# Towards a Formal Semantics of UML 2.0 Activities

Harald Störrle
Institut für Informatik/PST
LMU München
stoerrle@pst.ifi.lmu.de

Jan Hendrik Hausmann
Institut für Informatik
Universität Paderborn
hausmann@upb.de

**Abstract:** The new version 2.0 of the Unified Modeling Language (UML) was targeted at improving expressiveness and semantic precision. These developments are particularly evident in activity diagrams which have not only acquired many new features, but a completely new metamodel and semantic foundation. The UML contains some hints that Petri-nets are the inspirational source for the new semantics. In this paper we will investigate how strong the alignment of UML's activity diagrams to Petri-nets really is. We start by providing a mapping of the basic elements of activity diagrams to Petri-nets and discuss the problems arising when trying to extend this approach to some of the advanced features of activity diagrams, namely exceptions, traverse-to-completion, and streaming. This examination raises several syntactic and semantic questions concerning activities. We conclude that for basic activities, the analogy works pretty well, but for higher-level constructs, no such intuitive connection exists.

## 1 Introduction

Activity diagrams have been added to the UML rather late. They have always been poorly integrated, lacked expressiveness, and did not have an adequate semantics. In UML 2.0, several new concepts and notations have been introduced, e.g., exceptions, collection values, streams, loops, and so on. The intended range of applications spans from the modeling of high-level business processes down to the description of basic computations. The semantics of these constructs is described in natural language by token flow rules inspired by Petri-Nets: "*Activities are redesigned to use a Petri-like semantics*" (cf. [OM03, p. 292]). Thus activity diagrams promise to integrate high-level constructs with a precise and formal semantics. But the specification never substantiates the promise of this integration by providing either a precise formulation of the flow rules or a mapping to Petri-nets.

In [St04b], a semantic mapping of basic activities to P/T-nets has been proposed (see Figure 1 for an overview of this mapping). Figure 2 (a-b) provides an example of this mapping. Obviously, the mapping for the basic constructs is rather straightforward and appeals to intuition. But does this mapping in particular and the mapping to Petri-Nets in general hold for more complicated constructs of activities as well? We tried to answer this question by closely examining possible semantics for some characteristic constructs and concepts in UML Activities, namely exceptions, streaming and traverse-to-completion.

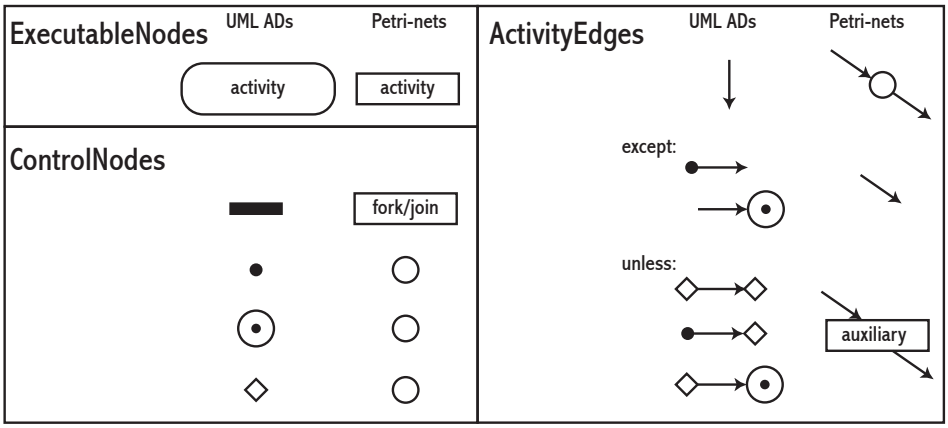


Figure 1: Intuition of semantic mapping from activity diagrams to Petri-nets.

## 2 Exceptions

One of the most prominent additions in UML 2.0 Activities are exceptions. Consider the example in Figure 3 (adapted from the standard). It is supposed to be interpreted as follows: While processing the part of the activity that is enclosed by the dashed line (a so called “InterruptibleActivityRegion”), the reception of an “Order cancel request” event triggers the preemptive abortion of this part of the activity, and continues execution with the “Cancel Order”-Action.

There are a couple of problems with this interpretation that require substantial semantic clarification. For instance, what happens to a parallel thread (i.e., “Send Invoice” etc.) outside the InterruptibleActivityRegion? Also, what happens if the JoinNode is outside the InterruptibleActivityRegion? Should such an activity diagram be considered syntactically correct, and how would such a syntax check be performed without computing the semantics first?

But there is also another, more fundamental question: Exceptions are a prime example of non-local behavior. Abstractly speaking, raising and handling an exception means switching from one of a number of specified program states (read: Action) to some other state in a single step, i.e. a kind of multi-goto. In Petri-nets, on the other hand, states (“markings”) are distributed over the whole net, and state changes are local. So, how can non-local behavior like an exception be modeled using Petri-nets? There is a couple of technical problems to be solved.

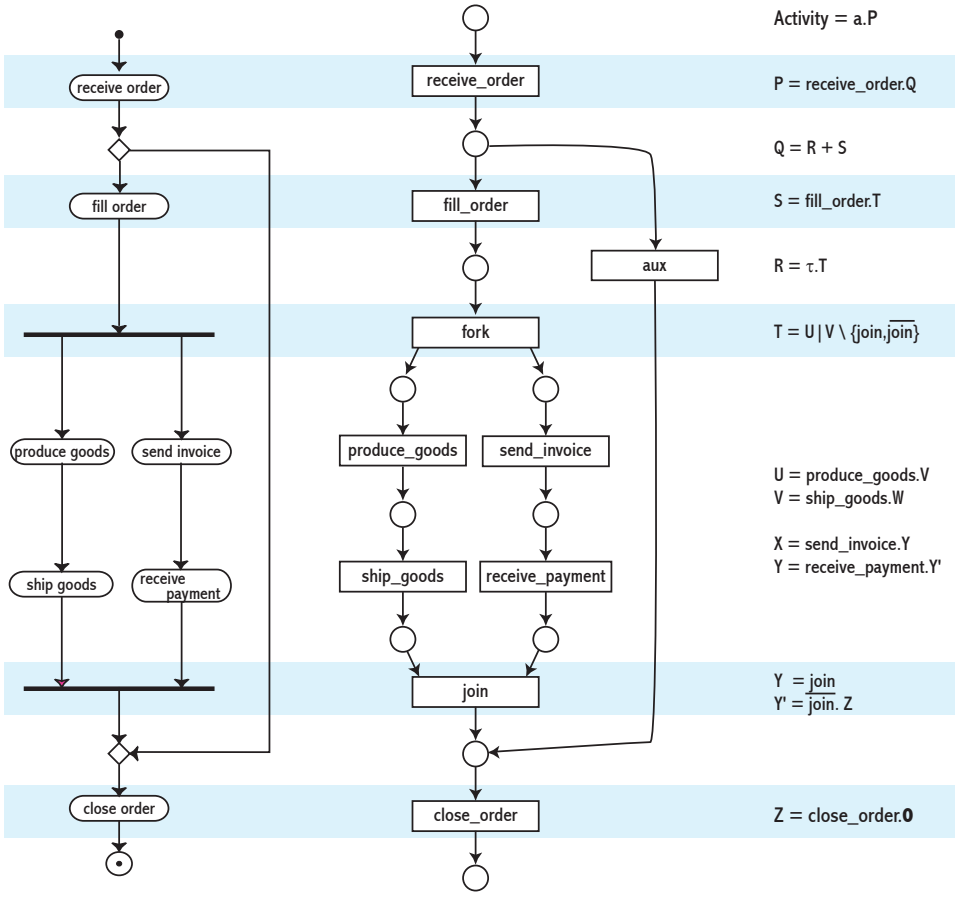


Figure 2: Example for the semantic mapping of Fig. 1, and comparison of Petri-nets to CCS as a semantic domain.

### 2.1 Number of preemptable states

First of all, observe that preempting one of several states means that there are *several* markings, all of which must be removed in the case of an exception raising event. But the number of such markings may grow exponentially when there is concurrency within the image of the InterruptibleActivityRegion under $[\![\_]\!]$.[1] Now, there are two ways how the relevant set of markings may be removed.

First, one might compute the reachable set of markings during a compile run, and create

[1]For the remainder of this discussion, we assume, that there is a semantic function $[\![\_]\!] : \mathsf{A} \rightarrow \text{Petri-net}$ along the lines of the semantic function defined in [St04b], where A is an Activity or fragment of an Activity.

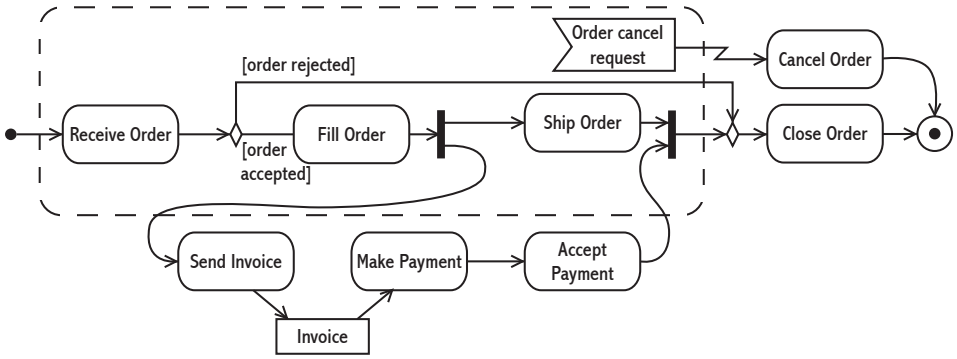


Figure 3: ExceptionSyntax

one transition to remove each of them. That is, in order to translate an Activity A into a Petri-net ⟦A⟧, the resulting Petri-net ⟦A⟧ must be executed. Obviously, this is slightly inconvenient.

Second, one might create some net structure that ensures that all possible distributions of tokens over places are covered, not just the reachable markings. This in turn could be done in either of three ways (see Figure 4).

1. For simplicity, assume that we are dealing with 1-safe nets, that is, there can always be at most one token on each place. Then, by introducing one transition for each subset of the places in the net-fragment representing the InterruptibleActivityRegion, we are sure to be able to remove all possible markings, including the reachable ones. However, the number of transition grows exponentially with the size of the InterruptibleActivityRegion, and also, one would have to introduce priorities between transitions to make sure that all tokens that are removable are actually removed: in Figure 4 (a), $\mathsf{Preempt}_1$ might fire, even if $\mathsf{Preempt}_2$ is also activated.

2. Another approach might be to introduce constructs like flush arcs, i.e., arcs that are capable of removing all tokens from a place irrespective of their number. In Figure 4 (b), the asterisk on the arcs is used to denote a flush arc.

3. Yet another approach is to resort to high-level nets in general, and model the required kind of behavior using the capabilities of some clever inscription language. A typical pattern would be to use “list of tokens” rather than “tokens” as the color (i.e., type) of such places (see Figure 4 (c), [] is used to denote the empty list).

Either way, the nice and direct correspondence between Activities and Petri-nets is lost. All three approaches would also increase the expressive power of the underlying type of Petri-nets to being Turing-equivalent. The third solution would also lose some concurrency. So, summing up: all possible solutions have their drawbacks. The solution proposed in [St04c], on the other hand, avoids all of these drawbacks—at the cost of resorting to a customized and rather arcane net formalism.

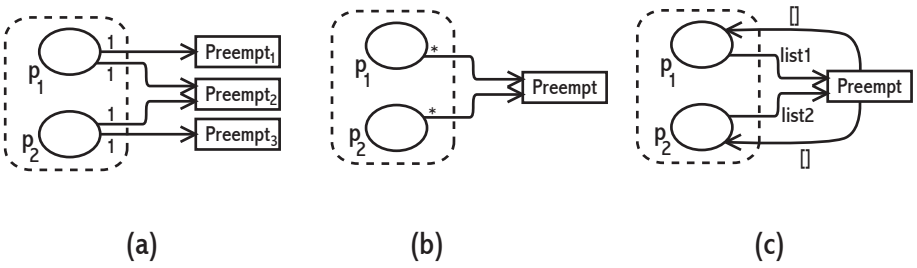


Figure 4: Possibilities for preempting a subnet: an individual transition for each marking (a), flush-arcs (b), and high-level nets with list-typed markings (c).

### 2.2 Size of the InterruptibleActivityRegion

A problem more related to the standard proper rather than the semantic domain is: where is the boundary of an InterruptibleActivityRegion? Consider again the net in Figure 5. Doubtlessly, the tokens in those places that are colored in dark gray ($p_1$, $p_2$, and $p_7$) should be removed when the trigger to the exception arrives. But what about the tokens in those places colored in light gray ($p_0$, $p_3$, $p_6$, $p_8$, and $p_9$)? They might be considered too, for different reasons.

It makes no sense to preempt tokens in $p_0$ from an application point of view. But the initial state of an Activity is not really a state as such but an adornment of the Action it is leading to, expressing, that this is the first action to be executed. So, what does it mean to have the initial state outside the InterruptibleActivityRegion? Or rather, when does an Activity start—when it executes its first action, or when it is instantiated? For doubtlessly, the metaclass Activity should be instantiated to represent a concrete behavior, as the Metamodel-Architecture of the UML suggests.

A similar argument applies to fork and join nodes, i.e., $p_3$, $p_6$, and $p_9$: when exactly is the flow of control split and merged again? Yet another, and even more problematic case is posed by $p_8$: if it is not considered part of the InterruptibleActivityRegion, an exception might result in an unfinished thread of control when the final state is reached. Is this admissible? If so, how will the dangling token in $p_8$ be removed?

The problem described in this section is both a semantical and a syntactical problem. In [St04c], a pragmatic solution has been proposed. The OMG has not adressed this issue so far, and it is currently unclear, which stance it will take.

## 3 Streaming

As a second probe, consider streaming (see Figure 6 for an introductory example). A naive translation according to Fig. 1 leads to several problems. The first problem is ordering. The

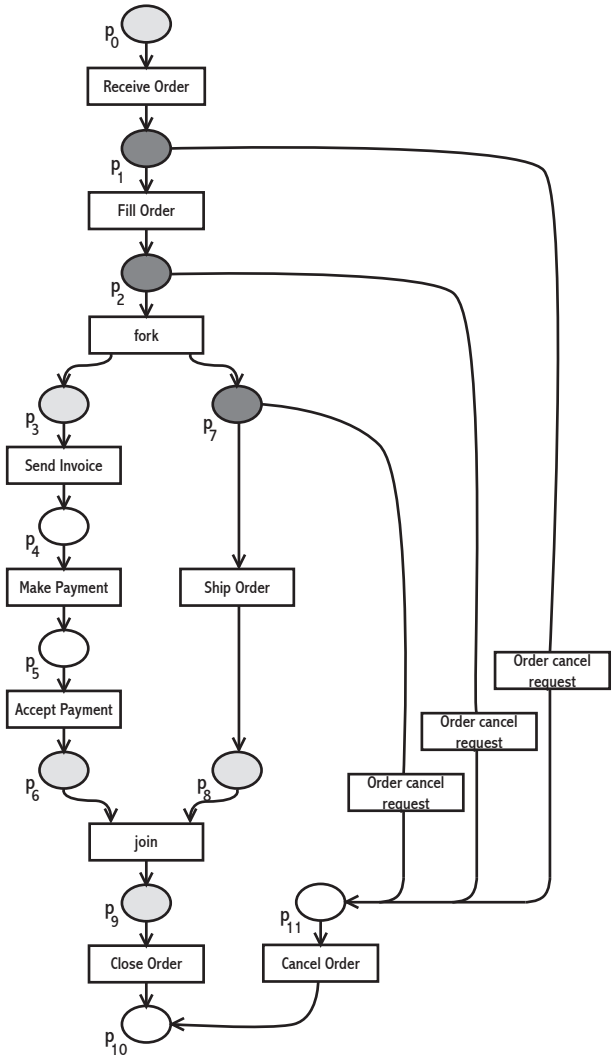


Figure 5: Naive Petri-net semantics of the exception example in Figure 3.

intuition of a stream suggests that the ordering of the elements of a stream is to remain stable. In a plain Petri-net, however, places are unordered bags of tokens. In order to preserve the order among them, the place would have to be turned into a FIFO-buffer. This could be done either by adding FIFO-places as a native construct (i.e., by extending the net formalism, again), by using lists to represent streams as in Figure 7 (a), or by using explicit counters to ensure the right sequence as in Figure 7 (b). The second solution would loose some degree of concurrency, and both the latter would require high-level nets—another

net formalism *again*.

The second problem with streaming is isolation: if several streams are passing through an Activity, should these streams be isolated from each other, or should they interact? And how do we achieve either?

Consider a family of streams $\mathbb{S}$ being processed by some Activity $A$. At any point during processing, the set of stream elements currently processed by $A$ contains at most one such element for each stream. Call this set a slice of $\mathbb{S}$. Isolation is thus the question whether the elements of a slice may influence each other.

If an Action is refined in a macro-expansion way, there is no isolation, and arbitrary interactions may occur. Consider the example of Figure 6: inputing several streams of frames at the same time to Audio/Video processing may result in arbitrary order-preserving selection and interleaving of frames. Completely forbidding all interactions is no solution either, since this would disallow even the simplest of operations like split AV and join AV. So, one might introduce additional constraints that prevent interference where it is not wanted, as proposed in [St04d] (see Figure 8). A macro-expansion, however, would loose the intuition of a transition as an atomic entity.

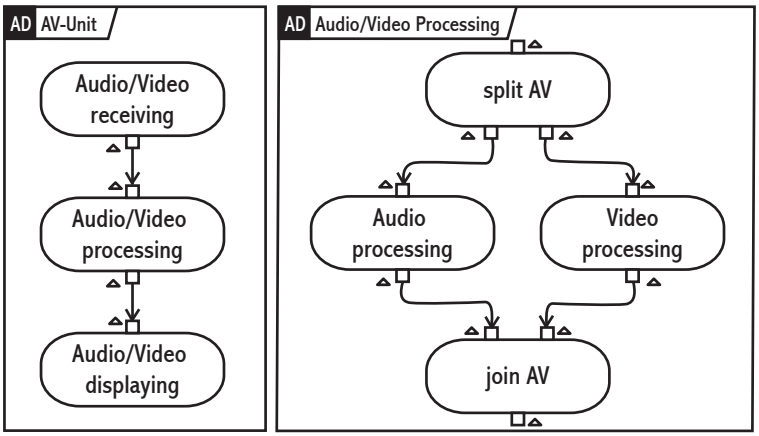


Figure 6: Example of streaming activities.

## 4 Traverse-to-completion semantics

A characteristic syntactic property of Petri-nets is that they are represented by bipartite graphs of places and transitions. Activity diagrams also distinguish between two types of nodes: Executable and Object nodes both of which may hold tokens on the one hand, and on the other hand control nodes, which cannot hold a token but just determine the way tokens flow between Executable and Object nodes. Control nodes can, however, also be connected to other control nodes. A single token flow might thus entail traversing

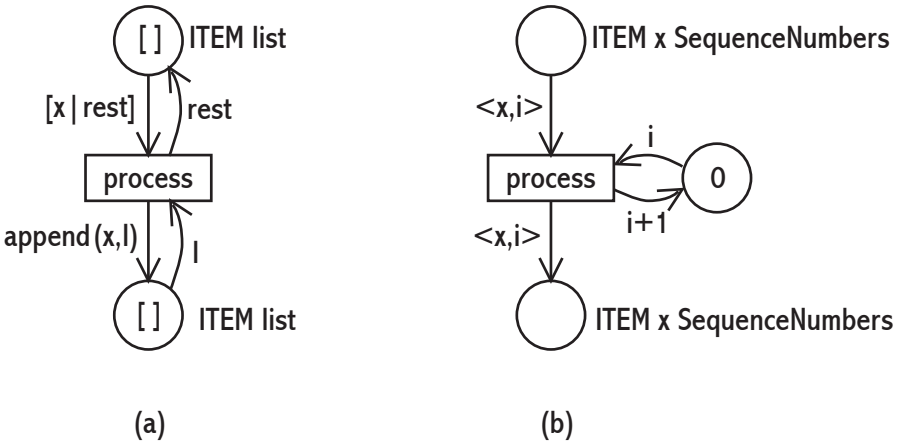


Figure 7: Different Petri-net semantics for streaming: streams as lists (a), streams by sequence numbering (b).

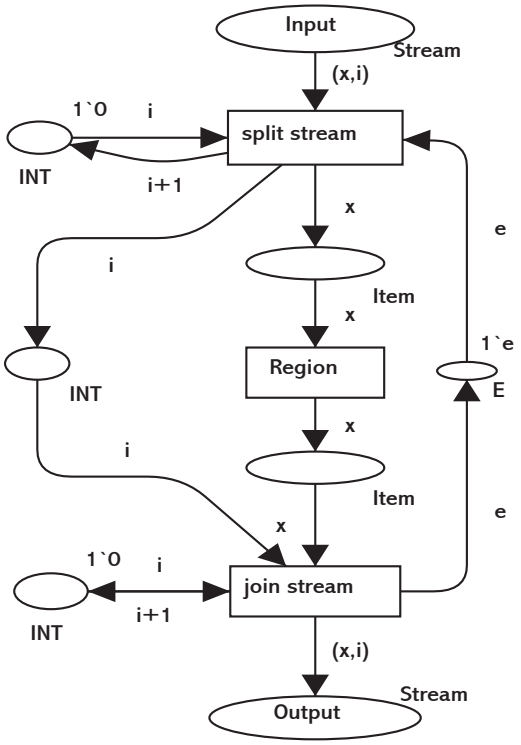


Figure 8: A more complex semantical approach to streaming: only a single stream is channeled into the region. The inscriptions are the formalism used in the CPN toolset (cf. [CP04]).

numerous control nodes and their connecting edges.

To still retain a ”Petri-like“ synchronization mechanism, the semantics of activity diagrams follow the traverse-to-completion principle (ttc) [Bo04].[2] According to ttc, tokens will only leave their current position and move on to another node if the whole path to the destination node is open. The evaluation of the path is described by the term ”offering“. A token is offered to the adjacent edges of its current position and these offers spread through the activity graph until one of them is accepted. In situations with token competition (e.g., multiple outgoing edges from an object node), the first accepted offer determines which way the token moves. With ttc the whole path (though constructed form numerous elements) acts like a single Petri-net transition. If a token can only move down one way, this semantics presents no observable difference to a stepwise token passing. In situations with token competition however (e.g., multiple outgoing edges from an object node) ttc prevents token from getting stuck in the middle of a path since the token moves down the first path that can accept it completely. In the example in Fig. 9 a token provided by action B will thus never flow toward the join node unless actions A and X can provide tokens as well.

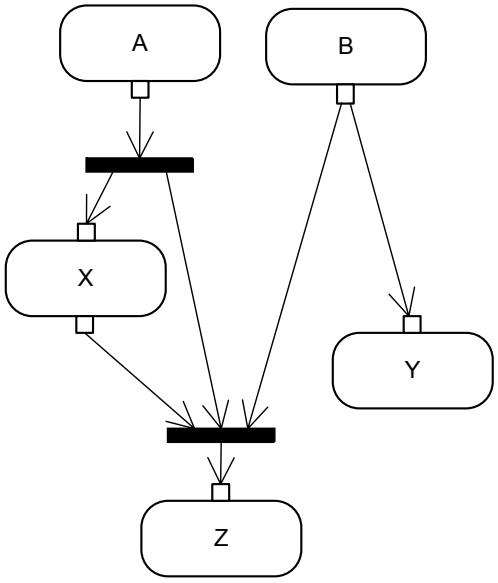


Figure 9: Example activity diagram

[2]This principle is not a concept of the standard proper, but since it is proposed by one of the main persons behind this part of the standard, it may be assumed that this is, to some degree, the official position of the OMG, too.

In combination with forks and joins, ttc presents some problems. According to the adopted UML 2.0 specification, fork nodes can only pass tokens, if all outgoing paths can accept the token. Join nodes require all incoming edges to offer tokens before firing. If we regard the example activity diagram in Fig. 9 we can observe that under this semantics the left hand part of the diagram has no valid execution due to the following circular dependency: the fork node requires the join node to accept a token, the join node requires a token to be offered from action X which it can only do if the fork node provides its input token. Due to this observation, the semantics of the fork node will presumably be relaxed in the finalization process (cf. [Ob04], issue 7221) to require only one outgoing path to accept its token. Copies of the passing token will be stored for the non-successful outgoing edges.

If we perform an intuitive element-wise mapping of activity diagrams to Petri-nets (as described in Figure 1) we are faced with the dilemma that Petri-nets have a strictly local synchronization, but ttc semantics require synchronization beyond this scope. There are two possible ways out of this dilemma: either we do map whole structures to single transitions or we use—once again—a semantically richer Petri-net variant, e.g., Zero-safe nets (cf. [BM]).

As already stated above, sometimes, whole structures in activity diagrams behave like single transitions in Petri-nets. We could thus discard the simple mapping presented above and devise a more complicated compilation approach which evaluates the different possible flows through a control structure of an activity diagram and maps these to simple Petri-net transitions (synchronising all inputs and outputs). But to actually provide such a general compilation mechanism, all possible feature interactions in activity diagrams have to be understood. That is, we have to know the precise semantics for all cases and embed that knowledge in the transformation. We cannot assume such a knowledge a priori.

Zero-safe nets provide a notion of transactions over parts of a net by distinguishing between so calles ”stable places“ and ”zero places“. A marking in which all tokens are on stable places is called a stable marking and is supposed to represent an observable state of the net. From such a stable marking, tokens may flow according to the usual rules on either stable or zero places. Markings where tokens rest on zero places represent internal states. A commit rule ensures that all tokens for stable places produced from internal markings are placed at once and only if no more tokens rest on any zero place. The net thus has a notion of transactions between the stable markings, which is internally specified by a number of transitions.

While a detailed mapping of activity diagrams to zero-safe nets has not yet been done, it seems obvious that a) these nets provide the kind of mechanism needed to express ttc and that b) most control nodes would have to map to zero places. Thus irrespective of the actual details of such a mapping, we are once again faced with the situation to not only choose a new or extended semantical domain but also to tweak the intuitive mapping. This is especially harmful for the case of ttc since it is a general semantic principle which is supposed to hold for *all* activity diagrams. Since there are cases, in which there exists an observable difference between ttc and the interpretation given by our basic mapping, we need to be aware that the basic mapping itself is not entirely adequate.

## 5 Conclusions

In this paper we have examined some semantic questions concerning UML activities by trying to provide a mapping from activities to Petri-nets. Summarizing our investigation we have to state that while the mapping of basic UML activity diagrams to elementary Petri-nets is quite simple and intuitive, this is not the case for exceptions, streaming and traverse-to-completion. Not only do these three cases require different Petri-net variants as their semantics domain, but they also impose modifications to the basic mapping, thereby breaking the basic intuitions to a certain extent.

Furthermore, it is not obvious that combining all the various target domains is trivial. In particular data-flow and procedure calling (cf. [St04a] and [St04b], respectively) also need to be taken into account. Even if the construction of such a unified formalism was possible, no analysis tools or theoretic results would be readily available for it.

On the other hand, for other semantical domains (e.g. CCS and algebraic approaches in general), even the basic mapping is unintuitive (cf. Figure 2). A translation from activities directly into, say, partial words or transition systems would cover a huge distance between syntax and semantics, thus suffering from the same problem.

Different conclusions can be drawn from this result: On the one hand one we could state that the problem lies with the semantic foundations, that is, the formal methods as we know them now. Thus, further scientific work might be necessary to find a semantical domain which is able to express the semantics of UML activity diagrams in a concise and intuitive way. A special unified Petri-Net variant or other formalisms might serve this way. However, such a formalism would be a kludge, and the original appeal of Petri-nets is lost, so what is the point in demanding a "*Petri-like*" semantics in the first place?

On the other hand one might raise the question whether the misalignment of activity diagrams to Petri-Nets is not a shortcoming of the standard. If Petri-Nets are indeed supposed to serve as an intuitive semantic domain, the authors of the UML should at least make sure that a) this mapping is being made explicit and that b) high-level extensions must be compatible to the degree that they do not break the intuition provided in the basic mapping. Without these measures, the claimed alignment of activity diagrams to Petri-Nets remains rather superficial and does not yield any benefit.

## References


[BM] Bruni, R. und Montanari, U.: Transactions and Zero-Safe Nets. volume 2128 of *LNCS*. p. 380ff.

[Bo04] Bock, C.: UML 2 Activity and Action Models: Object Nodes. *J. Object Technology*. 3(1):27–41. January/February 2004. available at `www.jot.fm`.

[CP04] CPN Tools Team: CPN Tools Manual. Technical report. Univ. of Aarhus. 2004. available at `http://wiki.daimi.au.dk/cpntools/cpntools.wiki`.

[Ob04] Object Management Group. Issues Database. 2004. available at `www.omg.org/issues`.

[OM03] OMG: OMG Unified Modeling Language: Superstructure (final adopted spec, version 2.0, 2003-08-02). Technical report. Object Management Group. November 2003. Available at `www.omg.org`, downloaded at November $11^{\text{th}}$, 2003.

[St04a] Störrle, H.: Semantics and Verification of Data-Flow in UML 2.0 Activities. In: Minas, M. (Ed.), *Proc. Intl. Ws. on Visual Languages and Formal Methods (VLFM'04)*. pp. 38–52. IEEE Press. 2004. available at `www.pst.informatik.uni-muenchen.de/~stoerrle`.

[St04b] Störrle, H.: Semantics of Control-Flow in UML 2.0 Activities. In: Bottoni, P., Hundhausen, C., Levialdi, S., und Tortora, G. (Eds.), *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. pp. 235–242. Springer Verlag. 2004.

[St04c] Störrle, H.: Semantics of Exceptions in UML 2.0 Activities. Technical Report 0403. Ludwig-Maximilians-Universität München, Institut für Informatik. 2004.

[St04d] Störrle, H.: Semantics of Expansion Nodes in UML 2.0 Activities. In: Porres, I. (Ed.), *Proc. $2^{nd}$ Nordic Ws. on UML, Modeling, Methods and Tools (NWUML'04)*. 2004.