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 [4] has introduced significant changes and additions to “the lingua franca of Software Engineering”. One of the most interesting innovations is exception handling. Building on [5, 8], 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 [3].

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

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 ActivityGraphs\textsuperscript{1} 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.

2 Approach and related work

Since the standard stipulates that Activities “use a Petri-like semantics” (cf. [4, 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 [5], 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 [5] a bit further to also cover exceptions?

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 (see [5] or [8] for an exhaustive survey and comparison) has become obsolete. 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: [1] examines expansions and stream-

---

\textsuperscript{1}Adopting the convention of the standard, words with unexpected initial Capitals or “CamelCaps” refer to meta-classes.
ing. [5, 8] 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.

3 Activity Diagrams without exceptions

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 [5] and [8]. For lack of space, we can only give a short summary of the intuition here.

While in UML 1.5, Activity Diagrams have been defined as a kind of State Machine Diagrams (ActivityGraph used to be a subclass ofStateMachine in the Metamodel), there is now no such connection between them: “Activities are redesigned to use a Petri-like semantic” (cf. [4, p. 292]). The intuition of the semantic mapping is presented in the first six compartments of Figure 7.

For basic Activities, the mapping to Petri-nets is rather simple. Intuitively, Actions that are ExecutableNodes become net transitions, ControlNodes become net places or small net fragments, and ActivityEdges become net arcs, possibly with auxiliary transitions or places.

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.

For procedure calling, traditional P/T-nets are not sufficient, so that [5] resorts to procedural Petri-nets (first described in [3]). We require that each Activity is represented by a separate boxed and named Activity Diagram similar to UML 2.0 Interaction Diagrams (cf. Figure 2 and [7, 6]), each of which may then be transformed separately into a plain Petri-net. These individual nets are held together by a refinement relationship ρ, exploited at run-time, i.e., when the Petri-net is executed.

Figure 1. A small portion of the UML 2.0 meta-model (simplified).

Figure 2. 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.
4 Concrete syntax and intuition of exceptions

Obviously, there are two aspects of exceptions—raising and handling—and both need appropriate syntactic representations. Unfortunately, the standard is somewhat vague in this respect, and does not properly distinguish these two cases (see Figure reffig:std:260).

4.1 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 ExecutableNode (the protected node) with an ActivityEdge carrying a lightning-adornment (see Figure 4). Note that this is just a visual cue without semantics in itself.

![Figure 4. Specifying an ExceptionHandler (top, see [4, p. 323, Fig. 240]); two ways to raise an exception (bottom).](image)

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 1). 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 as a token to start execution of the handler body." (cf. [4, p. 323f]) Then, since the "successors to an exception handler body are the same as the successors to the protected node" (cf. [4, 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. [4, p. 323]).

4.2 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 4). 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 4). However, the metamodel fails to define elements to represent Events in Activity Diagrams. Thus we have introduced AcceptEventNode and SendEventNode (gray boxes in Figure-1), 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 4). 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. [4, 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 3, 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. [4, 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 [4, 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.

5 Metamodel

A small portion of the UML 2.0 metamodel is shown in Figure 1. 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 \((\text{Nodes}, \text{Edges}, \text{Handlers})\), the elements of which are further partitioned according to the subclasses in the metamodel. For instance, \text{Nodes} has subsets \(EN\) for the ExecutableNodes, \(ON\) for ObjectNodes, and so on. Similarly, a \text{Handler} is represented as a tuple \((\text{protected}, \text{body}, \text{input})\). Since \(p^N_\rho\) and \(p^\rho_\sigma\) 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 2), and that Activity Diagrams are available as tuples \((\text{Name}, \text{Activity})\). We introduce the term Activity Specification for sets of Activity Diagrams.

6 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 [3] and [5].

Definition 6.1 (structure of procedural Petri nets)

A pair \(NS = (N, \rho)\) is a procedural Petri-net (PPN), iff \(N\) is a finite set of Petri-nets with initial and final markings \((m_i, m_f)\), respectively) and \(\rho\) 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 \(e \in C \times T \times M \times F\), where

\(^2\)Using flush-arcs would be a possible way out, but this leads to even more trouble.
C is the set of callers, defined as \( \text{dom}(\rho) \cup \{ \bot \} \);
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 \( \bot \) as the “caller”. □

Observe that \( \bot \) 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 6.2 (ordinary transitions)**
An unrefined transition \( t \) is an \( i \)-transition, if \( (c, i, m, f) \in s \) with \( *t \leq m \) and either \( t \in T_{\rho(c)} \) or \( c = \bot \). If \( t \) is activated in \( s \), it may fire reaching \( s' \) with

\[
s' = s - \langle c, i, m, f \rangle + \langle c, i, m - t, f \rangle.
\]

The notations \( *x \) and \( x^* \) denote the pre- and post-set of \( x \) as usual. For the behavior of calling a procedure and returning from it, consider Figure 5. There, a transition \( \text{protection} \) is refined to \( N' \). When \( \text{protected} \) fires, an instance of \( N' \) with the new id \( j \) is instantiated, and \( N'_j \) is provided with its initial marking (event \( t^j_{\text{call}} \)).

**Definition 6.3 (procedure call transitions)**
A refined transition \( t \) is activated to do a procedure call \( t^j_{\text{call}} \) of instance \( j \) of net \( \rho(t) \) in state \( s \), if there is \( (c, i, m, f) \in s \) with \( *t \geq m \) and \( j \) is currently not used in \( s \). When \( t^j_{\text{call}} \) is activated, it may fire, creating a new instance \( j \) of \( \rho(t) \), reaching a new state \( s' \) with

\[
s' = s - \langle c, i, m, f \rangle + \langle c, i, m - *t, f \cup \{ j \} \rangle + \langle t, j, m_{\rho(t)}(\emptyset) \rangle.
\]

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

**Definition 6.4 (procedure return transitions)**
The instance \( j \) of a refined transition \( t \) may perform a procedure return in state \( s \) (written \( t^j_{\text{return}} \)), if it is activated, i.e., it has reached its final marking and all of its function calls have terminated (formally \( (c, i, m, f) \in \{ j \} \mid \langle t, j, m_{\rho(t)}(\emptyset) \rangle \in s \). When \( t^j_{\text{return}} \) is activated in \( s \), it may fire reaching \( s' \) and removing instance \( j \), with

\[
s' = s - \langle c, i, m, f \cup \{ j \} \rangle + \langle c, i, m + t^*, f \rangle - \langle t, j, m_{\rho(t)}(\emptyset) \rangle.
\]

The straight run in Figure 8 (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 [2] (or any other kind of high-level Petri-nets).

**Definition 6.5 (structure of exception Petri nets)**
An exception Petri-net (EPN) is a PPN \( NS = (N, \rho) \), where each \( N \in C \) is colored Petri-nets with special places \( p^r_N \) and \( p^p_N \), and a special transition \( t^p_N \) such that \( \rho(p^p_N) = \emptyset = \rho(p^r_N) \) and \( p^p_N = t^p_N \). A state of an EPN is a multiset of state elements \( C \times 1 \times M \times F \) similar to the state of a PPN, only that \( M \) are now markings of colored nets. □

Continuing the explanation from above, suppose that at some point, \( p^r(\rho(t)) \) may be marked, triggering the event \( t^j_{\text{exit}} \). Now the execution of invocation \( j \) of \( \rho(t) \) may be aborted, transferring the tokens on \( p^r(\rho(t)) \) to \( p^p_N \). Then, either handle deals with the exception, or propagate moves the token to \( p^p_N \), 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 6.6 (procedure exit transitions)**
Instance \( j \) of a refined transition \( t \) may perform a procedure exit in state \( s \) (written \( t^j_{\text{exit}} \)), if it is activated, i.e., its raised-place \( p^p_N \) is marked (formally: \( (c, i, m, f) \cup \{ j \} \rangle \langle t, j, m', \text{sub} \rangle \in s \), with \( m'(p^r(\rho(t))) \geq m \).

When \( t^j_{\text{exit}} \) 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 \( s \xrightarrow{t^j_{\text{exit}}} s' \)), with

\[
s' = s - \langle c, i, m, f \cup \{ j \} \rangle + \langle c, i, m + t^*, f \rangle - \langle t, j, m, \text{calls} \rangle - \text{CALLS},
\]

where \( \text{CALLS} \) is the largest fixed point of the equation

\[
\text{SUB}_{s'}(\text{calls}) = \{ \langle t, j, m, \text{calls} \rangle s' \mid j \in \text{calls} \} \cup \text{SUB}_{s'}(\text{calls}').
\]

**7 Semantic mapping**

Now the mapping from Activities to EPNs is rather straightforward: a specification \( \text{Spec} = \{ \text{Activities, Handlers} \} \) maps into a PPN \( (N, \rho) \), 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 \( p^p_N \). All ActivityEdges with lightning-adornments that raise an exception in the Activity that map into transitions \( t^j \) of \( \rho(t) \)
with $t' = p_r^N$. See Figure 5: the gray net elements result from translating one handler.

At this point, we reuse and adapt earlier results: $\cdot \cdot \cdot$ from [8] maps an Activity into a colored Petri-net, and $\cdot \cdot \cdot$ from [5] maps a set of Activities into a PPN. All we need to do is extend the functions such that they also map ExceptionHandlers, and yield EPNs instead of PPNs.

Three additions are needed (see Figure 6). First, the special places $p_N^N$ and $p_c^N$ and the transition propagate must 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 6 shows the details.

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 7). For every ActivityEdge that enters the InterruptibleActivityRegion, an ad-

---

**Figure 5.** Schema of raising, handling, and propagating exceptions in exception Petri-nets.

**Figure 6.** Mapping an ExceptionHandler.

**Figure 7.** The intuition of the semantic mapping for Activities.
ditional 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 3 were outside the InterruptibleActivityRegion, this translation scheme breaks 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 2. It shows a specification Spec based on the main example from the standard (cf. [4, p. 323], here reproduced as Figure 3), 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 9 and \( \rho \) is \{Process Order \( \mapsto \) N_2, Send Invoice \( \mapsto \) N_3\}. The initial marking of \( E \) is \( \langle \bot, 0, p_0, \emptyset \rangle \), its final marking is \( m_f \) \( \langle \bot, 0, p_3, \emptyset \rangle \). Figure 8 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.

8 Conclusion

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.

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 StateMachines, whose natural semantic domains must be related to Petri-nets.

Figure 9. The result of translating the specification shown in Figure 2.
References


Figure 8. Some runs of the net system in Figure 9, representing the set of Activity Diagrams of Figure 2. The names of the refined transitions have been abbreviated to PO and SI.