending on a state in STATE i. e., depending on a $\mathsf{DataStore} \in \mathsf{DATASTORE}$, a $\mathsf{ControlStore} \in \mathsf{CONTROLSTORE}$, and an $\mathsf{EventStore} \in \mathsf{EVENTSTORE}$, that moreover observes certain rules. These rules capture the essence of the object-oriented specification language of interest.

3 UML 2.0 interactions

UML 2.0 interactions describe message exchanges between instances. The pictorial representation of a basic interaction carries the intuitive meaning of a partial order of event occurrences; see Fig. 2(a). Indeed, the dispatch of a message occurs before the arrival of the same message, and the event occurrences on the lifeline of an instance are ordered from top to bottom. The natural semantic domain of interpretation for interactions is thus the universe of partial orders of events—or, more intuitive, simply the traces obtained by linearisation of those partial orders. So for the example in Fig. 2(a), the following two traces satisfy the interaction:

$$t_1 = \mathsf{snd}(\underline{b}, \underline{a}, m_1)\cdot\mathsf{rcv}(\underline{b}, \underline{a}, m_1)\cdot\mathsf{snd}(\underline{a}, \underline{b}, m_2)\cdot\mathsf{rcv}(\underline{a}, \underline{b}, m_2)\cdot\mathsf{snd}(\underline{a}, \underline{b}, m_3)\cdot\mathsf{rcv}(\underline{a}, \underline{b}, m_3)$$

$$t_2 = \mathsf{snd}(\underline{b}, \underline{a}, m_1)\cdot\mathsf{rcv}(\underline{b}, \underline{a}, m_1)\cdot\mathsf{snd}(\underline{a}, \underline{b}, m_2)\cdot\mathsf{snd}(\underline{a}, \underline{b}, m_3)\cdot\mathsf{rcv}(\underline{a}, \underline{b}, m_2)\cdot\mathsf{rcv}(\underline{a}, \underline{b}, m_3)$$

UML 2.0 puts a number of interaction-building operators at disposal. We have, for instance, two kinds of sequential composition, parallel composition, disjunction, loop, ignore, assert, and negation. An example is shown in Fig. 2(b); this interaction is satisfied by the following two traces:

$$t_3 = \mathsf{snd}(\underline{b}, \underline{a}, m_1) \cdot \mathsf{rcv}(\underline{b}, \underline{a}, m_1) \cdot \mathsf{snd}(\underline{a}, \underline{b}, m_2) \cdot \mathsf{rcv}(\underline{a}, \underline{b}, m_2)$$

$$t_4 = \mathsf{snd}(\underline{b}, \underline{a}, m_1) \cdot \mathsf{rcv}(\underline{b}, \underline{a}, m_1) \cdot \mathsf{snd}(\underline{a}, \underline{b}, m_3) \cdot \mathsf{rcv}(\underline{a}, \underline{b}, m_3)$$

In sequential composition, the behaviour of the resulting interaction is the behaviour of the first operand followed by the behaviour of the second one. Strict sequential composition requires the behaviour of the first operand be completed before starting with the behaviour of the second operand. Weak sequential composition only requires of each lifeline to complete the behaviour specified for it within the first interaction before starting with the behaviour specified for it within the second interaction. Two parallel interactions are to be executed simultaneously. Loop repeats the execution of its interaction argument at least a number of times m, where m can be zero, and at most n times, where n can be ∞. Ignore allows certain messages to be ignored.

UML 2.0 Interactions exemplarily describe patterns of communication, and negation allows one to discard unwanted ones. The interaction of Fig. 3(a) is *negatively* satisfied by the following trace:

$$t_5 = \mathsf{snd}(\underline{b}, \underline{a}, m_2) \cdot \mathsf{rcv}(\underline{b}, \underline{a}, m_2) \cdot \mathsf{snd}(\underline{a}, \underline{b}, m_3) \cdot \mathsf{rcv}(\underline{a}, \underline{b}, m_3)$$

Thus two kind of traces can be this way defined: positive and negative ones. Traces that are neither positive nor negative are termed inconclusive.[2] Assertion discards inconclusive traces, i. e., only positive traces of its interaction argument are positive for the assertion, any other trace is negative.

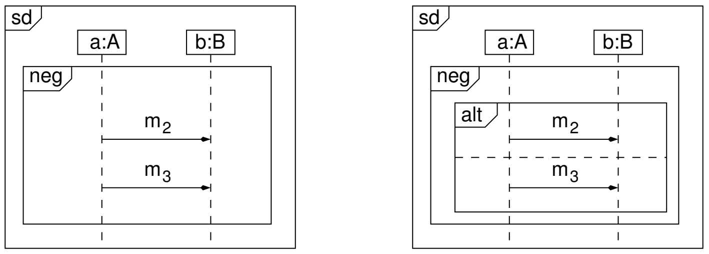
While the meaning of the negation of a basic interaction is straightforward, negation composed with other interaction-building operators was given more than one semantics. From a classical logic point of view, the interaction of Fig. 3(b) has no negative traces associated, since the negation of a disjunction (of basic interactions) is equivalent to the conjunction of the negation (of each of those basic interactions), and given that basic interactions cannot be negatively satisfied; this proposal can be found in e. g. [9]. An alternative proposal in [10] states that the interaction of Fig. 3(b) is negatively satisfied by the following two traces:

$$t_6 = \mathsf{snd}(\underline{b}, \underline{a}, m_2) \cdot \mathsf{rcv}(\underline{b}, \underline{a}, m_2)$$

$$t_7 = \mathsf{snd}(\underline{a}, \underline{b}, m_3) \cdot \mathsf{rcv}(\underline{a}, \underline{b}, m_3)$$

In the literature, and besides the above mentioned ones, many proposals for the semantics of interactions can be found; see e. g. [11,12,13,14]. These semantics are all trace based. The choice of the approach, for the purposes of this work, is irrelevant: we assume any trace-based semantics has been chosen and try either to mimic the inference rules of this semantics in terms of a transition function, or to decide whether a given transition function satisfies a given UML 2.0 interaction.

[2] More accurately, the semantics of an UML 2.0 interaction is 4-valued: a trace can be positive, negative, inconclusive, or both positive and negative.



(a) Negated basic interaction diagram

(b) Negated disjunction

Fig. 3. UML 2.0 Interactions with negated fragments

4 Mapping UML 2.0 interactions

The challenge now consists in determining the semantics of a UML 2.0 interaction in terms of system models. This section presents some informal thoughts about this subject. A denotational approach is first considered, that is based on a previous work; see [9]. We transform a system run into an event trace, from which we filter all those events that are not associated with any object involved in the interaction diagram. Then it only remains to check whether the trace this way obtained satisfies the interaction diagram.

We afterwards consider an operational approach. Here we identify three ways to tackle the problem. The first one, called global operational, is again based on previous work; see [15]. The other one, called local operational, correspond to a work in progress on a local or distributed version of the [global] operational semantics. Two variants can be identified, namely the local "observing" and the local "acting" operational approaches.

None of these possibilities has been exhaustively explored yet.

4.1 Denotational approach

In the denotational approach, it is checked whether an event trace satisfies an UML 2.0 interaction. The participating events are just send and receive events. In order to make use of this semantics, what we do, roughly speaking, is to project system runs onto sequences of send/receive events involving only instances present in the given interaction. Then the check is performed on the traces this way obtained.

Let $\models$ be a binary relation between event traces and UML 2.0 interactions, for instance the relation $\models_{\mathrm{p}}$ defined in [9]. We write $t \models S$ if the event trace t satisfies the interaction S.

Let $SM = (\Sigma_{SM}, \Delta_{SM}, \mathsf{Input}_{SM}, \mathsf{Output}_{SM}, \mathsf{Init}_{SM})$ be a system model, let $\rho = \sigma_0 \cdot \sigma_1 \cdot \sigma_2 \cdot \ldots$ be a system run of SM.

Let S be a UML 2.0 interaction.

We assume that events are unequivocally identified, i. e., that two arbitrary events, even if they are both a send (or receive) event of the *same* message from the *same* sender to the *same* receiver, can be distinguished (e. g., via a time stamp). Thus, if a transition was triggered by an event, then this event can be identified. We write

$$\sigma_k \xrightarrow{e}_{\rho} \sigma_{k+1}$$

for σ_k and σ_{k+1} two Σ_{SM}-states adjacent in ρ and e an event, if the following conditions hold

(a) $\sigma_k = (ds, cs, es)$,
(b) $\sigma_{k+1} = (ds', cs', es')$,
(c) $e \in es$, and
(d) $e \notin es'$.

If no such e exists for two Σ_{SM}-states σ_k and σ_{k+1} adjacent in ρ, then we write

$$\sigma_k \xrightarrow{\tau}_{\rho} \sigma_{k+1}$$

where τ is the so-called silent event, that cannot be generated by objects. Informally, if $\sigma_k \xrightarrow{e}_{\rho} \sigma_{k+1}$, we assume that event e is the trigger of the transition. It is not ruled out that transitions be triggered by anything other than an event; for this reason we need the concept of silent event.

We define the event trace $tr_S(\rho)$ generated by ρ and S as follows. First we build the trace $\overline{e}_0 \cdot \overline{e}_1 \cdot \overline{e}_2 \cdot \ldots$, where $\overline{e}_i$ is either an event or τ such that $\sigma_i \xrightarrow{\overline{e_i}}_{\rho} \sigma_{i+1}$. From this trace, we remove

(e) occurrences of the silent event,
(f) occurrences of events that are neither a send nor a receive event,
(g) occurrences of send and receive events in which either the sender or the receiver is not involved in S.

The resulting event trace is denoted by $tr_S(\rho)$.

The system model SM satisfies an UML 2.0 interaction S, written $SM \models S$, if there exists a system run ρ of SM such that $tr_S(\rho) \models S$.

Discussion. Given a system run $\rho = \sigma_0 \cdot \sigma_1 \cdot \sigma_2 \cdot \ldots$, we have assumed that at most one event triggers a transition from a state σ_k to a state σ_{k+1}. If there were more than one event in the difference of the associated event stores, then $tr_S(\rho)$ would possibly be not just one but a set of traces. In this case, we must rephrase $SM \models S$. A possibility is to state that $SM \models S$ if there exists a system run ρ of SM and a trace $t \in tr_S(\rho)$ such that $t \models S$.

The last two steps above for obtaining $tr_S(\rho)$, be this a single trace or a set of traces, should not be made if the system model is supposed to satisfy *only* the given interaction. In case they do apply, the last one could also read as follows:

(g) occurrences of send and receive events in which neither the sender nor the receiver is involved in S.

This might be useful if the message exchange is projected onto one or more lifelines. Other participants of the exchange would be then considered as the environment.

Regarding the environment, it was totally neglected. The implication of doing so is unexplored yet. One is tempted to treat the environment(s)[3] just as another lifeline (or other lifelines). The main difference between the environment and an ordinary lifeline is that the events on the environment are not ordered from top to bottom; this should not be disregarded.

Further issues are termination and deadlock, that in the above approach might be identified (i. e., confused). For this latter matter, a concept of final states of a system model might provide a solution.

4.2 Operational approach

The operational semantics, instead of only "passively" verifying traces as in the denotational approach of above, it can "actively" generate transitions of a system model. This approach is based on previously defined operational semantics for UML 2.0 interactions, namely a global operational semantics (see [15]) and a local operational semantics (still unpublished, a summary of it can be found in the Appendix). The global version recognises (or generates) event occurrences and transforms the original interaction into a target one. The local version, on the contrary, projects the given interaction onto its lifelines, and allows the parallel recognition (or generation) of event occurrences with the help of a minimal synchronisation mechanism.[4]

Global version

Let C be a channel set, let $c \in C$ be a channel, and let $x \in \mathsf{T}(C)$. We know that $x.c \in \mathsf{UMSG}^*$, i. e., $x.c$ is a finite sequence of messages. C can be in particular the set of input channels. Messages deposited in an input channel are to be delivered to the corresponding object. This matter is outside the scope of this article. Given a finite sequence of messages $x.c$, we let $\mathsf{x.c}$ denote the set of events that are to be added to the event store in order for those messages to be correctly delivered.

We let e denote either a send or a receive event, we let $\overline{e}$ denote either the silent event τ or an e.

Let $\rightarrow$ be a relation between two UML 2.0 interactions and an arbitrary $\overline{e}$, for instance the positive reduction relation $\rightarrow_{\mathrm{p}}$ defined in [15]. We write $S_1 \xrightarrow{\overline{e}} S_2$ if the interaction S_1 reduces via $\overline{e}$ to the interaction S_2.

We define a set of states by

[3] Some authors propose the use of different environmental instances, see for instance [16].

[4] The synchronisation mechanism avoids, for example, that two instances participating on a loop construct perform a different number of iterations.

$$\mathsf{STATE}' = \{(S, es) \mid es \in \mathsf{EVENTSTORE} \wedge S \text{ a UML 2.0 interaction}\}.$$

Given channel sets Input and Output, we furthermore define a function

$$\Delta' : (\mathsf{STATE}' \times \mathsf{T}(\mathsf{Input})) \to \wp(\mathsf{STATE}' \times \mathsf{T}(\mathsf{Output}))$$

with

(a) $((S_2, es \cup \{e\} \cup \bigcup_{c \in \mathsf{Input}} \mathsf{i.c}), \emptyset) \in \Delta'(((S_1, es), i))$
if $S_1 \xrightarrow{e} S_2$ and $e \notin es$, and
(b) $((S_2, es \cup \bigcup_{c \in \mathsf{Input}} \mathsf{i.c}), \emptyset) \in \Delta'(((S_1, es \cup E_1), i))$
if either $S_1 \xrightarrow{e} S_2$ and $E_1 = \{e\}$
or $S_1 \xrightarrow{\tau} S_2$ and $E_1 = \emptyset$.

The function Δ', thus, either generates or consumes the events along the reduction relation as derived by the operational semantics.

Let $SM = (\Sigma_{SM}, \Delta_{SM}, \mathsf{Input}_{SM}, \mathsf{Output}_{SM}, \mathsf{Init}_{SM})$ be a system model, let S be a UML 2.0 interaction. We say that the transition function Δ_{SM} is *compatible* with Δ' and S if

(i) $((S_2, es'), \emptyset) \in \Delta'(((S_1, es), i))$ implies
$((ds, cs, es'), \_) \in \Delta(((ds, cs, es), i))$ for any ds, for any cs, and
(ii) there exist $(ds, cs, es) \in \mathsf{Init}_{SM}$ and $i \in \mathsf{T}(\mathsf{Input})$
such that $((S, es), i) \in \mathrm{dom}(\Delta')$.

Now the system model satisfies the interaction, written $SM \models S$, if Δ_{SM} is compatible with Δ' and S.

Discussion. Notice that, instead of testing compatibility, one can state that the transition function Δ_{SM} by definition *is* the calculated Δ' rephrased onto STATE. In this case, on the one hand, the system model would satisfy the given UML 2.0 interaction and no other one, and great care should be taken in the definition of the set Init_{SM} of initial states. On the other hand, the behaviour of the system model would show just this one message exchange pattern, i. e., no other behaviour, as it can be for instance specified by an activity diagram or a state chart, would be satisfied by the system model obtained this way.

The relationship between input messages and event store is not quite satisfactorily explored. For instance the relationships that hold as stated in Sect. 2.1, namely

$$\mathsf{sender}, \mathsf{receiver} : \mathsf{UMSG} \to \mathsf{UOID},$$
$$\mathsf{sender}(m) = oid \iff \mathsf{SendEvent}(m) \in \mathsf{events}(oid), \text{ and}$$
$$\mathsf{receiver}(m) = oid \iff \mathsf{ReceiveEvent}(m) \in \mathsf{events}(oid),$$

must be updated when transforming a finite sequence of messages $i.c$ into the set $\mathsf{i.c}$ of events. Similar concerns apply to the data store and the control store. These seem to be useless if just one interaction is to be satisfied by the system model.

Finally, events generated by the environment through the use of input channels is not mirrored by the above definition when it comes to send events to the environment through the use of output channels. This matter, however, poses no true difficulty: after generation by step (a) of such an event, recognisable by having the environment as addressee, the event has to be casted to a message and put on an output channel (in this case, thus, the set of output messages is not empty).

Local version

In a distributed setting, it makes sense to consider the behaviour of each instance without simultaneously considering the behaviour of the context of that instance, be this context other instances or the environment. This approach is explored in [17]. This local operational semantics allows the derivation of judgements of the form $\gamma_1 \xrightarrow{\overline{e}} \gamma_2$, where γ_1 and γ_2 are configurations and $\overline{e}$ is either an event e or the silent event τ. Configurations, roughly speaking, consist of a number of pairs together with synchronisation information. There are as many pairs as lifelines involved in the given, initial, UML 2.0 interaction. The pairs associate each such lifeline with its still to process projection of the interaction. For details, see the Appendix.

As with the global operational semantics, information on the current configuration needs be recoverable from the state, on which a transition function based on the inference system for configurations can be based. We define the set of extended states by

$$\mathsf{STATE}' = \{(\gamma, es) \mid es \in \mathsf{EVENTSTORE} \wedge \gamma \text{ a configuration}\}.$$

Given channel sets Input and Output, we define a function

$$\Delta' : (\mathsf{STATE}' \times \mathsf{T}(\mathsf{Input})) \to \wp(\mathsf{STATE}' \times \mathsf{T}(\mathsf{Output}))$$

with

(a) $((\gamma_2, es \cup \{e\} \cup \bigcup_{c \in \mathsf{Input}} \mathsf{i.c}), \emptyset) \in \Delta'(((\gamma_1, es), i))$
if $\gamma_1 \xrightarrow{e} \gamma_2$ and $e \notin es$, and

(b) $((\gamma_2, es \cup \bigcup_{c \in \mathsf{Input}} \mathsf{i.c}), \emptyset) \in \Delta'(((\gamma_1, es \cup E_1), i))$
if either $\gamma_1 \xrightarrow{e} \gamma_2$ and $E_1 = \{e\}$
or $\gamma_1 \xrightarrow{\tau} \gamma_2$ and $E_1 = \emptyset$.

Along the same lines, given an UML 2.0 interaction S and a system model $SM = (\Sigma_{SM}, \Delta_{SM}, \mathsf{Input}_{SM}, \mathsf{Output}_{SM}, \mathsf{Init}_{SM})$, we define the concept of compatibility of Δ_{SM} with Δ' and S, and state that the system model satisfies the interaction, denoted by $SM \models S$, if Δ_{SM} is compatible with Δ' and S.

Discussion. In the same way as in the case of the global operational semantics, the transition function Δ' could be used to define Δ_{SM}. The relationship between input messages and event store needs likewise be explored, and similarly output messages generated in step (a) should be put in an output channel.

The benefit of using a local operational semantics becomes apparent in combination with the composition mechanism of system models. Indeed, system models can separately implement lifelines, and a single system model for the whole intended system can be obtained by composition. In this setting, configurations are no longer a number of pairs (one pair for each individual lifeline), but just one single pair. The additional information needed for synchronisation can be recorded in the data store. The data store or at least the synchronisation information, then, needs be shared and not local of each single system model implementing one lifeline. That is, a shared memory model or an ad hoc message exchange mechanism is indispensable, that deviates from the system model presented above. System models with shared memory pose a new challenge, since the composition mechanism needs be revised, and the same with the properties the present composition definition fulfils. If there is a real gain in doing so, though, is still to be explored.

5 Conclusions and outlook

The contribution of the work in progress reported above is a conciliation of the declarative language of UML 2.0 interactions with the more operational one of system models. One denotational and two operational approaches for the definition of the semantics of a UML interaction in terms of system models have been presented. These approaches are based on previously developed notions of semantics of UML 2.0 interactions, but in principle alternative proposals can also be employed. To what extent this is actually possible is a subject of future study.

System models provide a means for the interpretation of complex object-oriented specification languages. They offer a wide range of mechanisms that mirror well-established specification and programming techniques. In this context, the choice of the semantics of a UML 2.0 interaction makes it necessary to consider how to handle features that are not inherent in the language of the interactions. For this reason, the various possibilities presented in this survey show some loose ends that have been itemised in the respective discussions. In particular, none of the above approaches studies in detail the relationship between I/O-messages and internal messages. Moreover, when discussing the implementation of each lifeline by a single system model, the messages in the input and output channels possibly become internal ones by composition, i. e., they should be handled in the event store. It remains to check how to treat events from and to the environment in such a way that the corresponding input and output channels become feedback channels by composition; see Fig. 1.

Besides the denotational and the operational ones, an additional approach that could be termed equational, considers interactions as an equation on stream processing functions; see e. g. [18,19,20]. Indeed, the intuitive semantics of an interaction is a set of traces. An interaction can thus be regarded as an equation on stream processing functions.[5] We can therefore state that a system model satisfies an interaction if the interface abstraction of the system model satisfies the equation induced by the interaction. This seems straightforward for basic interactions. It is not trivial to combine the obtained system models in such a way that reflects the semantics of the different interaction building operators. The weak composition of stream processing function is worthy of special interest; see [7]. This matter is still unexplored.

The equational approach is appealing since it seems the most abstract one, whereas the operational approaches seem the most concrete ones. Once they all are fully developed, they should be proved to be equivalent, i. e., if a system model satisfies an interaction in one of those approaches, then the satisfaction relation must also hold according to another approach.

A chapter of itself deserves the negation of interactions. There is more than one proposed semantics for this construct, each with its own advantages and disadvantages. The great difficulty resides in the negation being non-classical and, in particular, in the fact that double negation does not correspond to the identity. This makes it very difficult to handle.

All in all, the interactions case poses an interesting challenge. There is a wide range of possibilities, and—after they have been deeper studied—discerning the better or most convenient one is the next task.

Acknowledgements. Special thanks go to Alexander Knapp for fruitful discussions.

References

1. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML – Towards a System Model for UML: The Structural Data Model. Technical Report TUM-I0612, Institut für Informatik, Technische Universität München (2006)
2. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML – Towards a System Model for UML: The Control and Scheduling Model. To appear. (2007)
3. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML – Towards a System Model for UML: The State Machine Model. To appear. (2007)
4. Object Management Group: UML 2.0 Superstructure Specification. Final adopted specification, OMG (2003) http://www.omg.org/cgi-bin/doc?ptc/03-08-02.
5. Object Management Group: UML 2.0 Superstructure Specification. Revised final adopted specification, OMG (2004) http://www.omg.org/cgi-bin/doc?ptc/03-08-02.
6. Henzinger, T.A., Manna, Z., Pnueli, A.: Timed transition systems. In de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G., eds.: Real-Time: Theory in Practice, (REX Workshop, Proceedings). Volume 600 of Lecture Notes in Computer Science., Springer (1992) 226–251
7. Krüger, I.: Distributed System Design with Message Sequence Charts. PhD thesis, Technische Universität München (2000)
8. Grosu, R., Rumpe, B.: Concurrent Timed Port Automata. Technical Report TUM-I9533, Institut für Informatik, Technische Universität München (1995)

[5] Note that the interface abstraction of a system model is a stream processing function.

9. Cengarle, M.V., Knapp, A.: UML 2.0 Interactions: Semantics and Refinement. In Jürjens, J., Fernandez, E.B., France, R., Rumpe, B., eds.: 3rd Int'l Workshop on Critical Systems Development with UML (CSDUML'04, Proceedings), Technical Report TUM-I0415, Institut für Informatik, Technische Universität München (2004) 85–99
10. Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: STAIRS towards formal design with sequence diagrams. Journal of Software and System Modeling (SoSyM) **4** (005) 355–367
11. Haugen, Ø., Stølen, K.: STAIRS - Steps to Analyze Interactions with Refinement Semantics. In Stevens, P., Whittle, J., Booch, G., eds.: 6th Int'l Conference on the Unified Modeling Language (UML'03, Proceedings). Volume 2863 of Lecture Notes in Computer Science., Springer, Berlin (2003) 388–402
12. Knapp, A.: A Formal Semantics for UML Interactions. In France, R., Rumpe, B., eds.: 2nd Int'l Conference on the Unified Modeling Language (UML'99, Proceedings). Volume 1723 of Lecture Notes in Computer Science., Springer, Berlin (1999) 116–130
13. Störrle, H.: Assert, Negate and Refinement in UML-2 Interactions. In Jürjens, J., Rumpe, B., France, R., Fernandez, E.B., eds.: Workshop on Critical Systems Development with UML (CSDUML'03, Proceedings), San Francisco, Technical Report TUM-I0317, Technische Universität München (2003)
14. Störrle, H.: Semantics of Interactions in UML 2.0. In: IEEE Symposium on Visual Languages and Formal Methods (VLFM'03, Proceedings), Auckland (2003)
15. Cengarle, M.V., Knapp, A.: Operational Semantics of UML 2.0 Interactions. Technical Report TUM-I0505, Institut für Informatik, Technische Universität München (2005)
16. Graubmann, P.: Describing interactions between MSC components: the MSC connectors. Computer Networks **42** (2003) 323–342
17. Cengarle, M.V., Knapp, A.: Distributed Operational Semantics of UML 2.0 Interactions. In preparation. (2007)
18. Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. PhD thesis, Technische Universität München (1996) In German.
19. Stephens, R.: A survey of stream processing. Acta Informatica **34** (1997) 491–541
20. Broy, M.: A logical basis for modular software and systems engineering. In Rovan, B., ed.: Current Trends in Theory and Practice of Informatics (SOFSEM'98, Proceedings). Volume 1521 of Lecture Notes in Computer Science., Springer (1998) 19–35
21. Pratt, V.: Modeling Concurrency with Partial Orders. Int. J. Parallel Program. **15** (1986) 33–71

Appendix: Local operational semantics (draft)

The definition contained in this section was developed together with Alexander Knapp. The local operational semantics is still under construction, we do not have a full proof of correctness or completeness yet.

The local operational semantics of interactions is based on the idea of distributing information to all the different lifelines and administrating some shared information necessary for correct synchronisation. This information is used by lifelines in order to find out the constraints, if any, that apply to the next decision and/or step to be taken. For the purposes of discerning decision points, the notion of path that uniquely identifies each combined fragment inside an interaction fragment is used. The initial interaction term is annotated with the paths to each operator. Each lifeline is associated with the projection onto it of the annotated interaction term. These pairs together with the synchronisation information constitute a configuration. An inference system permits the derivation of pairs of configurations related by an event; intuitively, this means that the outgoing configuration evolves, through raising the event, to the follow-up configuration.

More formally, the abstract syntax of the (fragment of the) language of UML 2.0 interactions introduced in Sect. 3 is captured by the context-free grammar of Tab. 1, where a basic interaction *Basic* is characterised as a pomset; see [9,21]. We assume two primitive domains for *instances* $\mathbb{I}$ and *messages* *Msg*. An *event* e is either of the form $\mathsf{snd}(s, r, m)$ or of the form $\mathsf{rcv}(s, r, m)$, representing the dispatch and the arrival of message m from *sender* instance s to *receiver* instance r, respectively. The set of events is denoted by $\mathbb{E}$.

$$\begin{array}{rcl}
\mathit{Interaction} & ::= & \mathit{Basic} \\
 & \mid & \mathit{CombinedFragment} \\
\mathit{CombinedFragment} & ::= & \mathsf{strict}(\mathit{Interaction}, \mathit{Interaction}) \\
 & \mid & \mathsf{seq}(\mathit{Interaction}, \mathit{Interaction}) \\
 & \mid & \mathsf{par}(\mathit{Interaction}, \mathit{Interaction}) \\
 & \mid & \mathsf{loop}(\mathit{Nat}, (\mathit{Nat} \mid \infty), \mathit{Interaction}) \\
 & \mid & \mathsf{ignore}(\mathit{Messages}, \mathit{Interaction}) \\
 & \mid & \mathsf{alt}(\mathit{Interaction}, \mathit{Interaction}) \\
 & \mid & \mathsf{neg}(\mathit{Interaction}) \\
 & \mid & \mathsf{assert}(\mathit{Interaction})
\end{array}$$

Table 1. Abstract syntax of interactions (fragment)

A given interaction term is first annotated with path information. *Paths* allow the univocal identification of each combined fragment within an interaction fragment. Paths are finite sequences of 1 (for left) and 2 (for right). The annotation procedure is defined as follows:

$$\begin{array}{l}
\mathit{num}(\mathit{const}, p) = \mathit{const} \\
\mathit{num}(\mathit{uop}(S), p) = \mathit{uop}_p(\mathit{num}(S, p.1)) \\
\mathit{num}(\mathit{bop}(S_1, S_2), p) = \mathit{bop}_p(\mathit{num}(S_1, p.1),\ \mathit{num}(S_2, p.2))
\end{array}$$

where *const* denotes a constant interaction fragment like e. g. Empty, $\mathit{uop}(-)$ a unary combined fragment like e. g. $\mathsf{loop}(-)$,[6] and $\mathit{bop}(-,-)$ a binary combined fragment like e. g. $\mathsf{strict}(-,-)$. For example, $\mathit{num}(\mathsf{seq}(\mathsf{par}(e_1, e_2), \mathsf{loop}(e_3)), 1) = \mathsf{seq}_1(\mathsf{par}_{1.1}(e_1, e_2), \mathsf{loop}_{1.2}(e_3))$.

[6] We disregard the lower and upper bounds that limit the number of iterations to be performed by a loop-construct. The treatment of these bounds is trivial, and would now only distract the reader from the focus of this exposure.

The *projection* $\pi_l(S)$ of an annotated term S onto the lifeline l replaces in S every occurrence of an event which is not active on l by Empty and is formally defined as follows:

$$\begin{aligned}
&\pi_l(\mathsf{Empty}) = \mathsf{Empty}\\
&\pi_l(e) = \begin{cases} e & \text{if } l = \alpha(e)\\ \mathsf{Empty} & \text{otherwise}\end{cases}\\
&\pi_l(uop_p(S)) = uop_p(\pi_l(S))\\
&\pi_l(bop_p(S_1, S_2)) = bop_p(\pi_l(S_1), \pi_l(S_2))
\end{aligned}$$

where $\alpha(e)$ denotes the *active* lifeline for event e, with $\alpha(\mathsf{snd}(s, r, m)) = s$ and $\alpha(\mathsf{rcv}(s, r, m)) = r$. That is, any configuration contains as many pairs as lifelines are involved in the original interaction term.

The *decisions* taken so far are recorded in a synchronisation variable h that is also part of any configuration. So for instance if a lifeline, faster than the other ones, arrives first to an alt-fragment, then the branch taken by this lifeline is the branch that any other lifeline must take. The path information that uniquely identifies this occurrence of the alt-operator is used to record the decision. So if the alt fragment is annotated with path information p and the branch chosen is the left one, then the variable h assigns 1 to p, i. e., $h(p) = 1$. Besides the choice of a branch in an alt-fragment, h needs to record the information about which lifelines have reached the end of the first operand of a strict-fragment. So if lifeline l has done so, i. e., the pair $l : \mathsf{strict}_q(\mathsf{Empty}, S_2)$ is within the current configuration, then the variable h assigns to q a set that contains l. If $h(q)$ contains every lifeline, then each pair $l : \mathsf{strict}_q(\mathsf{Empty}, S_2)$ can be replaced by $l : S_2$. In this way, it is ensured that the second operand is only entered when all lifelines finished processing the first one.

Special attention requires the loop-construct. During execution of a loop, the relevant synchronisation information is the number of iterations performed so far and if the loop has been already abandoned by any lifeline. This latter information is recorded in the shared variable. Regarding the first matter, we take advantage of the fact that a loop is equivalent to its unfolding as weak sequencing of its interaction argument. Indeed, $\mathsf{loop}(S)$ is equivalent to $\mathsf{seq}(S, \mathsf{loop}(S))$ if at least one more iteration is to be performed. In this case, we let $\mathsf{loop}_p(S)$ be equivalent to $\mathsf{seq}_p(S, \mathsf{loop}_{p.2}([p \mapsto p.2]S))$, where $[p \mapsto p.2]$ replaces the path p by $p.2$ of its annotated interaction argument. This substitution is formally defined as follows:

$$\begin{aligned}
&[p \mapsto p.n]\,const = const\\
&[p \mapsto p.n]\,uop_q(S) = \begin{cases} uop_{p.n.q'}([p \mapsto p.n]S) & \text{if } q = p.q'\\ uop_q(S) & \text{otherwise,}\end{cases}\\
&[p \mapsto p.n]\,bop_q(S_1, S_2) = \begin{cases} bop_{p.n.q'}([p \mapsto p.n]S_1, [p \mapsto p.n]S_2) & \text{if } q = p.q'\\ bop_q(S_1, S_2) & \text{otherwise}\end{cases}
\end{aligned}$$

(that is, $[p \mapsto p.n]$ is defined for any annotated term, even if p is not the prefix the paths involved). For example, the substitution $[1 \mapsto 1.2]$ has no effect on the annotated term $\mathsf{seq}_2(\mathsf{par}_{2.1}(e_1, e_2), \mathsf{loop}_{2.2}(e_3))$ since within that term paths begin with 2 and not with 1, whereas the same substitution applied on $\mathsf{seq}_1(\mathsf{par}_{1.1}(e_1, e_2), \mathsf{loop}_{1.2}(e_3))$ results in $\mathsf{seq}_{1.2}(\mathsf{par}_{1.2.1}(e_1, e_2), \mathsf{loop}_{1.2.2}(e_3))$.

Therefore, the synchronisation variable h maps paths to either a subset of $\{1, 2\}$ (for left-right decisions) or a subset of L (for strict sequencing), where L is a set of lifelines.

The initial *configuration* for an interaction term S is

$$\langle \{l : \pi_l(num(S)) \mid l \in L\}, \emptyset \rangle,$$

where L is the set of all lifelines involved in S. That is, each lifeline is associated with the projection onto it of the annotated interaction term, and the synchronisation function is empty. Configurations, in general, consist of a synchronisation function whose domain is contained in the set of paths of S and a set of pairs, one for each lifeline involved in S, that associate a lifeline with an annotated term.

The inference system allows the derivation of judgements of the form $\gamma_1 \xrightarrow{\overline{e}} \gamma_2$, where γ_1 and γ_2 are configurations and $\overline{e}$ denotes either an event e or the silent event τ.

In Tab. 2 a fragment of the inference system is presented. When writing the axioms and rules of the inference system, we may omit irrelevant information; this omission is to be understood as follows: in $\gamma_1 \xrightarrow{\overline{e}} \gamma_2$, whatever is omitted from γ_1 remains unchanged in γ_2.

For weak sequencing, an auxiliary predicate is needed that decides whether the first operand is Empty or equivalent to it (as e. g. $\mathsf{alt}(\mathsf{Empty}, \mathsf{Empty})$). This predicate is inductively defined as follows:

$$\begin{array}{l}
\varepsilon(\mathsf{Empty}) \\
\neg\varepsilon(e) \\
\varepsilon(uop_p(S)) \iff \varepsilon(S) \\
\varepsilon(bop_p(S_1, S_2) \iff \varepsilon(S_1) \wedge \varepsilon(S_2)
\end{array}$$

(Given that the interaction term is assumed to be projected on the lifeline of interest, for an event e we need not check if this lifeline is active for e: this is, by definition, the case.) With the help of this predicate, we let a lifeline advance to the second argument of a weak sequence. This is especially relevant when, in an alternative construct as first operand, the lifeline should be able not decide which one of both disjunctive terms is to be chosen (since the lifeline itself is not truly involved in any of both alternatives).

For ignore we use a function μ on events that returns the message carried by its argument, i. e., $\mu(\mathsf{snd}(s, r, m)) = m$ and $\mu(\mathsf{rcv}(s, r, m)) = m$.

In Tab. 2 the interaction building operators neg and assert are not considered. Their semantics in the local (or distributed) operational setting is still in development.

$$(\text{basic})\quad e \xrightarrow{e} \mathsf{Empty}$$

$$(\text{strict}_1)\quad \frac{S_1 \xrightarrow{\overline{e}} S_1'}{\mathsf{strict}(S_1, S_2) \xrightarrow{\overline{e}} \mathsf{strict}(S_1', S_2)}$$

$$(\text{strict}_2)\quad l : \mathsf{strict}_p(\mathsf{Empty}, S_2), h \xrightarrow{\tau} l : \mathsf{strict}_p(\mathsf{Empty}, S_2), h[p \mapsto h(p) \cup \{l\}] \quad \text{if } l \notin h(p)$$

$$(\text{strict}_3)\quad l : \mathsf{strict}_p(\mathsf{Empty}, S_2), h \xrightarrow{\tau} l : S_2, h \quad \text{if } h(p) = L$$

$$(\text{seq}_1)\quad \frac{S_1 \xrightarrow{\overline{e}} S_1'}{\mathsf{seq}(S_1, S_2) \xrightarrow{\overline{e}} \mathsf{seq}(S_1', S_2)} \qquad (\text{seq}_2)\quad \mathsf{seq}(\mathsf{Empty}, S_2) \xrightarrow{\tau} S_2$$

$$(\text{seq}_3)\quad \frac{S_2 \xrightarrow{\overline{e}} S_2'}{\mathsf{seq}(S_1, S_2) \xrightarrow{\overline{e}} \mathsf{seq}(S_1, S_2')} \quad \text{if } \varepsilon(S_1)$$

$$(\text{par}_1)\quad \frac{S_1 \xrightarrow{\overline{e}} S_1'}{\mathsf{par}(S_1, S_2) \xrightarrow{\overline{e}} \mathsf{par}(S_1', S_2)} \qquad (\text{par}_2)\quad \mathsf{par}(\mathsf{Empty}, S_2) \xrightarrow{\tau} S_2$$

$$(\text{par}_3)\quad \frac{S_2 \xrightarrow{\overline{e}} S_2'}{\mathsf{par}(S_1, S_2) \xrightarrow{\overline{e}} \mathsf{par}(S_1, S_2')} \qquad (\text{par}_4)\quad \mathsf{par}(S_1, \mathsf{Empty}) \xrightarrow{\tau} S_1$$

$$(\text{ignore}_1)\quad \mathsf{ignore}(M, \mathsf{Empty}) \xrightarrow{\tau} \mathsf{Empty} \qquad (\text{ignore}_2)\quad \frac{S \xrightarrow{\overline{e}} S'}{\mathsf{ignore}(M, S) \xrightarrow{\overline{e}} \mathsf{ignore}(M, S')}$$

$$(\text{ignore}_3)\quad \mathsf{ignore}(M, S) \xrightarrow{\overline{e}} \mathsf{ignore}(M, S) \quad \text{if } \mu(e) \in M$$

$$(\text{alt}_1)\quad l : \mathsf{alt}_p(S_1, S_2), h \xrightarrow{\tau} l : S_1, h[p \mapsto \{1\}] \quad \text{if } 2 \notin h(p)$$

$$(\text{alt}_2)\quad l : \mathsf{alt}_p(S_1, S_2), h \xrightarrow{\tau} l : S_2, h[p \mapsto \{2\}] \quad \text{if } 1 \notin h(p)$$

$$(\text{loop}_1)\quad l : \mathsf{loop}_p(S), h \xrightarrow{\tau} l : \mathsf{Empty}, h[p \mapsto \{1\}] \quad \text{if } 2 \notin h(p)$$

$$(\text{loop}_2)\quad l : \mathsf{loop}_p(S), h \xrightarrow{\tau} l : \mathsf{seq}_p(S, \mathsf{loop}_{p.2}([p \mapsto p.2]S)), h[p \mapsto \{2\}] \quad \text{if } 1 \notin h(p)$$

Table 2. Positive local operational semantics (fragment)