T U M

I N S T I T U T F Ü R I N F O R M A T I K

Semantics of UML

Towards a System Model for UML

Part 3: The State Machine Model

Version 0.7

Manfred Broy, María Victoria Cengarle, Bernhard Rumpe

with special thanks to

Michelle Crane, Jürgen Dingel, Bran Selic

TUM-I0711
Februar 07

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

[BCR07b] M. Broy, M. Cengarle, B. Rumpe.
Towards a System Model for UML. Part 3. The State Machine Model.
Munich University of Technology, Technical Report
TUM-I0711. February 2007.
www.se-rwth.de/publications

TUM-INFO-02-I0711-100/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2007

Druck: Institut für Informatik der
Technischen Universität München

Semantics of UML

Towards a System Model for UML

Part 3: The State Machine Model

Version 0.7

Manfred Broy[1]
María Victoria Cengarle[1]
Bernhard Rumpe[2]

[1] Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

[2] Institute for Software Systems Engineering
Technische Universität Braunschweig

with special thanks to
Michelle Crane
Jürgen Dingel
Bran Selic

Table of Contents

1. Introduction 3

2. State Transition Systems describing Objects 4

2.1 Message Signature of an Object 4

2.2 State of Objects 5

2.3 Timed Asynchronous Communication Channels 6

2.4 State Transition Systems 8

3. Behaviour and Composition 9

3.1 Composition of State Transition Systems 9

3.2 Interface Behaviour and Interface Abstraction 11

4. Concluding Remarks 13

4.1 Further extensions 13

5. References 14

6. Appendix: Central Model of Interaction: Streams and Stream Processing Components 15

6.1 Types of Models for Interactive Systems 15

6.2 Streams 16

6.3 Channels and Histories 19

6.4 Interfaces, I/O-Behaviours, Time, and Causality 21

6.4.1 Interfaces 21

6.4.2 Causality 22

6.5 Composition of Interface Behaviour 23

1. Introduction

In the first two parts of this report, we have clarified that the system model is designed to constitute the core and foundation of a formal definition of the UML semantics. Semantics in our terms is the “meaning” of a UML model – regardless whether this is a structural or a behavioural model [HR04].

This is the third part of the system model, namely the “state machine part”. The first part of the system model, namely the “static part” defines how data stores are built and structured. This static part together with a rationale and motivation of this effort as well as the roadmap to define the complete necessary system model is given in [BCR06].

The second part [BCR07], called the “control part”, defines how the information needed for the definition of the interaction between objects is coded on an event and a message level. Part 2 describes a general notion of message and event that allows various forms of interaction including asynchronous message passing, method calls and returns. Part 2 moreover defines a notion of concurrent threads, and maps those threads into a decentralised view, where “thread tokens” are passed around between objects to model various forms of method calls. These definitions allow us to encode ordinary sequential program models like those used in C++ and Java, as well as notions of active objects, dynamic threads, etc.

We assume the reader is familiar with the definitions from [BCR06] and [BCR07] as we refer to these definitions without reintroducing them here.

However, we briefly repeat the main concepts from the first two parts: A data store, roughly speaking, consists of a set of objects each equipped with a unique object identifier. An identifier among other things acts as reference to the locations (the attributes) of the object. The store furthermore contains a mapping from locations to values contained in the store at a certain point of time. Object identifiers and locations are ordinary values and thus can be stored and passed around. Further values are e. g. integer numbers or Booleans.

A control store furthermore comprises a stack for each thread in the system as well as an event store that allows objects to manage events that need to be handled. Various scheduling strategies can be defined rather freely that delay, ignore or handle incoming messages in that store.

This third part of the report focuses on the interaction between individual objects on a more abstract, state based view. Each object is described by its data, control and event stores. Object signatures are given through their method and message interfaces as well as the events an object reacts to. These concepts give us the basis to determine the state space and transitions of each object.

The purpose of a state transition system (STS) is to link the state space of an object system with its behaviour. While the stores describe structural issues and therefore the state space, a state transition system uses this state space and describes behaviour in form of reactions of objects to incoming events. Thus a state transition system consists of a state space, which is a set of states, and a state transition function.

One of the main features of the STS developed here is that it does not only describe the behaviour of a single object, but also of compositions of groups of objects. STS therefore are compositional. This allows the description of the whole system as a large STS as well as the composition of views on collaborating group of objects, thus giving rise to the notion of “component”.

In this report, we introduce the general notion of timed state transition systems and link it to the data store, the control store, and the event store. As a general result of this report, we have a complete description of how systems are decomposed into objects, what states objects may have, and how objects interact.

The theory of STS is based on the theory of streams to describe I/O-behaviour of state transition systems appropriately. This theory of streams is sketched in the Appendix to allow the reader a self-contained understanding of the following definitions.

2. State Transition Systems Describing Objects

As motivated in the last section, we use state transition systems (STS) for a modular theory of object structures, behaviour and interaction between objects. These objects are allowed to act concurrently, pass messages as desired, but also to share threads for a classical sequential programming paradigm. In this section, we put together all elements necessary to define timed state transition systems (TSTS) to describe all relevant parts of objects. This relies heavily on the elements defined in the first two parts of the system model [BCR06, BCR07].

2.1 Message Signature of an Object

In our system model we allow for many events. An event can be the sending or receiving of a message or an internal event corresponding to a certain state change. Typical examples for messages are method invocation messages, return messages, asynchronous communication messages, signals and timeouts. Part 2 [BCR07] defines the universe of messages by UMESSAGE.

Basically, the structure of the messages exchanged between objects of classes is determined by the signature of the objects, which includes methods and their parameters. Each message furthermore contains the object identifier of the object that has generated the message, and the object identifier of the object that is the receiver.

The messages that an object $o \in$UOID may accept and send are (see Part 2 [BCR07]):

- msgIn(o) $\subseteq \wp$(UMESSAGE) for the incoming messages and
- msgOut(o) $\subseteq \wp$(UMESSAGE) for the outgoing messages

2.2 State of Objects

According to Part 2 [BCR07], the states of an object system are defined by USTATE. Such a state consists of a data store, a control store, and an event store. A state of an individual object $o \in UOID$ consists of the values of its attributes (stored in locations), the buffer of events to process, and a stack of frames for each active thread (see Part 2 [BCR07]). The state space of $o \in UOID$ is given by a function states(o):

- $states(o) \subseteq \wp(\ (ULOC \to UVAL) \times (UTHREAD \to Stack(UFRAME)) \times Buffer(UEVENT)\ )$

In a given object system state $(ds,cs,es) \in USTATE$, the state of an object $o \in UOID$ is determined by:

- $state((ds,cs,es),o) = (vals(ds,o),\ cs(o),\ es(o)) \in states(o)$

where vals is a function on data stores and objects that retrieves the mapping of attribute names to values for the given object in the given data store; see Part 1 [BCR06]. We extend the notion of state to groups of objects, because this allows us to deal with composition of state and behaviour[1]:

Definition of state space of a group of objects

- $state: USTATE \times \wp(UOID) \to (ULOC \to UVAL) \times (UOID \to (UTHREAD \to Stack(UFRAME)) \times (UOID \to Buffer(UEVENT))$

  with $state((ds,cs,es),oids) = (\bigoplus_{o \in oids} vals(ds,o),\ cs|oids,\ es|oids)$,
  where cs|oids is cs restricted to the objects in $oids \subseteq \wp(UOID)$
  and es|oids is analogously defined.
- For simplicity, we identify $state((ds,cs,es),oid) \cong state((ds,cs,es),\{oid\})$, as both are isomorphic.
- In general, state((ds,cs,es),oids) represents the product of its object states, as these are also isomorphic:
  $$\bigoplus_{o \in oids} state((ds,cs,es),o) \quad \cong \quad state(u,oids)$$
- The possible state space of a group of objects is accordingly defined by
  $$states(oids) = \{\ state((ds,cs,es),oids)\ |\ (ds,cs,es) \in USTATE\ \} \cong \bigoplus_{o \in oids} states(o)$$
- We also identify (due to isomorphism): $states(o) \cong states(\ \{o\}\ )$.

---

[1] The operator $\oplus$ was introduced in Part 1 [BCR06] to combine partial mappings. For instance for variable assignments a and b, we have that $(a \oplus b)(x) = a(x)$ if $x \in dom(a)$ and $(a \oplus b)(x) = b(x)$ otherwise. If we consider functions as sets of pairs (that fulfil the function condition, i. e., no left component is associated with two different right components), then the operator $\oplus$ is simply the union of functions. The operator $\oplus$ is further extended to tuples of partial functions: $(a,b) \oplus (c,d) = (a \oplus c, b \oplus d)$.

2.3 Timed Asynchronous Communication Channels

For our system model we assume a discrete global time; see the Appendix. Each step (transition) of the state machine corresponds to a tick of time. A system executes in steps, each consuming a fixed amount of time. Timed state transition systems (TSTS) are transition systems that deal with this kind of paradigm. TSTS are introduced in the next section. Roughly speaking, in each step a finite set of input events is provided to a TSTS, and a finite set of output events are produced by the TSTS.

One crucial question is the choice of the appropriate communication or interaction mechanism. Two basic flavours are asynchronous and synchronous. In the Appendix, we discuss advantages of both and justify our decision to use the asynchronous approach. However, both approaches can model each other, and we therefore have encoded synchronous method calls into an asynchronous message passing mechanism. In particular, our time based approach allows us to use a simple abstraction on the time scale to look at communication as being synchronous.

In our system model the object and component instances cooperate by asynchronous message passing; see [BS01]. Method invocation is therefore modelled by the exchange of two messages, the method invocation message and the method return message.

Communication between objects is dealt with by channels. Channels, on the one hand, allow us to compose groups of objects into larger units and hide their internal communication. On the other hand, UML provides linguistic constructs like "pins" in some of its diagrams; these pins resemble communication lines between objects.

A communication channel is a unidirectional communication connection between two objects (or other communication entities). Each channel has a name, e. g. $c \in UCN$, and the type of messages that may flow through c is given by type(c). The set of all possible observations on a channel $c \in UCN$ is denoted by $\mathbb{H}(c)$[2] and the observations on a channel set $C \subseteq UCN$ by $\mathbb{H}(C)$. Each object has a number of incoming and outgoing channels and each message knows through which channel it flows:

[2] See the definition of histories in the Appendix.

Definition of channels signatures of objects

- UCN denotes the universe of channel names.
- sender, receiver: UCN $\rightarrow$ UOID
  assign a sending and a receiving object, respectively, to each channel.
- channel: UMESSAGE $\rightarrow$ UCN
  assigns a channel to each message.
- inC, outC: UOID $\rightarrow$ $\wp$(UCN)
  denote the *channel signature* of each object, defined by:
  inC(oid) = { c | receiver(c) = oid }
  outC(oid) = { c | sender(c) = oid }
- Messages flow on the channels they belong to:

  sender(m) = oid $\Rightarrow$ sender(channel(m)) = oid

  receiver(m) = oid $\Rightarrow$ receiver(channel(m)) = oid

  for each m$\in$UMESSAGE, oid$\in$UOID (see Part 2 [BCR07]).
- The type of each channel c$\in$UCN is given by

  type(c) = { m$\in$UMESSAGE | channel(m) = c }

The existence of the sender and the receiver function has an interesting impact. Each message knows on which channel it flows and from which object it originates. This allows us to conclude that each channel can be in the output signature of only one object:

$$\forall a,b\in UOID:\ a\neq b \Rightarrow outC(a) \cap outC(b) = \emptyset$$

(This also follows from the second and fourth bullets above.)

In the system model, we assume a fine enough time granularity, i. e., so fine that the output in a step does not depend on the input received in that step. This way, strong causality between input and output is preserved. The composition of state machines is moreover simplified, since feedback within one time unit is ruled out, and thus causal inconsistencies are avoided. Even so, we are able to abstract away from the actual (real-time) time point of events.

The above is a very flexible concept of systems including e. g. classical sequential systems (in this case, there are only one input and one output channel). For instance, we may restrict the input and output events in such a way that, in each step, at most one input event is received or one output event is dispatched. At the other extreme, we can model highly concurrent systems with a large number of input and output events in one state transition step.

2.4 State Transition Systems

A timed state transition system (TSTS) is given by a state transition function with input and output (a generalised Moore automaton). Thus each object $o \in UOID$ can be described through a nondeterministic state transition function $\Delta.o$ of the form

$$\Delta : UOID \rightarrow (USTATE \times (UCN \rightarrow UMESSAGE^*)) \rightarrow \wp(USTATE \times (UCN \rightarrow UMESSAGE^*))$$

More precisely, $\Delta.o$ is a function of the form

$$\Delta.o : (states(o) \times T(inC(o))) \rightarrow \wp(states(o) \times T(outC(o)))$$

where T(C) denotes the set of channel time slices for the channels in C; see the Appendix.

We assume the state transition function describes the behaviour of a Moore machine [Kat93]. The output in a set therefore only depends on the state, not on the input. This property is captured by the following rule:

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

The rule expresses that if an output y is possible for some state σ and some input x, this output is possible in this state for all other inputs, too. One way to interpret this rule is that the granularity of time is fine enough to trace state changes in such a detailed way that the reaction to input is always delayed by at least one time unit (one state transition step). The immediate consequence is that feedback cycles include a time step and thus preserve causality. Another consequence is that the output of a transition is independent of the input of this transition and, therefore, intermediate storage for the output in the state space of the described object is inevitable.

Furthermore, we require the state transition function to be “total” (also called input enabled). This means that Δ has to provide a reaction to any possible sequence of inputs in any state:

$$\Delta.o(\sigma, i) \neq \emptyset \quad \text{for any } \sigma \in states(o),\ i \in T(inC(o))$$

This property only expresses that a UML system reacts in a specified way to every input in its set of input patterns.[3]

A timed state transition system (TSTS) is defined as follows:

---

[3] Of course there may exist certain inputs that are not valid at certain states.

Definition of a timed state transition system (TSTS) for one object

A timed state transition system (TSTS) for object $o \in UOID$ is defined by

- $tsts(o) = (states(o), \Delta.o, inC(o), outC(o), init(o))$

where $init(o) \subseteq states(o)$ is a non empty set of initial states

and the state transition function $\Delta.o$ is as described above:

- $\Delta.o : states(o) \times T(inC(o)) \rightarrow \wp(states(o) \times T(outC(o)))$
- $(\sigma', y) \in \Delta.o(\sigma, x) \Rightarrow \forall x': \exists \sigma'': (\sigma'', y) \in \Delta.o(\sigma, x')$
- $\Delta.o(\sigma, i) \neq \emptyset$ for any $\sigma \in states(o), i \in T(inC(o)))$

Note that each object o has *exactly one single* timed state transition system tsts(o). However, as tsts(o) is a nondeterministic state machine, it allows all forms of underspecification. Therefore, there is no need to add an additional concept of underspecification by, e. g., assigning to each object a set of possible TSTS. Any UML model, however, may have an impact on the elements of a timed state transition system. For instance, the sets of reachable states can be constrained, the initial states restricted to be a singleton, or the nondeterminism reduced by enforcing a behaviour that is deterministic in reaction and time.

3. Behaviour and Composition

Given the state transition systems for individual objects, we are now interested in a description of the overall system. In particular, we are interested in at least two views of the system and its components, namely a state transition based view (as defined above) and a behavioural view (as defined in the Appendix).

Both views, state based and behavioural, are compositional and their composition is moreover fully compatible. The following property is the main result of this section: for any set of objects $oids \subseteq UOID$, the following holds:

$$B[\bigoplus_{o \in oids} tsts(o)] = \bigoplus_{o \in oids} B[tsts(o)]$$

where B is an interface function that abstracts away from the local encapsulated state, as will be defined below. In words, abstraction of composed objects is composition of abstracted objects.

3.1 Composition of State Transition Systems

In this section, we introduce a composition mechanism for state machines analogous to the composition of behaviours (see the Appendix). Let o_k (k = 1, 2) be any two objects, let

$$\Delta.o_k: (states(o_k) \times T(inC(o_k)) \rightarrow \wp(states(o_k) \times T(outC(o_k)))$$

be their corresponding TSTS. According to the formalisation introduced so far, the output channel sets $outC(o_1)$ and $outC(o_2)$ are disjoint. This poses no restriction for object-oriented systems, as both channels and messages flowing on these channels are distinguished by the sending object. We define the feedback lines as follows (see Fig. 1):

Emitted by 1, consumed by 2:	$L_1 = inC(o_2) \cap outC(o_1)$
Other way round:	$L_2 = inC(o_1) \cap outC(o_2)$
Internal feedback lines:	$L = L_1 \cup L_2$
Remaining input:	$I = (inC(o_1) \cup inC(o_2)) \setminus L$
Remaining output:	$O = (outC(o_1) \cup outC(o_2)) \setminus L$

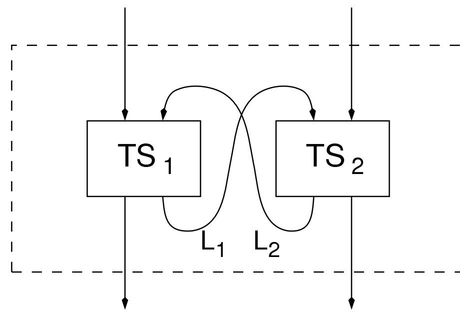

Fig.1 Composition of TSTS, feedback lines

The composed state space and transition system can be defined as follows using the composition of the states of the objects $states(\{o_1,o_2\})$:

$$\Delta: (states(\{o_1,o_2\}) \times T(I)) \rightarrow \wp(states(\{o_1,o_2\}) \times T(O))$$

by

$$\Delta((\sigma_1, \sigma_2), x) = \{ \ ((\sigma'_1, \sigma'_2), z|O) \ | \ \exists z \in T(L \cup I \cup O): \ z|I = x \wedge (\sigma'_k, z|outC(o_k)) \in \Delta_k(\sigma_k, z|inC(o_k)) \ \text{ for } k = 1, 2 \}$$

Given that the output only depends on the state, and according to the fact that each of the machines is a Moore machine, the formula can always be fulfilled.

The formula expresses that the input to the composed machine is split into input to the first machine and input to the second machine. With this input, and possibly additional input from feedback, both machines carry out their transition and produce output. The new states of the small machines define the new state of the composed machine; the output of the composed machine is built using the output of the small machines.

Note that this composition of Moore machines yields Moore machines.

For the composition of two transition functions we write

$\Delta = \Delta_1 \otimes \Delta_2$

Composition of TSTS is commutative, i. e., $\Delta_1 \otimes \Delta_2 = \Delta_2 \otimes \Delta_1$. Furthermore, as we do not have overlapping input channels, it is also associative and can thus be generalised to any finite and (by induction) also infinite set of TSTS.

Hence we define a TSTS for any set of objects:

Definition of a timed state transition system (TSTS) for a set of objects

A timed state transition system (TSTS) for a set objects $oids \subseteq UOID$ is defined by

- tsts(oids) = (states(oids), Δ(oids), I, O, inits)

with
- transitions $\Delta(oids) = \otimes_{o \in oids} \Delta(o)$,
- initial states $inits = \otimes_{o \in oids} init(o)$,
- remaining input $I = (\cup_{o \in oids} inC(o)) \setminus (\cup_{o \in oids} outC(o))$
- remaining output $O = (\cup_{o \in oids} outC(o)) \setminus (\cup_{o \in oids} inC(o))$

In particular, we now have a TSTS for the whole system that includes all snapshots and all system states and thus is capable of describing any behavioural and structural restrictions by tsts(UOID).

Note that we have a closed world assumption now: The overall system transition system tsts(UOID) does not have external channels anymore, but incorporates all "objects". This also includes objects that have direct connections to interfaces to other systems, mechanical devices or users and thus can act as surrogates for the context of the system. In [Rum96] we have discussed how to deal with this to model open, reactive systems in a closed world assumption and what the advantages are.

3.2 Interface Behaviour and Interface Abstraction

State machines provide a very detailed model for systems, because the structure of the state is shown explicitly. However, if a state is encapsulated, a representation of the system behaviour without considering the structure of states seems most appropriate. This is essentially what we call the interface of a system. The interface abstraction of a state machine is made explicit in the following.

First, we define the approach of mapping a TSTS to an I/O-behaviour; see the Appendix. Given a system represented by the TSTS whose transition function is

$$\Delta: (STATE \times T(I)) \rightarrow \wp(STATE \times T(O))$$

the state transition function Δ naturally induces an I/O-behaviour function:

$$B[\Delta]: STATE \rightarrow (\vec{I} \rightarrow \wp(\vec{O}))$$

where $\vec{C}$ denotes the set of channel histories for the channels in C; see the Appendix.

B[Δ] provides the interface abstraction of the state transition function Δ. This interface abstraction produces a behavioural description that excludes internal states and transition steps, such that the overall behaviour becomes easier to grasp.

For each state $\sigma \in$ STATE, each input pattern $z \in T(I)$, and each input channel valuation $x \in \vec{I}$, the interface function B[Δ] is the inclusion maximal solution of the recursive equation

$$B[\Delta] (\sigma).(\langle z \rangle\hat{}x) = \{\langle r \rangle\hat{}y: \exists \sigma' \in \text{STATE}: (\sigma', r) \in \Delta(\sigma, z) \wedge y \in B[\Delta] (\sigma').x\}$$

Note that the right hand side of the equation above is inclusion monotonic in B[Δ]. If we add elements to B[Δ](σ).x, the set is also increased. B[Δ] is recursively defined by an inclusion monotonic function, which even is guarded. Hence there exists a unique inclusion maximal solution. B[Δ](σ) defines an I/O-behaviour for any initial state σ, which represents the behaviour of the component described by the state machine Δ if initialised by the state σ.

The fact that the TSTS is input enabled and provides the behaviour of a Moore machine (output reactions have at least one tick delay) guarantees that B[Δ](σ) is a causal I/O-behaviour. B[Δ] generalises to our set Init of initial states of the TSTS:

$$B[\Delta] (\text{Init}) = \{ y \in B[\Delta] (\sigma): \sigma \in \text{Init} \}$$

The adaptation of this approach to the TSTS of our system model allows us to define I/O-behaviour of individual objects as well as compositions of objects:

Definition of I/O-behaviour of objects

The I/O-behaviour of an object oid∈UOID is defined by

- behaviour(oid) $\in$ IH(inC(oid)) → ℘(IH(outC(oid)))
- behaviour(oid) = B[Δ] (init)   where (S, Δ, I , O, init) = tsts(oid)

The I/O-behaviour of a set of objects oids ⊆ UOID is consequently defined by:

- behaviour(oids) $\in$ IH(I) → ℘(IH(O))
- behaviour(oids) = B[Δ] (init)   where (S, Δ, I , O, init) = tsts(oids)

Note that behaviour(oid) = behaviour({oid}). Furthermore, both models and their composition operators fit together:

$$B[\Delta_1 \otimes \Delta_2] = B[\Delta_1] \otimes B[\Delta_2]$$

The proof is done by induction of the time intervals and can be found for example in [GR95] on a variation of this approach.

In our terms: composing TSTS and deriving their abstraction to I/O-behaviours gives exactly the same result as deriving the abstraction to I/O-behaviours and composing them.

Now we have two models for systems available: state machines and interface behaviours that are fully compatible. This is an important property of the construction of this system model, as it demonstrates that mapping UML constructs to either interface behaviours or state machines is both possible.

4. Concluding Remarks

In this report, we have introduced the third part of the system model, namely the general notion of timed state transition systems, and integrated it with the object, control and event stores. As a general result of this report, we have a complete description of how systems are decomposed into objects, what states objects may have and how objects interact. As motivated in the first part of the report, we have developed the mathematical theory in layers, each building up an algebra that introduced some universe of elements, functions and laws for these functions.

As motivated in the first part, we have chosen this approach, because we want a semantics that is not biased by the choice of a concrete formal language or tool. Even the use of mathematical theories probably has biased the semantics a little, but we hope as little as possible. Such bias easily creeps in. We furthermore did not address executability, because this includes one of the biggest biases a modelling language can have: A model shall be underspecified, it shall be open for a specification of many different implementations. An executable semantics for an underspecified UML model therefore must necessarily contain choices added by the semantic mapping.

To prevent that, we have chosen a specific style of description. The form of description used throughout these three parts allows us to leave quite a number of definitions open. In general, we have introduced a universe of X and then characterised the properties of its elements, without fully determining how many elements X has or how these elements look like. Sometimes, we only described a subset of the elements of X, but allowing other kinds of elements to be in X as well (e. g., the universes of events, messages and values are defined in such a way).

This gives us and others many chances to specialise such variation points according to specific situations. In “UML words”, we could for example define a “system model profile” that specialises in sequential, single threaded systems like in java (without use of Threads), with |UTHREAD|=1, to static systems without introduction of new objects, or absence of subclasses, etc.

While the system model is an underlying basis for this kind of systems, it does not provide such specialisation directly; this is matter of further work.

4.1 Further extensions

Of course this system model that can be seen as a hierarchy of algebras, may and probably should be extended by adding further functional machinery to ease description of the mapping of UML constructs to the system model. However, we wanted to keep the system model as simple as possible and therefore did not concentrate on this additional machinery very much. However, “users” of the system model are invited to add whatever they feel appropriate.

There are also a number of loopholes that can be further investigated by providing additional machinery to clarify a mapping of UML concepts to the system model further.

For example, one constraint on timed state transition systems defined above enforces that messages received cannot be reacted directly upon. This makes sense both from a foundational view as from a view of the reactive systems. This nevertheless enforces a delay between input and reaction to the input in different transitions. As a consequence, either the input or the output needs to be temporarily stored in the state of the TSTS. Therefore, either we store the output at least one step (or longer) or the input at least one step, before we can react. We do not have the storage for outputs available in the state of the object. An extra component that would act as bus and thus enable us to describe message delay is equally awkward, as it would blur our composition concept, both on TSTS and on I/O-behaviours. Therefore, the remaining possibilities are: (a) The input store can be split into "processable input" and "recent input", where each arriving tick moves the latter to the first. (b) We allow the start of a new message processing only when a time tick occurs. In this approach each tick is starter for a scheduling round that decides, which message to handle next (even in the pure sequential case). (c) As a refinement of (b), we could use time ticks in a so fine grained manner that only one "atomic action" per object and time tick is possible. This works nicely, though it also produces a lot of fine grained steps in the state machines.

A number of higher-level concepts could be added to the system model more or less directly. As we have demonstrated with associations, which are manifested as retrieval functions on the object store, we might add a basic set of actions and activities or components thereof like the "pins" of the activity diagrams, features like in [KPR97], or workflow elements as in [RT98].

Acknowledgements: We would like to thank Gregor v. Bochmann, Alain Faivre, Christophe Gaston, Sébastien Gérard and Hans Grönninger for their very helpful comments.

5. References

[BCR06] Manfred Broy, María Victoria Cengarle, Bernhard Rumpe. Semantics of UML. Towards a System Model for UML. The Structural Data Model, Version 1.0. Technical Report TUM-I0612, Institut für Informatik, Technische Universität München, 2006.

[BCR07] Manfred Broy, Maria Victoria Cengarle, Bernhard Rumpe. Semantics of UML. Towards a System Model for UML. The Control Model, Version 1.0. Technical Report TUM-I0710, Institut für Informatik, Technische Universität München, 2007.

[BG92] Gérard Berry, Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87—152. Elsevier North-Holland, 1992.

[BS01] Manfred Broy, Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[EHS97] Jan Ellsberger, Dieter Hogrefe, Amardeo Sarma. *SDL: Formal Object-Oriented Language for Communication Systems*. Prentice Hall, 1997.

[GR95] Radu Grosu, Bernhard Rumpe. *Concurrent Timed Port Automata*. Technical Report TUM-I9533, Technische Universität München, 1995.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. Communications of the ACM, 21(8):666—677. ACM Press, 1978.

[HR04] David Harel, Bernhard Rumpe. Meaningful Modeling: What's the Semantics of „Semantics"? In: Computer, Volume 37, No. 10, pp 64-72. IEEE, October 2004.

[Kah74] Gilles Kahn. The Semantics of a simple language for Parallel Programming. In *IFIP Congress'74 (Proceedings)*, pages 471—475. North Holland Publishing, 1974.

[Kat93] Randy H. Katz. *Contemporary Logic Design*. Addison Wesley Publishing Company, 1993.

[KPR97] Cornel Klein, Christian Prehofer, Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In: Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems. Ed.: P. Dini. IOS-Press. 1997.

[Mil80] Robin Milner. A Calculus of Communicating Systems. In vol. 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Rum96] Bernhard Rumpe. Formale Methodik des Entwurfs verteilter objektorientierter Systeme. PhD Thesis. TU München. 1996.

[RT98] Bernhard Rumpe, Veronika Thurner. Refining Business Processes. In: Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications). Eds.: H. Kilov, B. Rumpe, I. Simmonds. Technical report TUM-I9820 TU München. 1998.

6. Appendix: Central Model of Interaction: Streams and Stream Processing Components

This section describes a closed, well-defined theory on stream processing components and state transition systems that define two different views on distributed, interacting systems. Both, streams processing components and state transition systems are well related through appropriate mapping functions and their refinement and composition techniques are compatible.

6.1 Types of Models for Interactive Systems

There are a number of different theories and fundamental models of interactive systems. Most significant for them are their paradigms of interaction and

composition. We identify three basic concepts of communication in distributed systems that interact by message exchange:

- *Asynchronous communication* (message asynchrony): a message is sent as soon as the sender is ready, independent of the fact whether a receiver is ready to receive it or not. Sent messages are buffered (by the communication mechanism) and can be accepted by the receiver at any later time; if a receiver wants to receive a message but no message was sent it has to wait. However, senders never have to wait (see [Kah74,EHS97]) until receivers are ready since messages may be buffered.
- *Synchronous communication* (message synchrony, rendezvous, handshake communication): a message can be sent only if both the sender and the receiver are simultaneously ready to communicate; if only one of them (receiver or sender) is ready for communication, it has to wait until a communication partner gets ready (see [Hoa78,Mil80]).
- *Time synchronous communication* (perfect synchrony): several interaction steps (signals or atomic events) are conceptually gathered into one time slot; in this way, systems are modelled with the help of sequences of sets of events (see [BG92] as a well-known example).
- *Traditional method call:* It gathers some characteristics of all three previously described approaches. In sequential method calls, progress of time is not such a big issue, which allows programmers to think of synchronous message passing and even of perfect synchrony. The receiver, however, cannot prevent the sender (caller) to start the method call. The receiver must accept the call and react somehow.

Any of the first three models can be used to define each other, and method calls can be simulated in all of them. Furthermore, a general purpose modelling language, like the UML, attempts to provide mechanisms for all of these communication paradigms. Moreover, these communication paradigms shall be used within one system and work together. It is therefore necessary to integrate all communication paradigms, e. g. by encoding one within the other. It is a matter of taste to choose one of these paradigms as underlying mechanism. In the following, we work with asynchronous message passing since this model has, according to our experience, the finest properties for our purpose. We follow the system model given in [BS01] basing our approach on a concept of a component that communicates messages asynchronously with its environment via named channels within a synchronous time frame.

6.2 Streams

For a convenient specification of object behaviour, it is of interest to look not only at the currently incoming message, but at the overall sequence of messages that has arrived on a channel so far. We thus use channel histories to model traces of behaviour.

A *stream* is a finite or infinite sequence of elements of a given set to describe object behaviour. In interactive systems streams are built over sets of messages or actions. A stream describes an observation that an observer can make when sitting on a

directed communication channel. The behaviour of a component can then be modelled through a relation between its observed input and output streams. Streams are therefore used to represent interaction patterns by communication histories for channels or histories of activities.

Let M be a set (of messages). By M^* we denote the set of finite sequences of elements of M, and by M^∞ the set of infinite sequences of elements of M. The set M^ω of streams over M are finite or infinite sequences of elements of the set M. Thus $M^\omega = M^* \cup M^\infty$. If desired, streams over M can be understood as partial functions of form $x : [1..n] \to M$ with "length" $n \in \mathbb{N} \cup \{\infty\}$, where infinite streams are exactly the total functions $(\mathbb{N} \to M)$. We write x.t instead of x(t) as shorthand for selection of the element in x at position t. A finite stream x of elements x.1, …, x.n (in this ordering) is also written $\langle x.1,\ldots,x.n \rangle$. A special case is the empty stream, denoted by $\langle\rangle$. The set of streams has an adequate set of mathematical operations, forming a rich algebraic and topological structure. We introduce concatenation as an operator

$$\hat{} : M^\omega \times M^\omega \to M^\omega$$

On finite streams concatenation is defined as usual: given $x, y \in M^*$ of length n and m, respectively:

$$\langle x.1,\ldots,x.n \rangle \,\hat{}\, \langle y.1, \ldots, y.m \rangle = \langle x.1,\ldots,x.n, y.1,\ldots,y.m \rangle$$

For an infinite stream $s: \mathbb{N} \to M$ we define concatenation as follows:

$$s\hat{}x = s \qquad \text{for any stream } x \in M^\omega$$

be x finite or infinite, and

$$(\langle x.1,\ldots, x.n \rangle \,\hat{}\, s).t = \begin{cases} x.t & \text{if } t \le n \\ s.(t-n) & \text{otherwise} \end{cases}$$

By means of concatenation we define the binary prefix relation $\sqsubseteq$ on streams: Let $s, r \in M^\omega$

$$s \sqsubseteq r \Leftrightarrow_{def} \exists\, z \in M^\omega: s\hat{}z = r$$

$(M^\omega, \sqsubseteq)$ is a partial order, i. e., the relation $\sqsubseteq$ is reflexive, transitive and antisymmetric. This partial order is moreover well founded (i. e., it contains no countable infinite descending chains[4]) and complete (it has a least element, namely $\langle\rangle$, and each of its chains has a least upper bound). This property is very useful, as it allows the description of finite prefixes of streams and the use of inductive (or recursive) techniques for full stream characterisation. For example consider this stream: $s = \langle 1,0,0 \rangle \,\hat{}\, s$.

---

[4] A chain is a subset of M^ω which is totally ordered wrt. $\sqsubseteq$, i. e., of the form $\{ x_t \in M^\omega: t \in \mathbb{N} \wedge x_t \sqsubseteq x_{t+1} \}$.

Definition of streams (timed and untimed)

- M^{ω} is the set of finite or infinite untimed streams
- x^y is the concatenation of two streams
- $\langle x.1, \ldots, x.n \rangle$ denotes the finite stream of length n
- $x \sqsubseteq y$ denotes the prefix relation on streams (read "x is a prefix of y")
- $(M^{*})^{\infty}$ is the set of infinite timed streams

A stream represents the sequence of messages sent over a channel during the lifetime of a system. Of course, in concrete systems this communication takes place in a time frame. Hence, it is often convenient to be able to refer to this time. Moreover, as we will see, the theory of feedback gets much simpler. Therefore we work with *timed streams*.

Streams are used to represent histories of communications of data messages transmitted within a time frame. Given a message set M, we define a *(infinite) timed stream* as the elements in the set $(M^{*})^{\infty}$, or, equivalently, as the functions of form

$$s: \mathbb{N} \rightarrow M^{*}$$

For each time unit t, the finite stream s.t denotes the sequence of messages observed in the stream s at time slot t.[5] That is, a timed stream $s \in (M^{*})^{\infty}$ expresses which messages are transmitted at which time(s).

Throughout this paper we work with a couple of simple basic operators and notations for streams and timed streams, respectively. Some of these operators and notational conventions were already introduced above. They are summarised below. (Note the overloading on timed and untimed streams.)

[5] Or the sequence of actions executed in the t-th time interval, if the stream represents a sequence of actions.

Definition of operations on streams (timed and untimed)

The following operations exist on the algebras of streams:

- $\langle\rangle$ empty sequence or empty untimed stream,
- $\langle m \rangle$ one-element sequence containing m as its only element,
- x.t t-th element of the (timed or untimed) stream x,
- #x length of the untimed stream x including the special value ∞ for an infinite length,
- x^z concatenation of the finite stream x to the untimed stream z,
- $x{\downarrow}t$ prefix of length t of the (timed or untimed) stream x,
- S©x (timed or untimed) stream obtained from x by deleting all its messages that are not elements of the set S,
- S#x number of messages in the (timed or untimed) stream x that are elements of the set S,
- $\bar{x}$ (finite or infinite) untimed stream that is the result of concatenating all sequences in the timed stream x. Note that $\bar{x}$ is finite iff x carries only a finite number of nonempty sequences.

6.3 Channels and Histories

We use timed streams to model the communication histories of sequential, unidirectional communication media (e. g. between two objects) that we call *channels*. A system usually has a larger number of these communication streams. Therefore, we work with channels to refer to individual communication streams. Accordingly, a channel is simply an identifier (channel name) which is associated with a stream observation in every execution of the system.

A channel is thus modelled by an element of the universe of channel names (UCN). Each channel is given a “type” of messages that flow on the channel. The concept of a stream is then used to define the concept of a channel history. A channel history is given by the messages communicated over a channel. Such a history describes an observation on a channel, when recording the flow of messages during time.

Definition of channels and channel histories

- UCN denotes the universe of channel names
- UMESSAGE
  denotes the universe of messages
- type(c) $\subseteq$ UMESSAGE
  denotes the messages on channel c$\in$UCN
- x : UCN $\rightarrow$ (UMESSAGE*)$^{\infty}$
  denotes a *channel history*: it is a partial function that assigns a stream to its channels. For all c$\in$UCN with a defined value x.c:
  - x.c $\in$ (type(c)*)$^{\infty}$.
- channels(x) $\subseteq$ UCN
  is the domain of channel history x
- x.c $\in$ (type(c)*)$^{\infty}$
  denotes stream x(c) for c$\in$channels(x)
- $\mathbb{H}$(C) = $\vec{C}$ = { x : UCN $\rightarrow$ (UMESSAGE*)$^{\infty}$ | channels(x) = C }
  denote the set of channel histories for the channel set C$\subseteq$UCN
- (z$\oplus$z') denotes the *direct sum* of the histories z $\in$ $\mathbb{H}$(C) and z' $\in$ $\mathbb{H}$(C'), iff they are disjoint (C $\cap$ C' = $\emptyset$). It holds:
  - channels(z$\oplus$z') = channels(z) $\cup$ channels(z') = C $\cup$ C'
  - (z$\oplus$z').c = z.c for c $\in$ C
  - (z$\oplus$z').c = z'.c for c $\in$ C'
- z|C denotes the restriction of the mapping z on C$\subseteq$channels(z). It holds:
  - channels(z|C) = C
  - (z$\oplus$z')|channels(z) = z

All operations and notational conventions introduced for streams generalise in a straightforward way to histories applying them element-wise. As we deal with piecewise composed behaviour, we extend our notion of channels histories to partial histories of the form x: UCN $\rightarrow$ (UMESSAGE*)* and to time slices of the form u: UCN $\rightarrow$ UMESSAGE*. The latter is explicitly given by the following definitions (where T is used in place of $\mathbb{H}$):

Definition of channels time slices

- $u : UCN \rightarrow UMESSAGE^*$
  denotes a *channel time slice*: it is a partial function that assigns an untimed, finite stream to its channels. For all $c \in UCN$ with a defined value u.c:
  - $u.c \in type(c)^*$.
- $channels(u) \subseteq UCN$
  is the domain of channel time slice u
- $u.c \in type(c)^*$
  denotes stream u(c) for $c \in channels(u)$
- $T(C) = \{ u : UCN \rightarrow UMESSAGE^* \mid channels(u) = C \}$
  denote the set of channel time slices for the channel set $C \subseteq UCN$

The notion of a stream is essential for defining the behaviour of components in the following section.

6.4 Interfaces, I/O-Behaviours, Time, and Causality

In this section we introduce a theory of component behaviours and interface abstraction. Then we discuss issues of time and causality.

6.4.1 Interfaces

We start with a signature view on components in terms of syntactic interfaces and continue with a behavioural view.

Definition of syntactic component interface

- $(I \blacktriangleright O)$ denotes the *syntactic interface* of a component, where
- $I \subseteq UCN$ is a set of typed channels called the *input* and
- $O \subseteq UCN$ is a set of typed channels called the *output*.

The syntactic interface does not say much about the behaviour of a component. Basically it only fixes the basic steps of information exchange possible for the component and its environment.

Definition of behavioural component interface

Given the *syntactic interface* (I ▶ O) of a component, the function

- $F : \vec{I} \rightarrow \wp(\vec{O})$ describes the component behaviour.

F.x describes the output histories that may be returned for any input history $x \in \vec{I}$; the set F.x can be empty.

This definition basically introduces a relation between input and output histories. We do not distinguish semantically so far between input and output. In the next section we introduce the notion of causality as an essential semantic differentiation between input and output.

6.4.2 Causality

For input/output information processing devices there is a crucial dependency of output on input. Certain output messages depend on certain input messages. A crucial notion for interactive systems is therefore *causality*. Causality indicates dependencies between the messages exchanged within a system. It describes, which output message is a reaction on which input.

So far, I/O-behaviours are nothing but relations represented by set-valued functions. In the following we introduce and discuss the notion of causality for I/O-behaviours.

I/O-behaviours generate their output and consume their input in a time frame. This time frame is useful to characterise causality between input and output. Output that depends causally on certain input cannot be generated before this input has been received.

Definition of causality

- An I/O-behaviour $F: \vec{I} \rightarrow \wp(\vec{O})$ is *causal* (or *properly timed)* if, for all times $t \in \mathbb{N}$, we have
    - $x{\downarrow}t = z{\downarrow}t \Rightarrow (F.x){\downarrow}t = (F.z){\downarrow}t$.

A function F is causal if the output in the t-th time interval does not depend on input that is received after time t. This ensures that there is a proper time flow for the component modelled by F. F cannot predict the future input and react on it.

If F were not causal, there would exist a time t and input histories x and x' such that $x{\downarrow}t = x'{\downarrow}t$ and $(F.x){\downarrow}t \neq (F.x'){\downarrow}t$. A difference between x and x' occurs only after time t, but at time t the reactions of F in terms of output messages are already different. Thus F could predict the future.

Nevertheless, causality permits instantaneous reaction [BG92]: the output at time t may depend on the input at time t. This may lead into problems with causality between input and output, if we consider in addition delay free feedback loops known as causal loops. To avoid these problems we either have to introduce a sophisticated theory to deal with such causal loops for instance by least fixpoints in

an appropriate domain theory, or we strengthen the concept of proper time flow to the notion of strong causality.

Definition of strong causality

- An I/O-behaviour F: $\vec{I} \rightarrow \wp(\vec{O})$ is *strongly causal* (or *time guarded)* if, for all times t ∈ IN, we have
  - $x{\downarrow}t = z{\downarrow}t \Rightarrow (F.x){\downarrow}t+1 = (F.z){\downarrow}t+1$.

Strong causality simply enforces components to introduce a delay of one time unit before it can react. If the granularity of time units is fine enough, we can always detect such a delay.

In general, an I/O-behaviour F: $\vec{I} \rightarrow \wp(\vec{O})$ allows many implementations, as it allows many reactions to one input. Each one of the possible implementations can be described as a deterministic descendant of this behaviour. Such an implementation is given through a deterministic function f: $\vec{I} \rightarrow \vec{O}$.

Definition of deterministic implementation

- A function f: $\vec{I} \rightarrow \vec{O}$ is a deterministic I/O-behaviour F: $\vec{I} \rightarrow \wp(\vec{O})$, where F.x = {f.x} for all x ∈ $\vec{I}$.
- A function f: $\vec{I} \rightarrow \vec{O}$ is a deterministic implementation of the I/O-behaviour F: $\vec{I} \rightarrow \wp(\vec{O})$, iff f.x ∈ F.x for all x ∈ $\vec{I}$.
- F is called realisable, if there is at least one deterministic implementation.

6.5 Composition of Interface Behaviour

In this section, we introduce an operator for the *composition* of components. We prefer to introduce only one very general powerful composition operator. This operator generalises sequential and parallel composition as well as introduction of feedback loops.

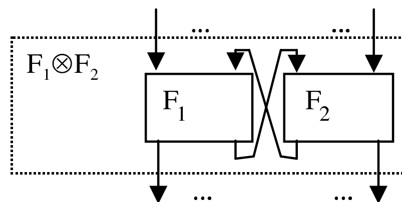

Fig 2 Parallel Composition with Feedback

Definition of composition

- Composition of two I/O-behaviours F_1 , F_2 of components is denoted by
  $F_1 \otimes F_2$
- Given the signatures $F_1 : (I_1 \blacktriangleright O_1)$ and $F_2 : (I_2 \blacktriangleright O_2)$ where
  - output channels are disjoint $O_1 \cap O_2 = \emptyset$
- the composition $(F_1 \otimes F_2)$ has signature $(I \blacktriangleright O)$, with
  - $I = (I_1 \cup I_2)\backslash(O_1 \cup O_2)$ and
  - $O = (O_1 \cup O_2)\backslash(I_1 \cup I_2)$.
- And describes a behaviour defined by
  - $(F_1 \otimes F_2).x = \{ y|O : y|I = x|I \wedge y|O_1 \in F_1(y|I_1) \wedge y|O_2 \in F_2(y|I_2), y \in \mathbb{H}(I \cup O) \}$

Here, y denotes a valuation for all the internal, input and output channels in the composition. y|C denotes the restriction of the valuation y to the channels in C. The composition formula essentially says that all the streams on output channels of the components F_1 and F_2 are feasible output streams of these components.

It is straightforward to prove that the composition is strongly causal, if one component is strongly causal. If both components are deterministic, then so is the composition. If both components are realisable, then so is the composition.

But as the most important result, the composition is designed in such a way, that it is compatible with independent development of its parts. This means, given a composition, we can chose a deterministic implementation for each part individually, compose these and get a deterministic implementation of the composition.

And finally, the composition is associative and commutative, which allows us to generalise the composition operation to any (signature compatible) set of components – including infinite sets.