Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets

Christoph Eichner, Hans Fleischhack, Roland Meyer, Ulrik Schrimpf, and Christian Stehno

Parallel Systems Group,
Department for Computing Science,
Carl von Ossietzky Universität,
D-26111 Oldenburg, Germany
`forename.surname@informatik.uni-oldenburg.de`

Abstract. With the introduction of UML 2.0, many improvements to diagrams have been incorporated into the language. Some of the major changes were applied to sequence diagrams, which were enhanced with most of the concepts from ITU-T's Message Sequence Charts, and more. In this paper, we introduce a formal semantics for most concepts of sequence diagrams by means of Petri nets as a formal model. Thus, we are able to express the partially ordered and concurrent behaviour of the diagrams natively within the model. Moreover, the use of coloured high-level Petri nets allows a comprehensive and efficient structure for data types and control elements. The proposed semantics is defined compositionally, based on basic Petri net composition operations.

1 Introduction

The long-standing and successfully applied modelling technique of Message Sequence Charts (MSC) [11] of ITU-T has finally found its way to the most widely applied software modelling framework, the Unified Modelling Language (UML) [18]. In its recent 2.0 version, sequence diagrams (SD, interaction diagram) were enhanced by important control flow features. This change is one of the major differences between UML 1.x and UML 2.0 for interaction diagrams. Most importantly, sequence diagrams may now completely specify a system's behaviour, while former UML versions only provided support for describing exemplary execution sequences.

We describe a Petri net based semantics of most elements of sequence diagrams. The semantics is built compositionally according to the structure of the diagrams. Due to the use of Petri nets as a basic model, the inherent partially ordered structure of sequence diagrams can be captured explicitly in the semantics.

In this paper, time inscriptions in sequence diagrams are not handled by the semantics, neither are object-oriented data structures. However, we define a data access model and show its general capabilities. In an extended version of the semantics we will add support for time features. Additionally, we will equip activity diagrams and interaction overview diagrams with semantics based on the same underlying model to also express object internal behaviour and relationships between different diagrams.

Although verification of the gained semantics would be possible with standard Petri net tools (such as the PEP tool, being developed in the authors' group), the main aspect of the current development is to capture behaviour, and simulate and visualize it. In addition to the translation described in this paper, we are developing an animation environment within the P-UMLaut project [19] capable of linking together 3D objects of the simulated world and entities in the UML model. The 3D world will be animated driven by the Petri net simulation of the semantics.

2 Related Work

Semantics for SDs and MSCs have been investigated for quite a long time. Since UML did not provide a properly defined semantics, most of the early papers on semantics dealt with MSCs.

For the basic UML concepts, the reference is OMG's standard [18], but also a number of good books exists for UML 2.0 (such as [8, 12]). A discussion of the major elements from interaction diagrams can be found in [23].

A thorough examination of MSC-96 regarding all details can be found in [22]. The thesis covers all MSC elements and presents a formal operational semantics. Another approach to define a semantics for MSCs uses pomsets [14].

Due to the inherent non-interleaving semantics of MSCs and SDs, a number of authors chose Petri nets to describe MSC semantics (such as [2, 10, 9, 15]). Most of the papers, however, deal with communication structures only, and abstract from data types to allow verification algorithms to be applied. Due to these restrictions, a number of elements of MSCs become useless and are thus not examined.

A major extension of both the concept of MSCs as well as the concept of SDs has been given as Life Sequence Charts (LSC) in [4]. LSCs introduce new concepts to combine different diagrams and additional elements. The goal is to allow a complete specification of the system's behaviour as well as of system properties within the same model. Although some features of LSCs have been integrated into MSCs and SDs, many aspects of a system can be described much more comprehensively and precisely by using LSCs.

Although object-oriented features are not handled by our semantics, there are solutions for expressing object-orientation by means of Petri nets (see [1, 16]). An integration of some of these concepts into the data handling proposed in this paper will be part of further research.

3 M-Nets – An Algebra of High Level Petri Nets

Petri nets are an easy and intuitively comprehensible visual model with a strong mathematical foundation. Thus, Petri nets, and especially high level Petri nets, have been established as a commonly used model on all levels of formal software engineering and verification (for example see [13, 17, 21, 20]).

The semantics given in this paper is based on the algebra of M-nets (multi-valued nets). The benefits of using M-nets rely on the fact that any M-net is constructed from very simple nets by application of a set of well defined operators. As a consequence, modelling semantics and verification of the semantic models is comparably simple when using M-nets.

An M-net consists of a high level Petri net with the usual inscriptions governing colored token flow and additional inscriptions governing composition and synchronization of nets.

Places are classified as *entry* (no incoming arcs; labeled by $+$), *exit* (no outgoing arcs; labeled by $-$) or *internal* (no restrictions with respect to arcs; no additional label). The composition operators include sequential, choice, and parallel composition and an iteration operator. For example the sequential composition of two nets N_1 and N_2 consists of their juxtaposition, where all exit places of N_1 are combined with all entry places of N_2 using a cross product operation.

Transitions of M-nets are inscribed with a set of (parameterized) synchronous action labels and a set of (parameterized) asynchronous action labels. Synchronous action labels are used for (CCS-like) synchronization and restriction. The combined operation of synchronization followed by restriction is called *scoping* (**sc**).

The asynchronous action labels are subject to the **tie** operator, which links two transitions t_1 and t_2 by inserting a place between both, making occurrence of t_2 dependent on occurrence of t_1.

Handling of a data variable is modeled by a *data box* within the algebra of M-nets. Essentially, a data box consists of a place p containing the actual value of the variable, and a transition t for communicating the actual value or storing a new value. Occurrence of t is usually governed by transitions of the control part of an M-net via synchronization with t.

The scoping and tie operations build up the correct links between data accessing transitions and data boxes and send and receive events, respectively. The effect of applying the **tie** operator can be seen in fig. 1. The data access to variables x and y would be resolved by synchronization over actions $X/2$ and $Y/2$. This would replace a with the current value of x, thus storing a in y while discarding the old value b of y. Since application of these operators is purely

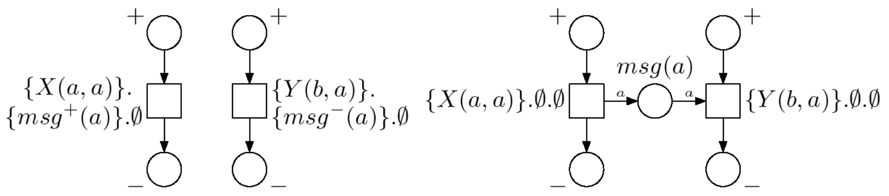

Fig. 1. Message creation with **tie**

technical, we will show throughout the paper only nets where **sc** is omitted and **tie** has already been applied.

For the translation of complete interaction fragments (see section 4.2) the algebra is extended by the new composition operator of *net concatenation*, defined as follows:

Definition 1 (Net Concatenation). *Let $N_3 = N_1 \circ N_2$ be the* concatenation *of two M-nets $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$. Let ${}^-P$ be a subset of the exit places of N_1 and ${}^+P$ be a subset of the entry places of N_2. The resulting net is defined by $N_3 = (P_3, T_3, F_3)$ with*

$$P_3 = P_1 \cup P_2 \setminus {}^+P$$
$$T_3 = T_1 \cup T_2$$
$$F_3 = F_1 \cup F_2 \setminus \{(p,t) \in F_2 | p \in {}^+P\} \cup \{(p,t) | p \in {}^-P, (h(p),t) \in F_2\}$$

$h : {}^-P \rightarrow {}^+P$ is a bijective mapping on the subsets to be connected that may be arbitrarily chosen.

A description of M-nets theory with applications to modelling and verification can be found in references such as [3, 5, 7].[1]

4 Translation

In this chapter a mapping $PN : SD \rightarrow M-nets$ of UML 2.0 sequence diagrams to M-nets is defined. This paper does not define the semantics of time events (which are not translated in this approach due to a time-free semantical model used), some combined fragment types (which are not applicable due to the different point of view on system specification), or part decomposition. Although part decomposition offers a number of useful features for comprehensible models, and most of these features should be translatable to Petri nets, extra global fragments are intrinsically non-compositional, and thus not implementable in our semantics. An extension of the proposed semantics with non-compositional features including extra global fragments will be part of future work.

4.1 Data Types

Since the semantics given in this paper shall describe the general behaviour of systems, we do not restrict ourselves to interactions describing exemplary execution traces. Thus, parameters given to messages are not necessarily constants, but may refer to attributes and variables found within the scope of the examined interaction.

Handling of data is only sketched in this proposal due to lack of space. However, the semantics for handling data types in communication parameters and

[1] At http://www.p-umlaut.de/paper/mnets.pdf a brief introduction to M-nets can be found.

within conditions and invariants is described in such a way that it depends only on a proper mathematical definition of the data domain to incorporate data types other than the proposed boolean and integers.

According to the restricted data types, the syntax of conditions used in the diagrams is defined inductively by

$$cond ::= b_1 \mid b_1 = b_2 \mid x_1 = x_2 \mid \neg cond_1 \mid cond_1 \wedge cond_2 \tag{1}$$

where b_1, b_2 are variables or values of type **bool**, x_1, x_2 are integer variables or values, and $cond_1, cond_2$ are conditions. Conditions are translated to transition guards, which ensure that firing of a transition is only possible if the condition is satisfied.

The behaviour of method calls in SDs is very similar to those in common programming languages. Each actual parameter of a method defines a variable local to the receiving instance. Such variables may be undefined and will cease to exist after the method has returned. Return values may be stored in previously defined variables and attributes of the caller's scope with UML's own shorthand definition `varname = methodname(parA, parB):RetValue`. Here, the variable `varname` is set to the explicitly given value `RetValue`, which may be a constant or an existing variable.

Thus, parameters only define the way data access has to be scoped and the way values have to be copied to different scopes. A semantics for this context has already been defined in [6, 7]. The idea is to define data boxes with a predefined scope, which handle all access to a distinct variable. While this concept does not provide for object-oriented relations like inheritance, all necessary features, such as instantiation and removal of variables, scoping, call by reference, and undefined values, are covered.

4.2 Compositional Construction

The semantics of an interaction diagram is built bottom-up: composition starts from innermost elements, and incrementally adds surrounding elements on each level of nesting.

To define the semantics, we need to define *maximal independent sets.* These sets contain partial lifelines whose elements are completely unordered with respect to all other contained elements that are not part of the same lifeline. Thus, these lifeline fragments do not need special care for sequentialization. The blocks will be later on glued together using a special concatenation operation.

Definition 2 (Maximal Independent Set). *Let $\mathcal{L} = \{L_1, \ldots, L_n\}$ be a set of lifelines. An* independent set *of events ordered by their causal ordering on the lifelines is a set $\mathcal{I} = \{e_{1,1}, \ldots, e_{1,k}, \ldots, e_{n,1}, \ldots, e_{n,k_n}\}$ of events of the different lifelines. The projection of $\mathcal{I}$ to a single lifeline leads to a set of consecutive elements. A* maximal independent set (MIS) *has a combined fragment or the diagram border both at its start and at its end.*

With this definition, we can find a partition of a sequence diagram into maximal independent sets. This partition is unique and groups partial traces with only messages inside, thus separating unordered parts of the diagrams from parts with orderings imposed by combined fragments.

We will define our Petri net semantics based on maximal independent sets. The resulting nets are incrementally glued together with surrounding combined fragments and other blocks until all parts are grouped into one net.

The semantics of each MIS is defined compositionally and separately for each lifeline that is part of the set. Since we only have EventOccurences inside an MIS, we can give the semantics of one lifeline as the sequential composition of elementary boxes, as defined by the semantics of send and receive events in the next section. All lifelines are put in parallel due to their independent behaviour.

Each MIS is defined such that the first and last events either border start or end of lifelines, or such that all events border the same combined fragment. Thus, the semantics of a complete interaction fragment is defined by concatenating the different blocks with respect to causal ordering imposed by lifelines.

Send and receive events as well as data access inside conditions, actions, and message parameters are only handled by inscriptions of the Petri nets, using action symbols and tie symbols of M-nets. The resulting nets have to be completed in a last step to gain explicit representations of the Petri net semantics.

4.3 Elementary Diagram Elements

Interaction Frames. A sequence diagram is denoted by an interaction frame as shown in fig. 2. The variables par_1 to par_m declared in the diagram header are parameters receiving values while instantiating the frame. Local variables are given as $local_1$ to $local_n$ below the header. They are initialized by the *Init* action which occurs at the beginning of an interaction frame, and are destroyed upon *Term* at the end.

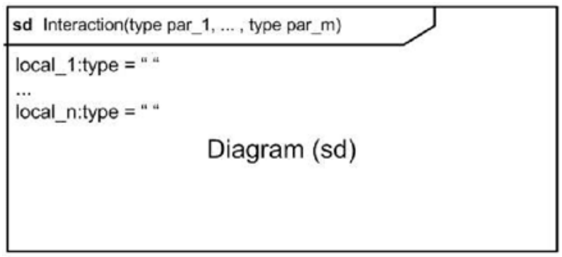

Fig. 2. Interaction frame

Messages are represented in the semantics just by their sending and receiving events. Concrete message representations are created by the **tie** operator when all such events are placed. The semantics of an interaction frame is given by

$$PN(IF) = \left(Init; PN(Diag); Term \parallel \mathop{\parallel}_{1 \le i \le m} DB(par_i) \parallel \mathop{\parallel}_{1 \le j \le n} DB(local_j) \right)$$
$$\textbf{sc}\ act(par_1) \ldots \textbf{sc}\ act(local_n) \textbf{tie} *$$

Lifelines. Lifelines denote the existence of objects during system execution. A lifeline starts with a named or anonymous object of a certain type as represented in fig. 3 (in the example with an additional creation message). Any event of the sequence diagram is connected to one or more lifelines. Lifelines themselves have no explicit semantics, but are represented by their events.

Since lifelines represent objects, object attributes have to be set up at creation time. All lifelines not generated by a creation message are initialized before the first event of each lifeline is handled. To achieve this, an elementary box is created with an initialization action for each attribute of the object. The data boxes for these variables are added to the local variables of the main frame. Initialization is only applied to the outermost interaction frame since diagrams used in reference frames only contain already existing lifelines.

In sequence diagrams new lifelines can be created by others using a *creation message*. The semantics of object creation is similar to that for lifelines starting from the beginning. The only difference is that initialization of attributes occurs when the creation message is received.

Destruction of a lifeline (STOP) is depicted as shown in fig. 4. When a destruction is reached, the corresponding object is deleted. The semantics of stop is given by an elementary box with termination actions for each attribute of the object to destroy the contents of each data box belonging to the object. Destruction of the object itself is implemented by an M-net stop box, which is put in sequence with the data boxes' termination.

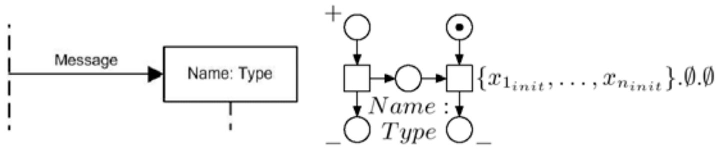
Message
Name: Type
+
$\{x_{1_{init}}, \ldots, x_{n_{init}}\}.\emptyset.\emptyset$
Name :
Type
_
_

Fig. 3. Object creation

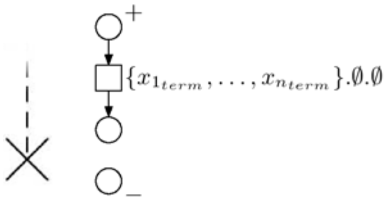
+
$\{x_{1_{term}}, \ldots, x_{n_{term}}\}.\emptyset.\emptyset$
_

Fig. 4. Object destruction

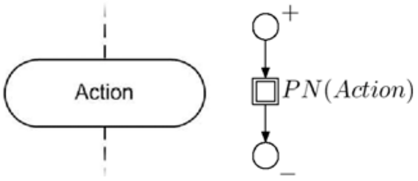

Fig. 5. Action symbol

EventOccurrences and Actions. EventOccurrences and Actions define an active part of the lifeline where (for example) calculations are done. Since these activities are arbitrarily and often not exactly specified, semantics for these elements is only vague. EventOccurrences, denoted as thin rectangles covering the part of the lifeline the activity is running, define a scope. This scope ensures the availability of parameters instantiated by a receive event as part of the EventOccurrence. Thus, local variables instantiated to hold the actual parameters of a message will be destroyed at the end of the EventOccurrence they were received in. EventOccurrences may be nested, such that subscopes may be defined. Scopes are directly related to the **sc** operation of M-nets, which hides all scoped variables. Thus, the semantics of event occurrences consists of creating proper data boxes and applying the **sc** function to all events inside the scope. The instantiation of the data boxes is described in section 4.3.

Action elements describe internal activities of one lifeline at a certain point rather than just the duration. The action need not be specified inside the diagram, but may refer to an Activity Diagram or just use natural language to describe the idea of what is happening inside. Since both alternatives are not translated into Petri nets in this paper, these actions are represented by silent transitions inside the Petri net, i.e. an elementary box without labels as shown in fig. 5. We are, however, working on an Activity diagram semantics which later on can be integrated into the semantics defined in this paper.

The only actions translated to Petri nets in the current semantics are assignments. An assignment action labelled `varname=expression` assigns the variable of the given name the value of the evaluated expression. Therefore, the variable has to be accessible from the current scope and the expression has to evaluate to a value of the data type of that variable. The semantics is defined as an elementary box with appropriate action symbols and the action as guard. For atomic actions at reception of a message, the guard is put to the receiving transition.

Messages. There are five message types in sequence diagrams: *asynchronous messages, synchronous messages, creation messages, lost messages, and found messages.* Messages consist of different parts: the sending event of a message, the sending process (denoted as a named arrow), and the receive event.

The sending event is represented by an elementary box with action symbols for each local variable accessed by the message's parameters. In addition, an export link is added, which will create the message with its actual parameters.

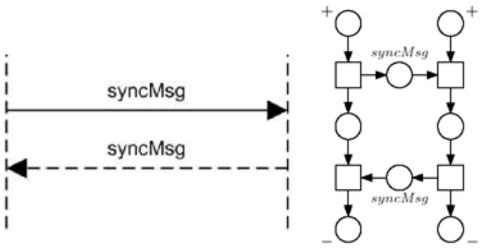

Fig. 6. Synchronous message

Thus, parameters become instantiated, such that only values and not references are sent.

The reception is modelled vice-versa, i.e. the elementary box gets an import link to receive the values of the message place. For parameters, local data boxes are created and initialized by the received values. If the message carries a return value, that value is saved to the local variable given in the assignment of the message.

The semantics of a message before and after applying the **tie** operation is shown in fig. 1. If scoping is also applied, the variables exported by the **tie** operator will be instantiated with actual values from the data boxes.

Synchronous messages, as depicted in fig. 6, consist of two messages. One message initiates the exchange, the second one is the the reply. Control flow in the initiating lifeline is halted until the reply has arrived. Thus, no event is allowed between the send event and its reply. Since this constraint is purely syntactical, the semantics for messages can be applied unchanged.

A *lost message* is a special message without a receive event. Thus, the message does not end on a lifeline. The idea is to have messages without an explicitly specified receiver. The semantics is defined analogously to normal send events. A *found message* is the dual of a lost message. Thus, having matching names, a lost message may be used by a found message's receiving event resulting in a complete message transmission. Since **tie** is applied when all send and receive events have been created, message finding is correctly implemented.

Gates are named points in an interaction frame that may be the source or target of a message. The semantics is equivalent to that of lost and found messages, except that messages are explicitly linked when formal and actual gates are merged during reference replacement.

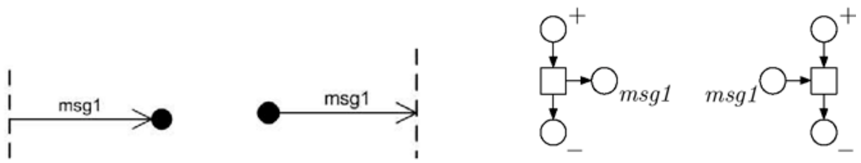

Fig. 7. Lost and found messages

State Invariants and Continuations. *State invariants* are necessary conditions for further progress of a lifeline. They are denoted similarly to actions but labelled with formulae as defined in (1). They are part of only one lifeline, while continuations cover more than one. Continuations, on the other hand, have just a label as inscription.

State invariants check the truth value of their expression. If it is true, the execution may continue, otherwise the lifeline gets stuck, leading finally to improper termination of the interaction diagram. The semantics is defined by an elementary box with the condition as guard and appropriate action symbols to read all necessary variable values.

There exist two kinds of continuations, which always occur pairwise. A *setting continuation* defines a new global boolean variable set to true. The name of the variable is defined by the label of the continuation. All lifelines synchronize on setting the newly created variable. A non-setting continuation is the dual of a setting continuation in that the lifelines synchronize on reading the boolean value created before by a setting continuation. Only if the value has been set to true, may the lifelines proceed with their execution. The semantics of these elements is defined as the generation of a new anonymous message on a globally accessible place named such as the continuation label. This is achieved by an elementary box similar to the one shown in fig. 8. The box exports the boolean variable to global scope instead of checking a condition, is put in sequence with the end of the combined fragment. Non-setting continuations are implemented by an elementary box importing the variable (that is receiving a message) placed in sequence at the the beginning of the combined fragment.

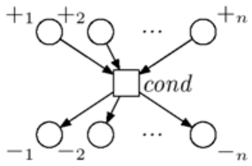

Fig. 8. Delimiting net for combined fragments

4.4 Combined Fragments

With *Combined Fragments*, high level programming constructs may be used in sequence diagrams. Combined fragments are denoted as frames with an operator in the left upper corner and sequence diagram elements inside. The semantics of the diagram depends on the operator. A combined fragment may be built of different operands: that is, separate sequence diagram fragments to which the operator is applied.

Entering a combined fragment, as well as leaving a combined fragment, is considered as an atomic event. In particular, entering a combined fragment has to be done synchronously by all lifelines in the proposed semantics. The rationale for this demand is that otherwise the change of data values might change evaluation of the fragment condition before all lifelines have entered it. Thus, in order to prevent inconsistencies caused by accessing combined fragments only by a subset of all participating lifelines there are two possibilities to interpret combined fragments:

– Either a black box semantics is applied as done in this paper, such that behaviour inside a combined fragment does not interfere with behaviour outside.
– Or the condition has to be evaluated when the first lifeline's execution enters the fragment, and that truth value has to be stored to allow consistent condition evaluation of following lifelines.

The two interpretations seem to respect all constraints of the standard, so the standard might be too imprecise in this case. Both interpretations are however implementable using Petri nets. Due to the artificial introduction of variables and implicit side-effects of the second alternative, we prefer and present the former.

Each combined fragment will thus be prefixed and postfixed sequentially by a net as shown in fig. 8. If a condition is attached to the combined fragment, this condition will be used as the guard of the transition. If more than one operand can be chosen as the first to be executed, a number of such nets might be combined as defined by the operator's semantics. The postfixed transitions do not use guards, as leaving a combined fragment is unconditional.

No semantics are given for *ignore*, *consider*, *neg* and *assert*. Mainly, these types are not considered since they are used for a different application domain of interaction diagrams. The semantics defined in this paper assumes that all behaviour is explicitly specified in the diagrams, such that the mentioned operators do not apply.

Alternative, Optional, and Break Fragments. An *alternative fragment* as given in fig. 9 may have several operands. Each operand has a condition as defined in (1). A special condition `else` can be used as a shorthand for a default operand. If a condition evaluates to true, the diagram fragment is executed and the alternative fragment has finished. If no condition is satisfied the `else` operand is processed. The semantics of an alternative fragment is defined by a choice operation over the semantics of each operand. For the alternative fragment, each operand net itself is prefixed and postfixed by a combined fragment delimiter net as shown in fig. 8. Since the `else` condition is not usable as-is in M-nets, this special condition has to be modelled using a negated disjunction of all other conditions.

In an *optional fragment* a condition indicates whether the (one and only) operand needs to be executed. Optional fragments are equivalent to an alternative fragment with empty `else` operand.

A *break fragment* tests a condition and, if necessary, processes the elements of the break fragment which in turn ends the interaction diagram. If the condition is not satisfied, the rest of the sequence diagram is executed. The semantics is equivalent to that of an alternative fragment with the contents of the break

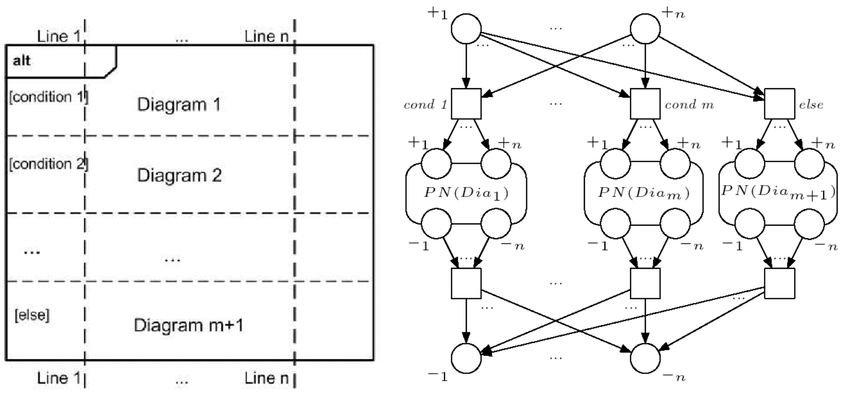

Fig. 9. Alternative fragment

fragment as one operand and all remaining elements of the diagram as `else` branch.

Loop Fragments. A *loop fragment* is given by a combined fragment with two integral parameters $0 \leq minint \leq maxint \leq \infty$ as shown in fig. 10. The diagram in the fragment is processed at least *minint* times, at most *maxint* times, and is optional as soon as *minint* is reached.

The semantics consists of an iteration part that may be executed several times, and a skip transition to bypass the loop initially. The bypass can be used if either the condition of the loop is not satisfied, or *minint* is zero such that the loop need not be executed.

To control the number of iterations, a token on an additional place named *Counter* is incremented in each execution of the loop body. Two transitions in conflict ensure that the loop is executed at most *maxint* times and is optional as soon as *minint* is reached. The loop body is inserted by transition refinement of the *loop* transition.

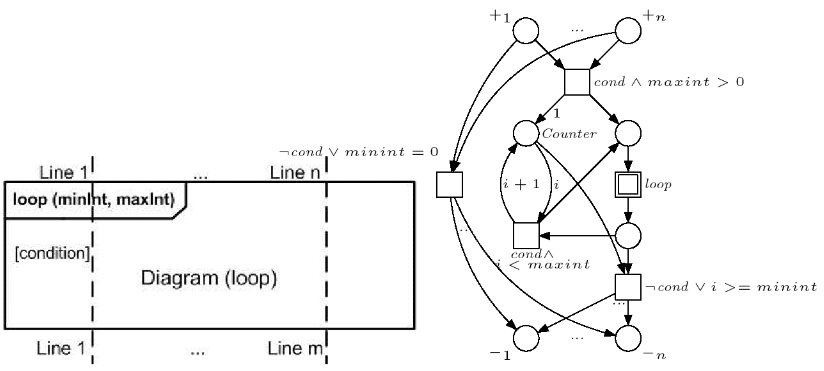

Fig. 10. Loop fragment

Parallel Fragments. Concurrent execution of operands of *parallel fragments* is modelled as shown in fig. 11. The semantics is given by the semantics of the operands composed by the parallel operator, which are sequentially prefixed by the entry transition with empty guard. *Coregions* are equivalent to a parallel fragment covering just one lifeline with each event being its own operand.

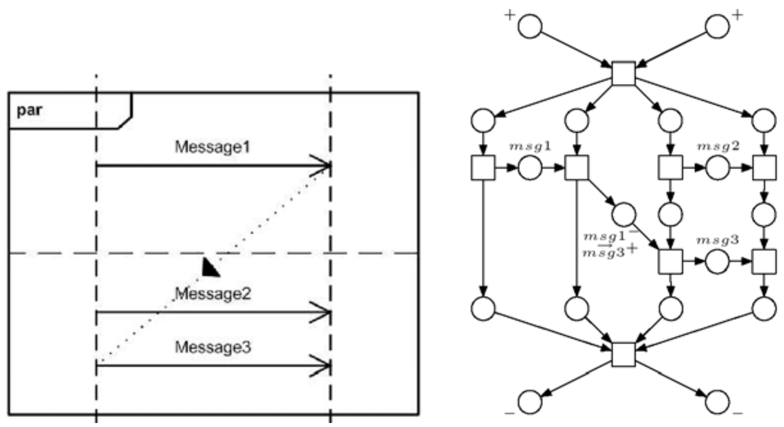

Fig. 11. Parallel fragment with general ordering

General Orderings. In parallel fragments some actions in different operands may be related, and thus need to be executed in a certain order. This is expressed by a dashed line with an arrow in the middle between the actions, called *general ordering*. The semantics is defined by interpreting the general ordering as a special kind of message between two linked events. This message makes the ordering explicit in that additional preconditions and postconditions are installed.

Sequencing. A *strict sequencing fragment* is given by a combined fragment where a strict execution order is imposed by the position of the events across all lifelines.

The semantics of a strict fragment is thus equivalent to a normal interaction diagram with general ordering applied to all events that are not explicitly ordered by messages or their lifeline. We will therefore not define a special semantics but use some preprocessing to add the general ordering.

Operands in strict fragments do not make sense since the ordering would delay the next operand until all events in the current operand are executed. Thus, removing the operand intersections does not change the semantics.

A *weak sequencing fragment* ensures a strict order inside the operands. When all actions in one operand for a lifeline have been processed, actions in the next operand for that lifeline may continue. The semantics of a weak sequencing fragment is thus defined as for strict sequencing, per operand. The different operands are then concatenated in order to execute some events concurrently, if no causal ordering is imposed.

Thus, weak sequencing defines local causality inside operands of a combined fragment, and strict sequencing defines global causality inside the combined fragment. Both operators apply only to the current nesting level, i.e. an ordering is not promoted to nested combined fragments.

Critical Sections. Some fragments of parallel executions may be executed atomically, meaning that other events of the participating lifelines are processed concurrently. This behaviour is captured in *critical sections* inside an operand of a parallel fragment.

In order to define the semantics for a critical section, we have to establish means to prohibit execution on parallel lifelines. This is achieved by an additional place for each concurrent instance of a covered lifeline, that has to be checked for an existing token each time an event of the lifeline is executed. The critical section would then collect all such tokens on entering the fragment, thus stopping all concurrent executions on the monitored lifelines. After the critical section is finished, the tokens are put back and the concurrent instances of the lifelines may continue.

Since each lifeline operates on its own "active" token, concurrency is not limited by this semantics. However, a compositional construction would need to introduce the activity token to every parallel operator, regardless of the presence of a critical section. Without loss of generality and due to enhanced readability, we introduced these concepts only at this point.

Interaction Reference. An *interaction reference* is given by a combined fragment with a name and probably with parameters. Another sequence diagram with that name has to be defined separately, with an identical set of lifelines.

The semantics of the reference is defined by replacing the reference with the actual sequence diagram, and parameters and gates replaced as needed. Since such a replacement can be done statically during compilation, no explicit Petri net semantics is needed. On the other hand, a reuse of the replacing diagram can be done if a procedure semantics is applied. Such a semantics has been defined for M-nets in [6]. Using a procedure semantics would also allow recursion to take place, while the replacement semantics (as proposed by the standard) would require infinite replacement in such cases.

5 Conclusion and Future Work

The semantics defined in this paper covers all elements of Interaction Diagrams of UML 2.0 with the mentioned exceptions. Although defining a semantics based on Time Petri nets should be possible, there has to be some further research on feasibility. The implementation of the proposed semantics has already started. The next step after finishing that implementation will be the extension of the proposed semantics to Activity diagrams and Interaction Overview diagrams.

We would like to thank all other members of the project group P-UMLaut, namely Eike Best, Eike Frost, Martin Hilscher, André Kaiser, Mark Ross, Casjen

Schnars, Tim Strazny, and Harro Wimmel for their support and fruitful discussions on the semantics. Additionally, we would like to thank Alexander Knapp and Cécile Bui Thanh for help on UML, object orientation and semantics, and Hanna Klaudel and Franck Pommereau for their M-nets introduction.

References

1. Gul Agha, Fiorella de Cindio, and Grzegorz Rozenberg, editors. *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*. Springer, 2001.
2. Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML sequence diagrams and statecharts to analysable petri net models. In Simonetta Balsamo, Paola Inverardi, and Bran Selic, editors, *Workshop on Software and Performance '02*, pages 35–45, Rome, Italy, 2002. ACM Press.
3. Eike Best, Wojciech Frączak, Richard P. Hopkins, Hanna Klaudel, and Elisabeth Pelz. M-nets: an Algebra of High-level Petri Nets, with an Application to the Semantics of Concurrent Programming Languages. *Acta Informatica*, 35(10):813–857, 1998.
4. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.
5. Raymond Devillers, Hanna Klaudel, Maciej Koutny, and Franck Pommereau. Asynchronous Box Calculus. *Fundamenta Informaticae*, 54(1):1–50, 2003.
6. Hans Fleischhack and Bernd Grahlmann. A Petri Net Semantics for $B(PN)^2$ with Procedures. In Gul Agha and Stefano Russo, editors, *Parallel and Distributed Software Engineering*, pages 15–27. IEEE Computer Society, May 1997.
7. Hans Fleischhack and Bernd Grahlmann. A Compositional Petri Net Semantics for SDL. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets*, volume 1420 of *Lecture Notes in Computer Science*, pages 144–164. Springer-Verlag, 1998.
8. Martin Fowler. *UML Distilled*. The Addison-Wesley Object Technology Series. Addison-Wesley Longman, 2004.
9. Thomas Gehrke, Michaela Huhn, Arend Rensink, and Heike Wehrheim. An Algebraic Semantics for Message Sequence Charts Documents. In Stanislaw Budkowski, Ana R. Cavalli, and Elie Najm, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV '98)*, pages 3–18. Kluwer Academic Press, 1998.
10. Stefan Heymer. A Semantics for MSC Based on Petri Net Components. In *SAM200*, pages 262–275, Col de Porte, Grenoble, France, 2000. VERIMAG, IRISA, SDL Forum.
11. ITU-T. *Recommendation Z.120 (11/99): Message Sequence Charts* ITU-T, Geneva, 2000.
12. Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, and Stefan Queins. *UML 2 glasklar*. Hanser, 2004.
13. Kurt Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *EATCS Monographs in Computer Science*. Springer, 1992.
14. Joost-Pieter Katoen and Lennard Lambert. Pomsets for Message Sequence Charts. In H. König and P. Langendörfer, editors, *Formale Beschreibungstechniken für verteilte Systeme*, pages 197–207, Cottbus, June 1998. GI/ITG, Shaker Verlag.

15. Olaf Kluge. Modelling a railway crossing with message sequence charts and petri nets. In Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg, and Herbert Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 197–218. Springer, 2003.
16. Johan Lilius. OB(PN)^2: An object based petri net programming notation. In Agha et al. [1], pages 247–275.
17. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
18. Object Management Group. *UML 2.0 Superstructure Specification*, 03-08-02 edition, August 2003.
19. Project P-UMLaut. http://www.p-umlaut.de.
20. Lutz Priese and Harro Wimmel. *Theoretische Informatik: Petri-Netze*. Springer-Verlag, 2002.
21. Wolfgang Reisig. *Petri nets – An introduction*. Springer, 1985.
22. Michel Adriaan Reniers. *Message Sequence Charts*. PhD thesis, Eindhoven University of Technology, 1999.
23. Harald Störrle. Semantics of Interactions in UML 2.0. In *2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 129–136, Auckland, New Zealand, October 2003. IEEE Computer Society.