

Semantics of Exceptions in UML 2.0 Activities

Harald Störrle

Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München, GERMANY
stoerrle@informatik.uni-muenchen.de

Abstract

The recent major revision of the UML [22] has introduced significant changes and additions to “the lingua franca of Software Engineering”. One of the most interesting innovations is exception handling. Building on [29, 34], this paper explores the meaning of the exception-related constructs by defining a mapping to a semantic domain that is a slight extension of procedural Petri nets [20].

Keywords: UML 2.0, Activity Diagrams, exception handling, procedural Petri-nets, modeling of web-services, workflows, and service-oriented architectures

1 Introduction

1.1 Motivation

The modeling of business processes and workflows is an important area in industrial software engineering, and, given that it typically occurs very early in a software development project, it is one of those areas, where model-driven approaches definitely have a competitive edge over code-driven approaches. As the UML has become the “*lingua franca of software engineering*” and is the cornerstone of the Model Driven Architecture initiative of the OMG, it is a natural choice for this task. Within the UML, Activity Diagrams are generally considered to be the appropriate notation for modeling business processes, workflows, and system-level behaviors, such as the composition of web-services. Unfortunately, the ActivityGraphs¹ of UML 1.5 have certain shortcomings in this respect, one of which is the lack of exception handling features (e.g. to implement transactional behavior of workflows).

As an example for the importance of exception handling facilities, consider implementing and maintaining business

use cases in an information system. Here, it’s a standard procedure to refactor the business use cases in a way such that on the one hand, there are a few standard cases (“sunshine scenarios”), which are modeled with great care and highly optimized. Apart from optimization, stability is the other great goal, i.e., making as few changes as possible in the basic cases. This keeps them simple and changeable. All difficulties and special cases are treated separately, and this is where exception handling plays an important role. With exception handling facilities, it is possible to factor out the code and business rules into separate business cases, implemented by separate code (or model) packages, thus enhancing the maintainability of the overall system.

The omission of exceptions has been corrected in the recent major revision (advancing the UML from version 1.5 to version 2.0), along with a complete redefinition of the respective part of the metamodel. However, the description in the standard raises a number of questions. This paper explores the new notions, both syntactically and semantically.

1.2 Approach

Since the standard stipulates that Activities “*use a Petri-like semantics*” (cf. [22, p. 292]), it is natural to use Petri nets as the semantic domain. However, exceptions imply a non-local flow of control which is notoriously difficult to model with Petri-nets, whose very purpose is to capture globally distributed states.

In [29], I have shown how procedure calling in UML Activity Diagrams might be mapped to a variant of Petri Nets. As raising an exception is similar to (prematurely) returning from a procedure call, the question is: is it possible to stretch the procedure-call approach of [29] a bit further to also cover exceptions?

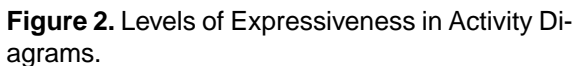
1.3 Related work

Since the UML standard has been written from scratch as far as Activity Diagrams are concerned, most of the previous work examining UML Activity Diagrams has become

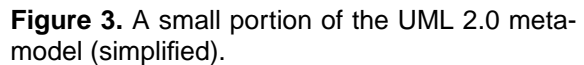
¹Adopting the convention of the standard, words with unexpected initial Capitals or “CamelCaps” refer to meta-classes.

In particular, exceptions, that have not been there in the UML 1.5 have not been addressed so far. It seems that so far, only very little has been published on the UML 2.0 Activity Diagrams: [3] examines expansions and streaming. [29, 34] provide formal definitions of the semantics of control-flow, procedure call, and data-flow in UML 2.0 Activities are provided, respectively. This paper builds on the latter two.

A detailed discussion of the concrete and abstract syntax of UML 2.0 Activities, the semantic domains of procedural and colored Petri-nets, respectively, and the semantic mapping of control- and data-flow of Activities is found in [29] and [34]. For lack of space, we can only give a short summary of the intuition here.



For basic Activities, the mapping to Petri-nets is rather simple. Intuitively, Actions that are ExecutableNodes become net transitions, ControlNodes become net places or



For data-flow, the mapping is similarly easy, but requires colored Petri-nets as the semantic domain to cover data-types, guards, and arc-inscriptions.

3 Concrete syntax and intuition of exceptions

3.1 Handling

In order to declare an `ExceptionHandler`, an `ExecutableNode` with an `ObjectNode` in Pin-notation is introduced as the handler. It is connected to some other Exe-

authors, references	semantic domain	expressiveness							rigor
		control flow	data flow	hierarchy	exceptions	streaming	expansion	loops	
Allweyer et al. [1]	–	wf	✓	–	–	–	–	–	low
Apvrille et al. [2]	LOTOS	wf	–	–	–	–	–	–	medium
Börger et al. [7]	ASM	wf	–	✓	–	–	–	–	medium
Bolton & Davies [5, 6]	CSP	wf	–	–	–	–	–	–	low
Eshuis & Wieringa [9, 10]	algorithm	wf, nwf	–	–	–	–	–	–	high
Eshuis & Wieringa [12, 11]	LTS	wf, nwf	–	–	–	–	–	–	high
Gehrke et al. [15]	PN	wf, nwf	(–)	–	–	–	–	–	medium
Pinheiro da Silva [25]	LOTOS	wf, time	–	–	–	–	–	–	low
Rodrigues [28]	FSP	wf	–	–	–	–	–	–	low
Li et al. [21]	LTS	wf	(–)	–	–	–	–	–	high
Störkle [29]	procedural PN	wf, nwf	–	✓	–	–	–	–	high
Störkle [34]	colored PN	wf, nwf	✓	–	–	–	–	–	high
Störkle [30]	colored PN	(wf, nwf)	✓	–	–	✓	✓	✓	medium
this paper	procedural colored PN	(wf, nwf)	(✓)	(✓)	✓	–	–	–	medium

Figure 1. Comparative categorization of the previous work leading to this article (in column “control-flow”, wf means well-formed, and nwf means non well formed, other abbreviations explained in text).

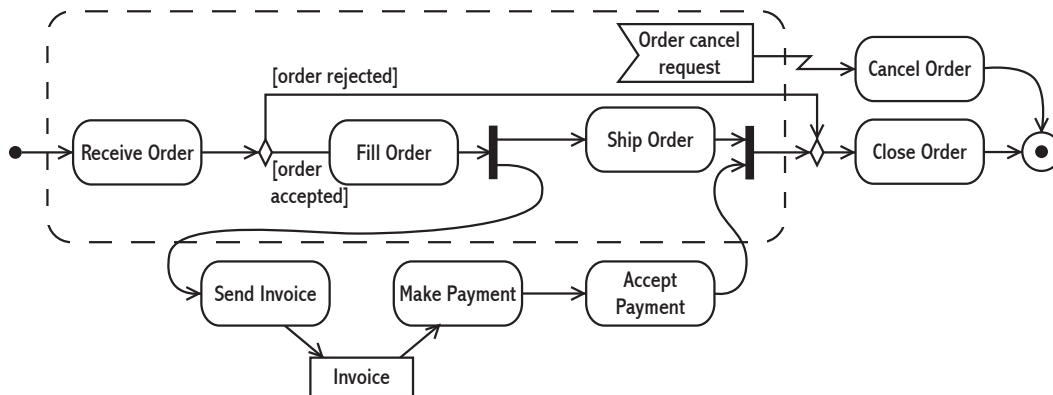


Figure 5. The example provided as Figure 260 by [22, p. 338], and which is the foundation for the definition of exceptions in UML 2.0 Activities.

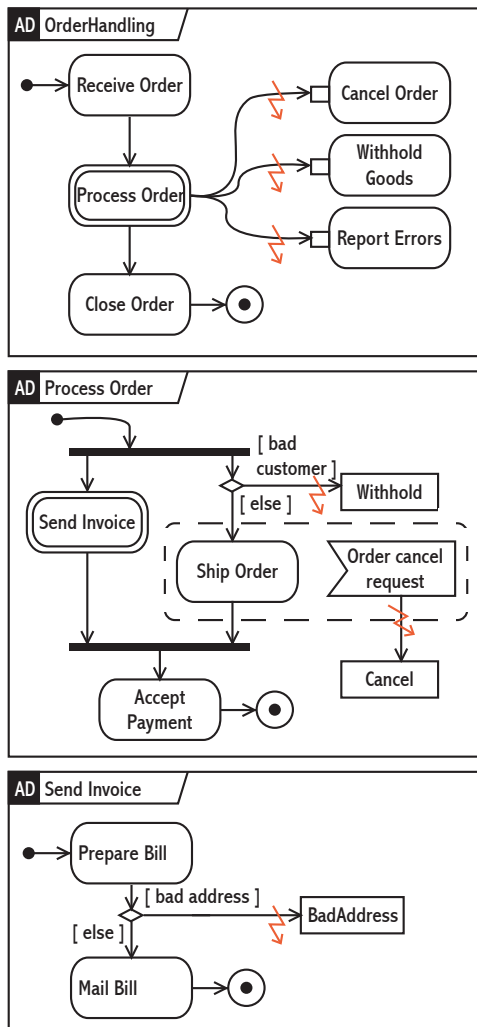


Figure 4. The running example consists of three nested Activity Diagrams. For clarity, Actions calling an Activity are presented with a double outline—this is not standard UML notation.

cutableNode (the protected node) with an ActivityEdge carrying a lightning-adornment (see Figure 5). Note that this is just a visual cue without semantics in itself.

In the metamodel, an ExceptionHandler consists of the protectedNode and the handlerBody (both of which are ExecutableNodes), the exceptionInput (an ObjectNode), and the exceptionType (a Classifier) of the exception (see again Figure 2). The standard explains the meaning of this construct as follows. “If an exception occurs during the execution of an action, the set of execution handlers on the action is examined for a handler that matches the exception. [. . .]. If there is a match, the handler catches the exception. The exception object is placed in the exceptionInput node

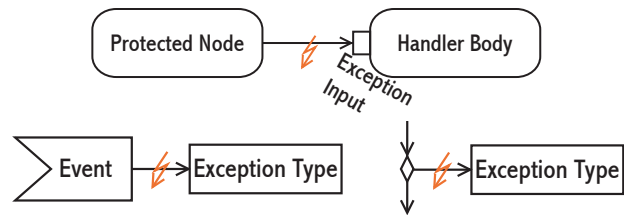


Figure 6. Specifying an ExceptionHandler (top, see [22, p. 323, Fig. 240]); two ways to raise an exception (bottom).

as a token to start execution of the handler body.” (cf. [22, p. 323f]) Then, since the “successors to an exception handler body are the same as the successors to the protected node” (cf. [22, p. 323]), the handler node dynamically replaces the (aborted) protectedNode: “the result tokens of the exception body become the result tokens of the protected node. Any control edges leaving the protected node receive control tokens on completion of execution of the exception body. When the execution body completes execution, it is as if the protected node had completed execution.” (cf. [22, p. 323]).

If some exception is not handled, aborting the enclosing Activities cascades upwards the calling hierarchy, leaving a trace of exit-transitions, similar to the stack trace of, say, Java.

3.2 Raising

There are three separate aspects of exception raising that need be considered, namely

- the **trigger**, i.e. the occurrence responsible for actually raising an exception;
- the **scope of readiness**, i.e. a kind of precondition in terms of the states of an Activity in which the exception may be triggered; and
- the **scope of preemption**, i.e. the set of (concurrent!) control- and data-flows that are preempted by raising a certain exception.

Consider the trigger first. The standard proposes only a single kind of trigger to raise an exception, namely (external) events (“event trigger”, see Figure 5). However, experience with exceptions in programming languages also suggests that raising an exception should also be possible as the consequence of a case distinction (“branch trigger”). This may be expressed simply as an ActivityEdge with lightning-adornment emanating from a BranchNode (see Figure 5).

However, the metamodel fails to define elements to represent Events in Activity Diagrams. Thus we have introduced AcceptEventNode and SendEventNode (gray boxes in Figure-2), intended to correspond to AcceptEventAction and SendEventAction.

Either way, the notation proposed in the standard is very confusing, since it uses the same notation for raising an exception and for handling it (cf. Figure 5). Also, it is not clear, how to declare the type of the exception or its parameters with this notation.

Thus, we propose the following notation for raising an exception: the exceptionInput (an ObjectNode) is presented in the “*standalone*” (cf. [22, p. 357]) presentation option, and is connected to the trigger with an ActivityEdge with lightning-adornment. This way, the parameter passed to potential handlers is made explicit. The handlerBody (an ExecutableNode) is omitted altogether, removing a source of confusion in the standard: for the handlerBody is part of *another* Activity than the other nodes, and mixing elements from different Activities in one diagram makes them hard to understand.

Progressing to the scope of readiness, one possibility is that any of the states of the whole Activity allows an exception to be raised, if the respective trigger occurs. The other possibility is to restrict the scope, and the standard introduces the notion of InterruptibleActivityRegion to do just this. It is presented as a dashed line around some elements of an Activity Diagram, the only restriction being that no two InterruptibleActivityRegions may overlap. For the example in Figure 4, this means that the occurrence of the “Order cancel request” triggers the “Cancel Order”-exception, provided that “Ship Order” has not been completed, even if “Send Invoice” has already been processed.

This leads us to the scope of preemption. Continuing the example, if there is an InterruptibleActivityRegion, should the whole of the Activity be aborted, or just the InterruptibleActivityRegion? The standard seems to support the second version by stating that “*when a token leaves an interruptible region via [lightning-bolt] edges, all tokens and behaviors in the region are terminated.*” (cf. [22, p. 337, emphasis added]) On the other hand, this is counter-intuitive, given that the InterruptibleActivityRegion is not a behavioral, but rather a syntactic construct. Also, the examples referring to Figures 240f in [22, p. 323f] seem to suggest that the whole protected node is aborted. Finally, the latter version would be excessively inconvenient to formalize using colored Petri-nets.² Therefore, I chose the interpretation that the scope of preemption is the whole smallest enclosing Activity.

²Using flush-arcs would be a possible way out, but this leads to even more trouble.

4 Metamodel

A small portion of the UML 2.0 metamodel is shown in Figure 2. In the metamodel, an Activity is a graph of ActivityNodes and ActivityEdges of various kinds. Exceptions are defined by ExceptionHandlers, which refer to three of the ActivityNodes as the protectedNode (where the exception is raised), the handlerBody (where it is handled), and the exceptionInput (the Parameter). Observe that ExceptionHandlers are not defined as part of an Activity by the standard.

Nevertheless, we assume here that an Activity is represented as a triple $\langle Nodes, Edges, Handlers \rangle$, the elements of which are further partitioned according to the subclasses in the metamodel. For instance, *Nodes* has subsets *EN* for the ExecutableNodes, *ON* for ObjectNodes, and so on. Similarly, a *Handler* is represented as a tuple $\langle protected, body, input \rangle$. Since p_N^c and $p_{\rho(t)}^r$ are unique for any protectedNode t , no further connection needs to be established.

For simplicity, we assume that each Activity Diagram appears boxed and named similar to UML 2.0 Interaction Diagrams (see Figure 3), and that Activity Diagrams are available as tuples $\langle Name, Activity \rangle$. We introduce the term Activity Specification for sets of Activity Diagrams.

5 Semantic domain

While it would be possible to use (hierarchical) colored Petri nets to express exception-like behavior, the mapping from Activities becomes rather clumsy and unintuitive. Thus we propose to change the net formalism instead. Since raising an exception is conceptually similar to (prematurely) returning from a procedure call, procedural Petri-nets are a good starting point. For lack of space, we can only briefly repeat the basic definitions here. Details and examples may be obtained from [20] and [29].

Definition 5.1 (structure of procedural Petri nets)

A pair $NS = \langle N, \rho \rangle$ is a procedural Petri-net (PPN), iff N is a finite set of Petri-nets with initial and final markings (\overline{m} and \underline{m} , respectively) and ρ is a partial function $\rho : T_N \rightarrow N$ from transitions of the nets of N into N . A state of NS is a multiset of elements $e \in C \times I \times M \times F$, where

- C is the set of callers, defined as $dom(\rho) \cup \{\perp\}$;
- I is a set of globally unique instance identifiers;
- M is the class of markings of the nets in NS ;
- F is the set of procedure instances called from e .

A state element for a net that is not called from anywhere (e.g., the initial marking) has \perp as the “caller”. \square

Observe that \perp is the root of a tree of call dependencies, i.e., a kind of least element. The definition of the behavior of PPNs is defined in separate rules for normal firing, procedure call and procedure return.

Definition 5.2 (ordinary transitions)

An unrefined transition t is an , it is activated in s , iff $\langle c, i, m, f \rangle \in s$ with $\bullet t \leq m$ and either $t \in T_{\rho(c)}$ or $c = \perp$. If t is activated in s , it may fire reaching s' with

$$s' = s - \langle c, i, m, f \rangle + \langle c, i, m - \bullet t + t^\bullet, f \rangle. \quad \square$$

The notations $\bullet x$ and x^\bullet denote the pre- and post-set of x as usual. For the behavior of calling a procedure and returning from it, consider Figure 6. There, a transition *protected* is refined to N' . When *protected* fires, an instance of N' with the new id j is instantiated, and N'_j is provided with its initial marking (event t_{call}^j).

Definition 5.3 (procedure call transitions)

A refined transition t is activated to do a procedure call t_{call}^j of instance j of net $\rho(t)$ in state s , if there is $\langle c, i, m, f \rangle \in s$ with $\bullet t \geq m$ and j is currently not used in s . When t_{call}^j is activated, it may fire, creating a new instance j of $\rho(t)$, reaching a new state s' with

$$s' = s - \langle c, i, m, f \rangle + \langle c, i, m - \bullet t, f \cup \{j\} \rangle + \langle t, j, \bar{m}_{\rho(t)}, \emptyset \rangle. \quad \square$$

If N' reaches its final marking \bar{m} . If all procedures called from N'_j have also terminated, the instance j is removed.

Definition 5.4 (procedure return transitions)

The instance j of a refined transition t may perform a procedure return in state s (written t_{return}^j), if it is activated, i.e., it has reached its final marking and all of its function calls have terminated (formally $\langle c, i, m, f \cup \{j\} \rangle \langle t, j, \bar{m}_{\rho(t)}, \emptyset \rangle \in s$). When t_{return}^j is activated in s , it may fire reaching s' and removing instance j , with

$$s' = s - \langle c, i, m, f \cup \{j\} \rangle + \langle c, i, m + t^\bullet, f \rangle - \langle t, j, \bar{m}_{\rho(t)}, \emptyset \rangle. \quad \square$$

The straight run in Figure 9 (left) provides an example of a run of a PPN. Exception Petri-nets have the additional property, that procedure calls may be aborted prematurely, when a special place p^r is marked. In order to distinguish between different tokens put on p^r , we need to use colored nets [18] (or any other kind of high-level Petri-nets).

Definition 5.5 (structure of exception Petri nets)

An exception Petri-net (EPN) is a PPN $NS = \langle N, \rho \rangle$, where each $N \in N$ is colored Petri-nets with special places p_N^c and

p_N^r , and a special transition t_N^p such that $\bullet p_N^c = \emptyset = p_N^r \bullet$ and $p_N^c = \bullet t_N^p$ and $t_N^p \bullet = p_N^r$. A state of an EPN is a multiset of state elements $C \times I \times M \times F$ similar to the state of a PPN, only that M are now markings of colored nets. \square

Continuing the explanation from above, suppose that at some point, $p_{\rho(t)}^r$ may be marked, triggering the event t_{exit}^j . Now the execution of invocation j of $\rho(t)$ may be aborted, transferring the tokens on $p_{\rho(t)}^r$ to p_N^c . Then, either *handle* deals with the exception, or *propagate* moves the token to p_N^r , thus propagating the exception one level up. This is achieved by an additional firing rule similar to the exit rule (see below). It has priority over “normal” firing and removes instance j and all the calls it has made since being instantiated.

Definition 5.6 (procedure exit transitions)

Instance j of a refined transition t may perform a procedure exit in state s (written t_{exit}^j), if it is activated, i.e., its raised-place p_N^r is marked (formally: $\langle c, i, m, f \cup \{j\} \rangle \langle t, j, m', sub \rangle \in s$, with $m'(p_{\rho(t)}^r) \geq \epsilon$).

When t_{exit}^j is activated in s , it must fire prior to any “normal” transition (ordinary, procedure call, or procedure return) reaching s' and removing instance j (written $s \xrightarrow{t_{exit}^j} s'$), with

$$s' = s - \langle c, i, m, f \cup \{j\} \rangle + \langle c, i, m + t^\bullet, f \rangle - \langle t, j, \bar{m}, calls \rangle - CALLS,$$

where $CALLS$ is the largest fixed point of the equation

$$SUB_{s'}(calls) = \{ \langle t, j, m, calls' \rangle \in s' \mid j \in calls \} \cup SUB_{s'}(calls'). \quad \square$$

6 Semantic mapping

Now the mapping from Activities to EPNs is rather straightforward: a specification $Spec = \{Activities, Handlers\}$ maps into a PPN $\langle N, \rho \rangle$, such that *protectedNodes* map into refined transitions t , and their *handlerBodies* map into $\rho(t)$. All *exceptionInputs* of *Handlers* of some *Activity* map into the same p_N^c . All *ActivityEdges* with lightning-adornments that raise an exception in the *Activity* that map into transitions t' of $\rho(t)$ with $t'^\bullet = p_{\rho(t)}^r$. See Figure 6: the gray net elements result from translating one handler.

At this point, we reuse and adapt earlier results: $\llbracket - \rrbracket_{DF}$ from [34] maps an *Activity* into a colored Petri-net, and $\llbracket - \rrbracket_{CF}$ from [29] maps a set of *Activities* into a PPN. All we need to do is extend the functions such that they also map *ExceptionHandler*s, and yield EPNs instead of PPNs.

Three additions are needed (see Figure 7). First, the special places p_N^r and p_N^c and the transition *propagate* must

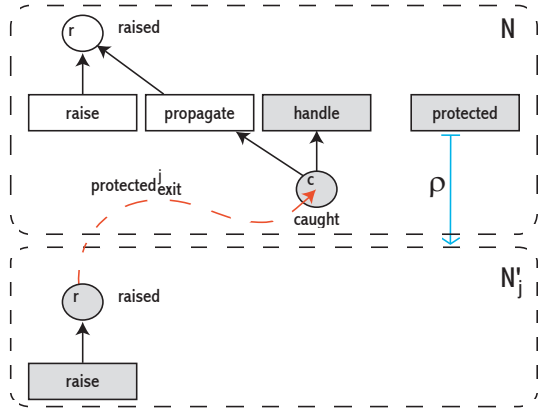


Figure 7. Schema of raising, handling, and propagating exceptions in exception Petri-nets.

be added with appropriate inscriptions. Second, for each handler, there must be a handle-transition. Third, for each Exception raised in the Activity, there must be a raise-transition. The exceptions raised in an Activity A may be determined by the handlers in the set of all Activities calling A . Figure 7 shows the details.

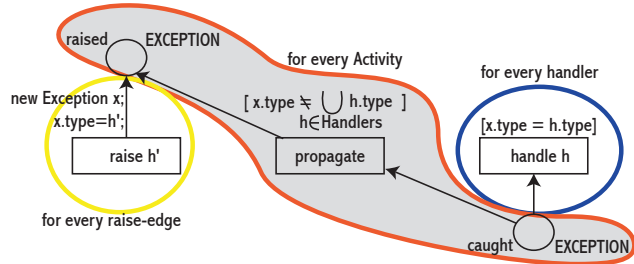


Figure 8. Mapping an ExceptionHandler.

Finally, InterruptibleActivityRegions must be translated: some given Exception is enabled while an Activity is in a on of a set of states. This can be modeled by a pair of arm/disarm-transitions (cf. Figure 8). For every ActivityEdge that enters the InterruptibleActivityRegion, an additional when an InterruptibleActivityRegion is entered, the arm-transition places a token on a run-place of the transition that resulted from translating the respective protectedNode. Conversely, for every ActivityEdge that leaves the InterruptibleActivityRegion, a disarm-transition is added. Note that this way, unbalanced synchronisation inside an InterruptibleActivityRegion can result in illegal raising of exceptions. For instance, if the JoinNode in Figure 4 were outside the InterruptibleActivityRegion, this translation scheme breaks

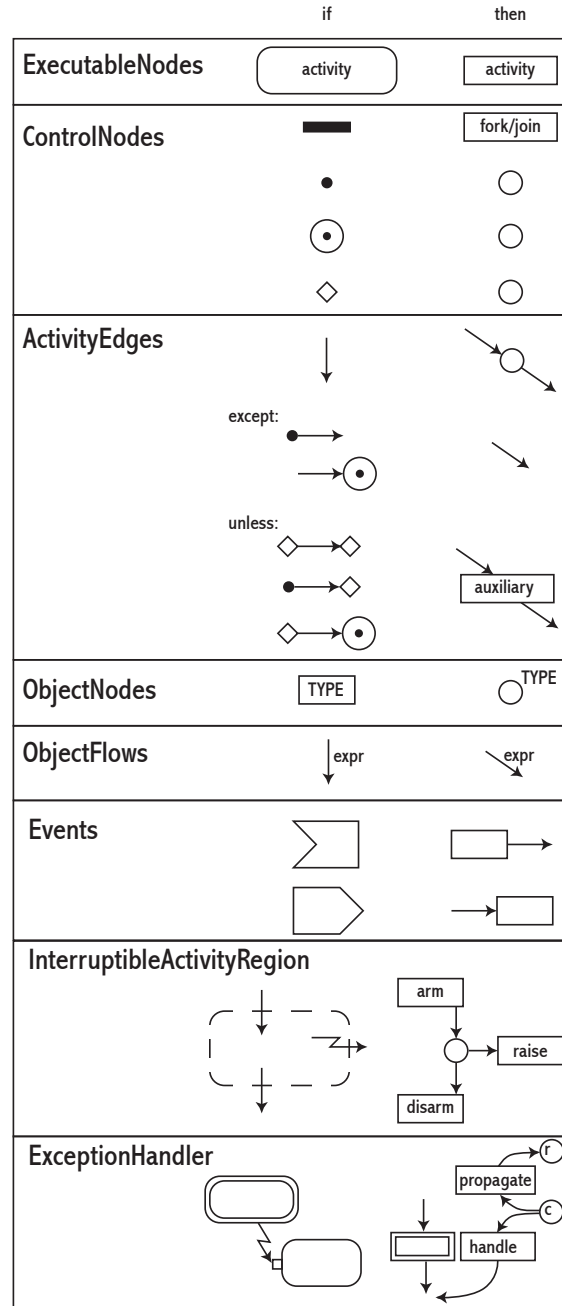


Figure 9. The intuition of the semantic mapping for Activities.

down. However, this problem cannot be avoided without a semantic analysis of the dataflow in the InterruptibleActivityRegion—but these may be arbitrarily complex (colored Petri-nets are Turing-complete).

As an example both for PPNs and for the semantic mapping, reconsider Figure 3. It shows a specification *Spec* based on the main example from the standard (cf. [22, p. 323], here reproduced as Figure 4), and contains three Activity Diagrams named *OrderHandling*, *Process Order*, and *Send Invoice*, respectively. Translating them yields the EPN $E = \langle \{N_1, N_2, N_3\}, \rho \rangle$, where N_1 to N_3 are shown in Figure 10 and ρ is $\{Process\ Order \mapsto N_2, Send\ Invoice \mapsto N_3\}$. The initial marking of E is $\langle \perp, 0, p_0, \emptyset \rangle$, its final marking is \underline{m} is $\langle \perp, 0, p_3, \emptyset \rangle$. Figure 9 shows some (fragments of) sample runs of the PPN *Spec* translates into.

On the left, there is a straight run $\alpha.\beta.\gamma.\delta$ without any exceptions raised or handled. When the exceptions W , C , and B are raised, traces $\alpha.W.\delta$, $\alpha.C.\delta$, and $\alpha.B$ may occur, respectively. Of course, other interleavings are possible. The last run nicely shows a cascade of (two) exception handling attempts which finally fails, aborting the PPN altogether.

7 Conclusion

7.1 Summary

In this paper, the exception-constructs of UML 2.0 Activities are examined by defining a compositional semantics based on a procedure call variant of Petri nets. A completely formal semantics for exceptions exists, but could not be presented here for lack of space. Some problems concerning the concrete and abstract syntax and the semantics have been uncovered in the standard (unbalanced fork/joins in InterruptibleActivityRegions, undefined preemption scope, unsuitable examples), and possible solutions have been proposed.

7.2 Contribution

There have been several proposals for semantics of Activity Diagrams, but most of these aim at UML 1.x. For UML 2.0, there are only [3, 29, 34]. This paper covers exceptions, and together Together with its companion papers [29, 34], this paper covers all of UML 2.0 Activity Diagrams in a critical and coherent style.

7.3 Open questions

There are still some concepts in UML 2.0 Activities, that have not yet been explored (expansion regions, streaming). The combination with other parts of the UML must be examined, in particular the relationship to Interactions and

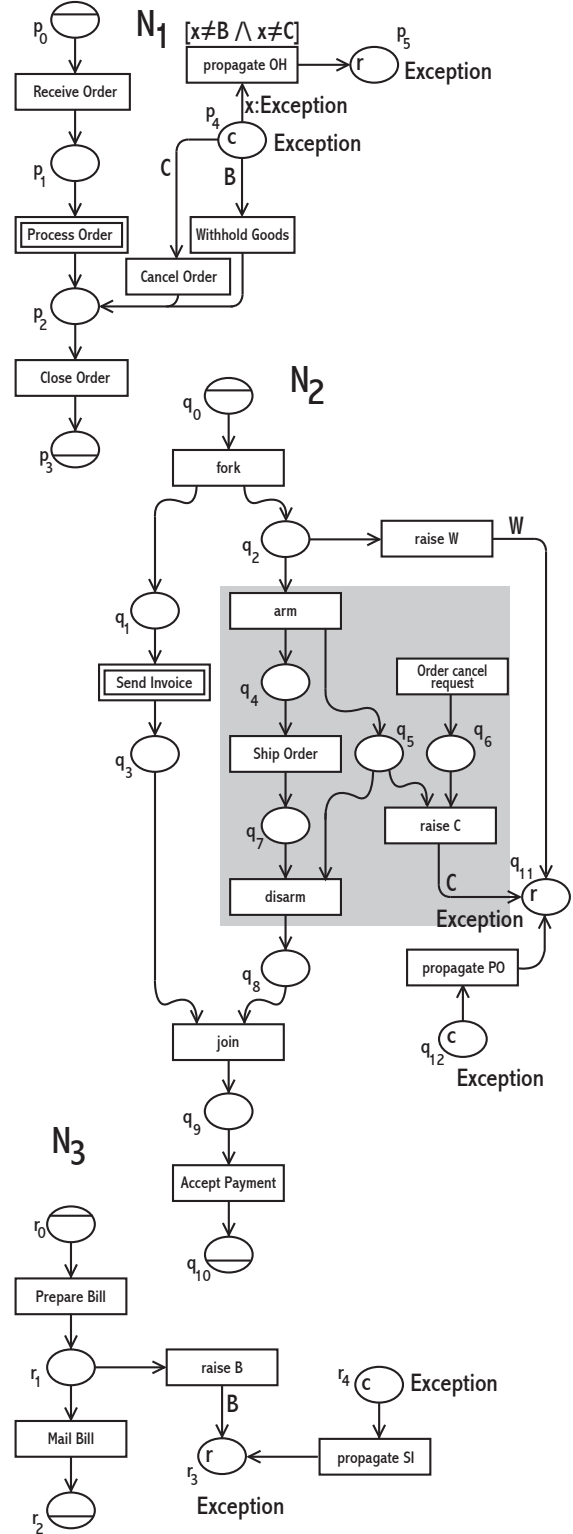


Figure 11. The result of translating the specification shown in Figure 3.

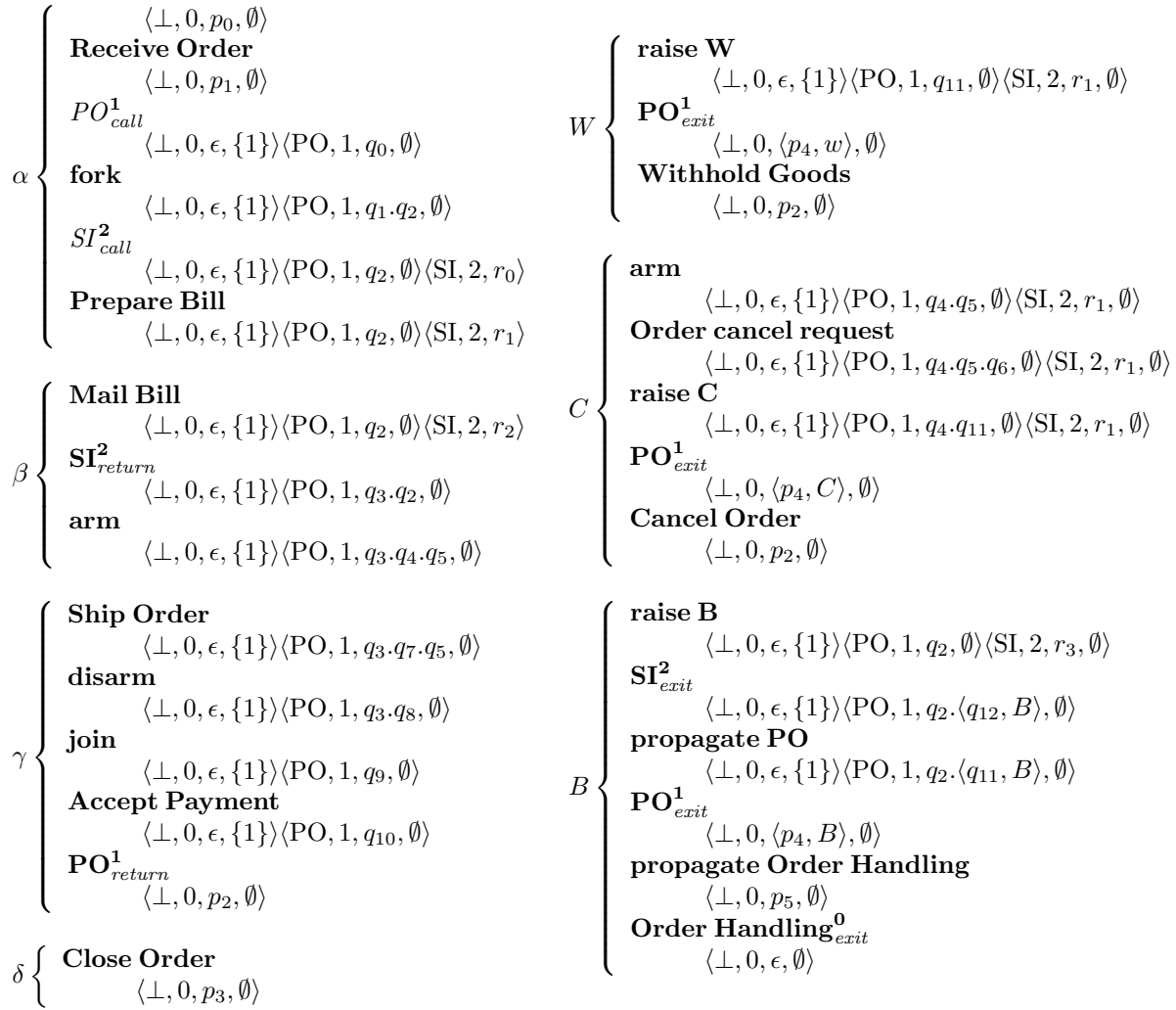


Figure 10. Some runs of the net system in Figure 10, representing the set of Activity Diagrams of Figure 3. The names of the refined transitions have been abbreviated to PO and SI.

StateMachines, whose natural semantic domains must be related to Petri-nets. Also, examples soon become too complex for manual treatment, and so we need tool-support.

?? Anschlu glaetten

7.4 Ongoing work

The work presented here is currently being extended along several lines.

Integration The standard defines three fundamentally different, but tightly integrated views of behavior: StateMachines, Activities, and Interactions. Their formal semantics should also be compatible, but it is

currently unclear, how this is best achieved. Some of the problems are:

- Each of the views has its own “natural” semantic domain, e.g. sequences for Interactions, Petri nets for Activities, and state machines for StateMachines. Bringing them together in a way that is meaningful for intuition and useful for practical purposes raises a number of issues.
- While most of the simpler parts of UML 2.0 Interactions are fairly well understood by now (cf. [32]), there are definitely open questions both for Interactions (cf. [31]) and Activities. Also, even if at first sight, StateMachines don’t seem to have much changed from UML 1.5 to 2.0,

this must be examined, and possible repercussions to the other parts of the behavioral semantics must be assessed.

Furthermore, behavioral descriptions must be aligned with static and functional descriptions, i.e. use cases, parts/ports (“architecture diagrams”), and classes.

Tools/Verification Finally, a tool implementation for verification tasks (see e.g. [9, 10, 13]), time analysis (see e.g. [21]), and in general, validating and further exploring the semantics and the standard is under way.

References

- [1] T. Allweyer and P. Loos. Process Orientation in UML through Integration of Event-Driven Process Chains. In P.-A. Muller and J. Bézivin, editors, *International Workshop «UML» '98: Beyond the Notation*, pages 183–193. Ecole Supérieure des Sciences Appliquées pour l'Ingénieur—Mulhouse, Université de Haute-Alsace, 1998.
- [2] L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, and J.-P. Courtiat. A New UML Profile for Real-Time System Formal Design and Validation. In Gogolla and Kobryn [17], pages 287–301.
- [3] J. P. Barros and L. Gomes. Actions as Activities as Petri nets. In Weber et al. [35], pages 129–135.
- [4] B. Baumgarten. *Petri-Netze. Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, Heidelberg, 1996. 2. Aufl.
- [5] C. Bolton and J. Davies. Activity graphs and processes. In W. Griesskamp, T. Santen, and W. Stoddart, editors, *Proc. Intl. Conf. Integrated Formal Methods (IFM'00)*. Springer Verlag, 2000. LNCS.
- [6] C. Bolton and J. Davies. On giving a behavioural semantics to activity graphs. In Reggio et al. [26], pages 17–22.
- [7] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, *Proc. 8th Intl. Conf. Algebraic Methodology and Software Technology (AMAST 2000)*, pages 293–308. Springer Verlag, May 2000. LNCS 1816.
- [8] M. Dumas and A. H. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In Gogolla and Kobryn [17], pages 76–90.
- [9] H. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, CTIT, U. Twente, 2002. Authors first name sometimes appears “Rik”.
- [10] R. Eshuis and R. Wieringa. A formal semantics for UML Activity Diagrams - Formalising workflow models. Technical Report CTIT-01-04, U. Twente, Dept. of Computer Science, 2001.
- [11] R. Eshuis and R. Wieringa. A Real-Time Execution Semantics for UML Activity Diagrams. In H. Hussmann, editor, *Proc. 4th Intl. Conf. Fundamental approaches to software engineering (FASE'01)*, number 2029 in LNCS, pages 76–90. Springer Verlag, 2001. Also available as wwwhome.cs.utwente.nl/~tcm/fase.pdf.
- [12] R. Eshuis and R. Wieringa. An Execution Algorithm for UML Activity Graphs. In Gogolla and Kobryn [17], pages 47–61.
- [13] R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. In *Proc. 24th Intl. Conf. on Software Engineering (ICSE'02)*, pages 166–176. IEEE, 2002.
- [14] R. Eshuis and R. Wieringa. Comparing Petri Net and Activity Diagram Variants for Workflow Modelling - A Quest for Reactive Petri Nets. In Weber et al. [35], pages 321–351.
- [15] T. Gehrke, U. Goltz, and H. Wehrheim. The Dynamic Models of UML: Towards a Semantics and its Application in the Development Process. Technical Report 11/98, Institut für Informatik, Universität Hildesheim, 1998.
- [16] H. Genrich and K. Lautenbach. *Predicate/Transition Nets*. In [19], 1991.
- [17] M. Gogolla and C. Kobryn, editors. *Proc. 4th Intl. Conf. on the Unified Modeling Language («UML» 2001)*, number 2185 in LNCS. Springer Verlag, 2001.
- [18] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. I*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.
- [19] K. Jensen and G. Rozenberg. *High-Level Petri Nets. Theory and Application*. Springer Verlag, 1991.
- [20] A. Kiehn. *A Structuring Mechanism for Petri Nets*. Dissertation, TU München, 1989.
- [21] X. Li, M. Cui, Y. Pei, Z. Jianhua, and Z. Guoliang. Timing Analysis of UML Activity Diagrams. In Gogolla and Kobryn [17], pages 62–75.
- [22] OMG Unified Modeling Language: Superstructure (final adopted spec, version 2.0). Technical report, Object Management Group, November 2003. Available at www.omg.org, downloaded at November 11th, 2003, 11³⁰.
- [23] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall Inc., Englewood Cliffs NJ, 1981.
- [24] D. C. Petriu and Y. Sun. Consistent Behaviour Representation in Activity and Sequence Diagrams. In B. Selic, S. Kent, and A. Evans, editors, *Proc. 3rd Intl. Conf. «UML» 2000—Advancing the Standard*, number 1939 in LNCS, pages 369–382. Springer Verlag, October 2000.
- [25] P. Pinheiro da Silva. A proposal for a LOTOS-based semantics for UML. Technical Report UMCS-01-06-1, Dept. of Computer Science, U. Manchester, 2001.
- [26] G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa, editors. *Dynamic Behavior in UML Models: Semantic Questions. Workshop Proceedings*, Oktober 2000.
- [27] W. Reisig. *Petri-Nets: an Introduction*. Springer Verlag, 1985.
- [28] R. W. Rodrigues. Formalising UML Activity Diagrams using Finite State Processes. In Reggio et al. [26], pages 92–98.
- [29] H. Störrle. Semantics of Control-Flow in UML 2.0 Activities. submitted for review at VLFM'04, March, 20th, 2004.
- [30] H. Störrle. Semantics of Expansion-Regions and Streaming in UML 2.0 Activities. submitted for ??
- [31] H. Störrle. Assert, Negate and Refinement in UML-2 Interactions. In J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, editors, *Critical Systems Development with UML - Proceedings of the UML'03 Workshop*, pages 79–94, 2003.

- [32] H. Störrle. Semantics of Interactions in UML 2.0. In J. Hosking and P. Cox, editors, *Human Centric Computing Languages and Environments*, pages 129–136. IEEE Computer Society, 2003.
- [33] H. Störrle. Trace Semantics of Interactions in UML 2.0. *J. Visual Languages and Computing*, t.b.d. 2003. submitted for review, February, 13th, 2004.
- [34] H. Störrle. Semantics of Data-Flow in UML 2.0 Activities. In N.N., editor, *Proc. 7th Intl. Conf. on the Unified Modeling Language (UML 2004)*, number tba in LNCS. Springer Verlag, 2004. submitted for review at UML’04, March, 31st, 2004.
- [35] M. Weber, H. Ehrig, and W. Reisig, editors. *Petri Net Technology for Communication-Based Systems*. DFG Research Group “Petri Net Technology”, 2003.

A Elementary Petri-nets

Classic references to Petri nets are [27, 23], or more recently [4] (only available in German, unfortunately).

Definition A.1 (structure of nets)

A triple $N = \langle P, T, A \rangle$ is a (Petri-)net, iff P and T are disjoint, and $F \subseteq P \times T \cup T \times P$. The elements of the triple are called Places, Transitions, and Arcs. The set $X = P \cup T$ is also called net elements. These are sometimes denoted as P_N, T_N , and A_N , and depicted as circles, boxes and arrows, respectively. The pre- and post-set of $x \in X$ (written $\bullet x$ and x^\bullet , respectively) is defined as $\bullet x = \{y \mid \langle y, x \rangle \in A\}$ and $x^\bullet = \{y \mid \langle x, y \rangle \in A\}$.

The markings of N are the multisets (or words) over P , i.e. $m \in P^*$. The notation $m(p)$ is used to denote the number of tokens at place p in marking m .

A quintuple $N = \langle P, T, A, \overline{m}, \underline{m} \rangle$ is a complete (Petri-)net, iff $\langle P, T, A \rangle$ is a net, and \overline{m} and \underline{m} are its initial and final markings. \square

Definition A.2 (behavior of nets)

Let $N = \langle P, T, A, \overline{m}, \underline{m} \rangle$ be a complete net. A transition t of N is activated under marking m (written $N : m \xrightarrow{t}$ or simply $m \xrightarrow{t}$, if N is clear), iff $m \geq \bullet t$. If t is activated, it may occur (or “fire”), yielding a new marking m' (written $N : m \xrightarrow{t} m'$) with $m' = m - \bullet t + t^\bullet$.

A sequence $w \in T^*$ (its length being denoted $|w|$) is called firing sequence of N starting at m , iff there are markings $\{m_0, \dots, m_{|w|}\}$, such that $\forall_{0 < i < |w|} : m_{i-1} \xrightarrow{w_i} m_i$. Abbreviating, one may write $m_0 \xrightarrow{w} m_{|w|}$.

A marking m' is reachable from a marking m (written $m \rightarrow m'$), iff there is a firing sequence w such that $m \xrightarrow{w} m'$. The set of markings reachable from m is denoted $m \rightarrow$. \square

B Colored Petri-nets

The classic reference for colored Petri-nets is [18]. Other dialects of higher-order Petri-nets are found in [19, 16].

Definition B.1 (structure of colored Petri-nets)

A tuple $\langle N, SigAlg, color, guard, effect \rangle$ is a colored Petri-net (CPN), iff

N	is a Petri net $\langle P, T, F \rangle$ of places, transitions, and flow arcs;
$SigAlg$	is a Σ -algebra $\langle \Sigma, Op \rangle$ of sorts and operations;
$color$	is a total function $P \mapsto \Sigma$ assigning a type (“color”) to each place;
$guard$	is a total function $T \mapsto Expr$ assigning a boolean expression to each transition;
$effect$	is a total function $A \mapsto Expr$ assigning an expression to each arc, its type being the color of the place of the arc.

For convenience, $color$, $guard$, and $effect$ may be specified partially, with black-dot tokens as the default. That is, if $color(p)$ is undefined, then $color(p) = \text{TOKEN}$ is intended, and analogously for $guard$ and $effect$. \square

The definition of the behavior of CPNs is a little more complicated, as we now need to take into account the values of tokens and the meanings of operations on them. A marking of a CPN is multiset (or word) over $\{\langle p, v \rangle \mid p \in P, v \in color(p)\}$.