

Semantics of Control-Flow 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 [23] has introduced significant changes and additions. In particular, the metamodel portion underlying Activity Diagrams has been completely reengineered, with Activity being the central concept, the successor of ActivityGraph in UML 1.5. In this paper, a denotational and compositional semantics for Activities is defined as a mapping from Activities into Procedural Petri nets [21]. The semantics excludes data type annotations and all features based on them, but includes all kinds of control flow, including non well-formed concurrency and, particularly, procedure calling.

Keywords: UML 2.0, Activity Diagrams, denotational and compositional semantics, modeling of web-services, workflows, and service-oriented architectures

1. Introduction

1.1. Motivation

Modeling of business processes and workflows is an important area in software engineering, and, given that it typically occurs very early in a project, it is one of those areas, where model-driven approaches definitely have a competitive edge over code-driven approaches. Activity diagrams have been introduced into the UML rather late. It has since been considered mainly as a workflow definition language, but it is also the natural choice when it comes to modeling web-services, and plays an important role in specifying system-level behaviors. The UML has become the “*lingua franca of software engineering*”, and it has recently undergone a major revision (advancing from version 1.5 to version 2.0), including a complete redefinition of Activities. Unfortunately, the standard has yet again failed to define a formal semantics, as would be necessary to take full advantage of the UML, e.g., in automated tools.

Compared to UML 1.5, the concrete syntax of Activity Diagrams has remained mostly the same, but the abstract

syntax and semantics have changed drastically. While in UML 1.5, Activity Diagrams have been defined as a kind of State Machine Diagrams (ActivityGraph used to be a subclass of StateMachine in the Metamodel), there is now no such connection between the two, and the meaning of Activity Diagrams is being explained in terms of Petri net notions like token, flow, edge weight and so on.

1.2. Approach

Given this emphasis, it is natural to use Petri nets as the semantic domain. However, Activity Diagrams also feature procedure-call like structuring and data-type related inscriptions. While the latter is more or less a standard extension to the basic Petri net model (“higher-order nets”) with many different approaches ([20, 18] being the most well-known), procedure-call behavior is less well covered in Petri net theory. Also, many of the existing semantics of UML activity diagrams (UML 1.5, nota bene) exclude this feature, and all Petri net based semantics do so. Hence, this is where the focus of this paper lies.

There are many approaches towards hierarchically structured nets (e.g. [16, 20, 30]), but they generally either are macro-like, and would have to simulate procedure call behavior rather awkwardly via complex inscriptions, i.e. higher-order Petri net constructs. Thus, a better starting point would be a net formalism that offers procedure-call-like features directly. Such a formalism has been proposed in [21], and here we use a simplified version of it.

1.3. Related work

While there is a rather large body of work on UML 1.x Activity Diagrams, it seems that so far, hardly any work (except [3]) has been published on the UML 2.0 Activity Diagrams—and the UML standard has been written from scratch as far as Activity Diagrams are concerned. Yet, some considerations and ideas of previous works still apply, and so it is worthwhile looking at the previous work

on UML 1.x Activity Diagrams, too. Here, there are four distinct categories of contributions.

First, there are “pragmatic” approaches that look into the pragmatics of Activity Diagrams, examining their usage either for comparing the expressive power of Activity Diagrams with that of (commercial) workflow description languages and workflow management systems (e.g. [9, 1, 15]), or for examining their methodological relationship to other diagram types of the UML (e.g. [25]). The main contribution of these approaches lies in exploring the potential of Activity Diagrams for certain purposes, and to interpret and develop the standard in such a way that the specific requirements of some particular purpose are better realized. While indeed discussing some semantic issues in doing so, the approaches in this category do not provide a semantics. Also, the new standard now defines the semantic model in mind—Petri nets—and lists the intended usage areas of Activities, namely “*procedural computations*”, “*workflows*”, and “*system level processes*” (cf. [23, p. 284]). Thus, considerations as to the appropriateness of particular semantic domains or about the modeling requirements of one way of modeling versus the other have now become obsolete.

Second, there are some approaches (e.g. [26]) that treat Activity Diagrams as a subclass of StateMachines, as declared by UML 1.x. This has always been a controversial issue, and has now disappeared from the standard.

Third, there are “operational” approaches, defining the meaning of Activity diagrams “by interpreter”, that is, to give them a meaning in terms of some execution mechanism such as a (commercial) workflow execution system, or a more or less formal execution algorithm (cf. [13, 12]) or analysis procedure (cf. [22, 10, 29, 11, 14]).

Fourth, there are those contributions that define a formal semantics in the proper sense, that is, some kind of mapping from Activity Diagrams or ActivityGraphs into some formal domain, e.g. [2, 7, 6, 5, 10, 11, 17]. These approaches may be categorized along the following three axes:

domain the semantic formalism into which Activity Diagrams are mapped;

rigor the degree of formality employed in defining the mapping, ranging from a set of examples to a mathematical function;

expressiveness the degree of coverage of Activity Diagram notions that is mapped, i.e., control-flow, non well-formed control flow, data flow, and hierarchy (flat/macro-expansion-style vs. procedure call).

See Table 1 for a comparative categorization of the contributions of the last two categories.

Let’s look at some of the contributions of the fourth category in more detail. Gehrke et al. [17] also use Petri nets as their semantic domain, but interestingly, they use places to represent ActivityStates (Activities in UML 2.0), possibly

misled by the passive sounding name in the UML 1.5 metamodel. Their work being focused on other issues, activity diagrams are treated only in passing and many interesting features are left out, including hierarchy.

Börger et al. [7] use Abstract State Machines as their semantic domain, and this is the only other semantics that treats SubActivityStates, i.e. procedural calling of Activities. However, the mapping is only given by example, based on concrete syntax and excludes non well-formed control flow. And, of course, it is based on UML 1.3, not on UML 2.0.

Eshuis and Wieringa [10, 11] have published a stream of papers dealing with various aspects of Activity Diagrams, including a kind of operational semantics using labeled transition systems. Again, hierarchy is left out.

2. Activity Diagrams in UML 2.0

In this section, we discuss Activities in UML 2.0 with a particular emphasis on the differences to UML 1.x.

2.1. Concrete syntax

The concrete syntax of Activity Diagrams is changed only slightly. One notable extension is the flexibility now provided by swim lanes, which are close to simulating a kind of use case maps in UML 2.0 (cf. [8]). It does not influence the behavior of an activity, however, and may thus be ignored here. SubactivityStates have vanished, and nesting is now accomplished by calling subordinate Activities from the Actions that define the behavior of superordinate Activities.

2.2. Abstract syntax

The metamodel for Activities has been redesigned from scratch in UML 2.0. The main concept underlying Activity Diagrams is now called Activity and “*replaces ActivityGraph in UML 1.5.*” (cf. [23, p. 263]). Activity is not a subclass of StateMachine any more.¹ The metamodel defines six levels increasing expressiveness. The first level (“BasicActivities”) already includes control flow and procedurally calling of subordinate Activities by ActivityNodes that are in fact Actions (see Figure 3), the second level (“IntermediateActivities”) introduces data flow. This paper is restricted to BasicActivities.

The basic two entities are Actions and Activities. While an Action “*is the fundamental unit of executable functionality*” (cf. [23, p. 280]), an activity provides “*the coordinated sequencing of subordinate units whose individual elements*

¹ Among other things, this considerably enhances the expressive power of Activity Diagrams, since the Activity in Figure 2 (right) is legal in UML 2.0, but not in UML 1.x.

authors, references	semantic style	semantic domain	expressiveness			rigor
			control flow	data flow	hierarchy	
Apvrille et al. [2]	by algorithm	LOTOS	wf	–	–	medium
Börger et al. [7]	denotational	ASM	wf	–	✓	medium
Bolton & Davies [6, 5]	by compiler	CSP	wf	–	–	low
Eshuis & Wieringa [10, 11]	informal	algorithm	wf, nwf	–	–	high
Eshuis & Wieringa [13, 12]	by algorithm	LTS	wf, nwf	–	–	high
Gehrke et al. [17]	by example	PN	wf, nwf	(–)	–	medium
Rodrigues [29]	informal	FSP	wf	–	–	low
Li et al. [22]	by algorithm	LTS	wf	(–)	–	high
this paper	denotational	procedural PN	wf, nwf	–	✓	high

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

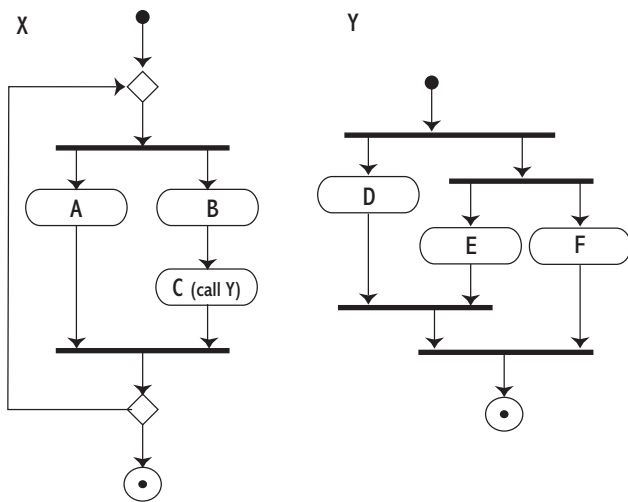


Figure 2. Two sample Activity Diagrams: Action C of Activity X (left) calls Activity Y (right).

are actions” (cf. [23, p. 280]). This coordination is captured as a graph of ActivityNodes connected by ActivityEdges (see figure 3). Since “there are actions that invoke activities” (cf. [23, p. 280]), Activities may be nested: “Activities may form invocation hierarchies invoking other activities, ultimately resolving to individual actions.” (cf. [23, p. 284]) And “Actions have no further decomposition in the activity containing them.” (cf. [23, p. 284]).

For convenience, we assume that an Activity is presented to us as a labelled graph, i.e., in the structure

$$\langle \text{Name}, \langle \text{ActivityNodes}, \text{ActivityEdges} \rangle \rangle,$$

where ActivityNodes and ActivityEdges are the nodes and

edges of the Activity, which the standard defines to be a graph in the mathematical sense. The functions *name* and *graph* to extract the first and second component of the elements of such a pair, and the function *nodes* extracts the first component of the second component. We use the dot-notation to extract attributes and associations from classes, like in *Action.call* and *Activity.name* (cf. Figure 3).

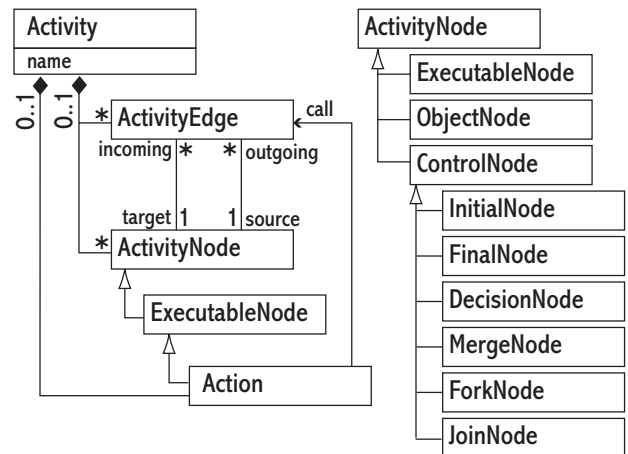


Figure 3. A small portion of the UML 2.0 meta-model: Activities either have Actions or a graph of ActivityNodes and ActivityEdges (left); kinds of nodes and edges (right).

2.3. Semantics

The semantics has changed even more than the abstract syntax: Activities now “use a Petri-like semantics instead of state machines.” (cf. [23, p. 263]). Unfortunately, the

standard does not elaborate on this promising statement. So, the semantics defined in this paper is also an attempt to fill in the blanks.

3. Semantics of control flow

In this section the formal semantics of basic control flow is defined, that is, sequencing, branching, and concurrency. For simplicity, it is assumed that every Activity contains only a single FinalNode, there are no ConnectorEdges, and SendSignalActions (cf. [23, pp. 218, 263]) and AcceptEventActions are not represented graphically but as inscriptions. Also, it is assumed that merging control flows is always properly modeled by a MergeNode (see the first ControlNode of X in Figure 2).

Other constructions like those presented in Figure 4, on the other hand, are admissible, if of little practical value.

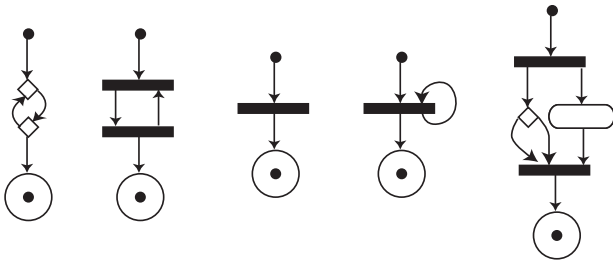


Figure 4. Even freak examples are translatable, although the results do not necessarily make more sense than the original Activity Diagrams.

3.1. Semantic domain

The standard states that an “Action is the fundamental unit of behavior specification” (cf. [23, p. 203]). Actions are also called “primitives”. Thus, for the semantic mapping, Actions are assumed to be atomic, i.e., we define the domain A of actions.).

The standard declares that “Activities are redesigned to use a Petri-like semantic” (cf. [23, p. 281]). Thus, we define the domain PN of (complete) Petri net is a tuple $\langle P, T, A, \overline{m}, \underline{m} \rangle$ where P , T , and A have the usual meaning, and \overline{m} and \underline{m} are its initial and final markings (see the Appendix for the complete definition).

Interestingly, the standard imposes a strong fairness conditions on Activities: “An action execution need not proceed immediately upon capturing its input tokens, but it will eventually begin execution.” (cf. [23, p. 253]). This is of course not necessary—for Petri nets, typically a progress notion is sufficient (i.e., if anything may happen, something will).

3.2. Semantic mapping

For basic Activities, the mapping 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. See Figure 5 for an intuitive account of the translation.

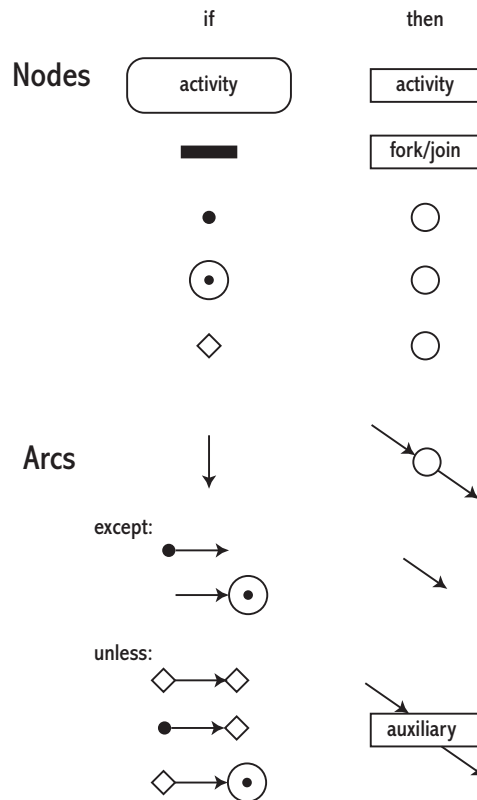


Figure 5. The semantic mapping for basic Activities.

The formal semantics is also straightforward. We assume that all Activities are available as named graph structures, i.e. as tuples $\langle Name, Nodes, Edges \rangle$ where $Name$ is the name of the Activity, and $Nodes$ are its ActivityNodes, partitioned as $Nodes = \langle EN, iN, fN, BN, CN \rangle$ with

- EN the set of ExecutableNodes (i.e. elementary Actions),
- iN, fN the InitialNodes and FinalNodes (of which there may be only one),
- BN the set of branch nodes, including both MergeNodes and DecisionNodes, and

CN the set of concurrency nodes, subsuming ForkNodes, JoinNodes and ForkJoinNodes.

$Edges$ is the set of ActivityEdges between all of these ActivityNodes. Note that this covers all ActivityNodes and ActivityEdges except those dealing with data flow. The translation for a basic Activity Diagram is thus

$$\llbracket \langle Nodes, Edges \rangle \rrbracket = \langle P, T, A, \overline{m}, \underline{m} \rangle$$

where

$$\begin{aligned} P &= \{iN, fN\} \cup BN \cup \\ &\quad \{p_a \mid a \in Edges, \{a_1, a_2\} \cap (EN \cup CN) \neq \emptyset\}, \\ T &= EN \cup CN \cup \\ &\quad \{t_a \mid a \in Edges, \{a_1, a_2\} \subseteq BN \cup \{iN, fN\}\}, \\ A &= \left\{ \begin{array}{l} \langle x_{\langle from, to \rangle}, to \rangle, \\ \langle from, x_{\langle from, to \rangle} \rangle \end{array} \mid \begin{array}{l} \langle from, to \rangle \\ \in Edges \end{array} \right\}, \\ \overline{m} &= iN, \\ \underline{m} &= fN. \end{aligned}$$

When using nodes and edges as transitions, places and arcs, we refer to their names, of course. We use the variable x to denote elements of $X = P \cup T$, i.e., any Petri-net node. We use the shorthand a_1 and a_2 to denote the first and second element of a pair. Observe that ActivityEdges are used as indexes for names for some PN places—this might seem awkward at first, but it simplifies the definition. For the resulting Petri net, the tuple is treated as an atomic symbol, and just serves as an index to create a unique name. See Figure 6 for the translations of the Activity Diagrams in Figure 2. Observe that these are two separate translations for two separate Activities. The fact that an Action of X will call Y is not captured at this point.

The justification for this translation is that if an Event E occurs for some Activity A , this event occurs for no reason perceivable from within A —the cause of E is outwith of A . Conversely, if E were caused in A and signalled to some other Activity, there is no effect of E perceivable within A .

4. Semantics of invoking Activities

4.1. Semantic domain

The standard defines CallActions which are supposed to call upon Behaviors such as Activities. In the Petri net domain, there are basically two choices for representing this. First, one may use traditional net morphisms (e.g. [16]) or static expansion of, say, transitions by subnets (e.g. [30]). Now, either, multiple concurrent calls to the same refinement are prevented (e.g., by requiring 1-safe nets), or there will be interactions between them. In order to avoid this, one would have to attach call-identifiers to all tokens in a “called” subnet, i.e., one would have to use higher-order

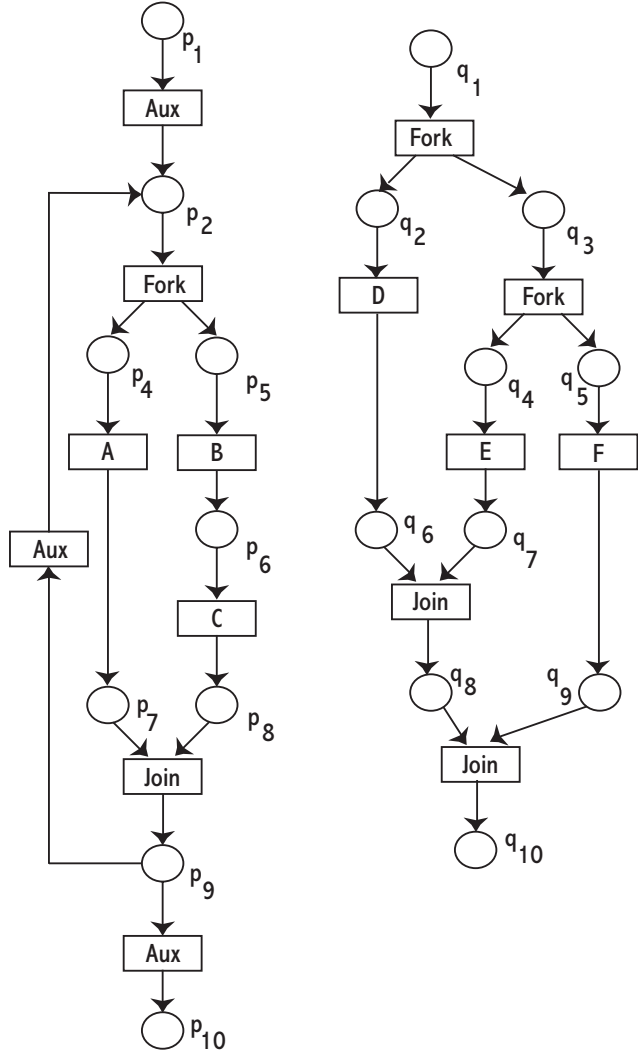


Figure 6. The Petri net $\llbracket X \rrbracket$ and $\llbracket Y \rrbracket$ representing the Activities X and Y of the Activity Diagrams in Figure 2.

nets (cf. [20, 18]) for simple control flows already. This awkward workaround would also result in infinite nets.

But there is a better way. Kiehn [21] has proposed a simple mechanism to integrate a procedure-call mechanism into elementary nets. Better still, it is orthogonal to higher-order Petri nets, so that this kind of extensions may be added later on (in fact, it has been done, but can not be shown here due to lack of space). So, we introduce the domain PPN of procedural Petri nets. The following definitions are simplified versions of the respective notions put forward in [21]. (The basic definitions of Petri-nets are added as an Appendix to this paper).

Definition 4.1 (structure of procedural Petri nets)

A pair $NS = \langle N, \rho \rangle$ is a procedural Petri net (or procedural net, for short), iff N is a finite set of complete Petri nets, and ρ is a partial function $\rho : T_N \rightarrow N$ from transitions of the nets of N into N .

A state element of the procedural net NS is an element of $C \times I \times M \times F$, where

- C is the set of callers, defined as $dom(\rho) \cup \{\perp\}$;
- I is an infinite set of globally unique instance identifiers;
- M is the class of markings of the nets in NS ;
- F is the set of current subordinate function calls.

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

Concerning the choice of the symbol \perp , observe that \perp is the root of a tree representing call dependencies, i.e., this root is a kind of least element.

The definition of the behavior of net systems is a little more complicated, as we now need separate rules for normal firing, procedure call and procedure return. We use s and s' to represent states of net systems.

Definition 4.2 (procedure call transitions)

A refined transition t (i.e. $t \in dom(\rho)$) may perform a procedure call in state s , if it is activated (written $s \xrightarrow{t_{call}}$), i.e., there is a $\langle c, i, m, f \rangle$ in s with $m \xrightarrow{t}$, for some values of c , i , and f .

When $s \xrightarrow{t_{call}}$ is activated, it may fire, creating a new instance i' and reaching a new state s' (written $s \xrightarrow{t_{call}^{i'}} s'$), with

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

\square

Definition 4.3 (procedure return transitions)

The instance i' of a refined transition t may perform a procedure return in state s , if it is activated (written $s \xrightarrow{t_{return}^{i'}}$), i.e., it has reached its final marking and there are no more subordinate function calls. Formally, this means that $s \xrightarrow{t_{return}^{i'}}$ iff $\langle c, i, m, f \cup \{i'\} \rangle \langle t, i', \underline{m}_{\rho(t)}, \emptyset \rangle \in s$.

When $t_{return}^{i'}$ is activated in s , it may fire reaching s' and removing instance i' (written $s \xrightarrow{t_{return}^{i'}} s'$), with

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

\square

Definition 4.4 (ordinary transitions)

Let t be an unrefined transition (i.e. $t \notin dom(\rho)$). Then t 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

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

\square

It is now easy to define the classic notions like net process, firing sequence and so on for procedural petri nets. Observe also, that higher-order net dialects can be put on top of this formalism.

4.2. Semantic mapping

At this point, we are able to translate individual activity diagrams into individual nets. In order to capture sets of Activity Diagrams belonging together and calling each other, we also need to extract the calling relationship as follows.

For convenience, we assume that a specification $Spec$ is a set of Activities as defined above. Pragmatically, one of these must be the top level Activity, that is, the one whose initial state is the initial state of the whole set of Activities. It is selected by the function $top(\cdot)$. We assume furthermore, that within each specification, names of Activities are unique, and all Activities are disjoint (i.e., their Nodes and Edges are). Thus we may translate specifications into procedural nets by

$$\llbracket Spec \rrbracket = \langle N, \rho \rangle$$

where $N = \{\llbracket graph(Activity) \rrbracket \mid Activity \in Spec\}$, and ρ is the set of calling relationships in $Spec$, i.e.

$$\rho = \bigcup_{Activity \in Spec} \left\{ \begin{array}{l} A.name \mapsto \llbracket A.call \rrbracket \mid \\ A.call \neq \text{nil}, A \in nodes(Activity) \end{array} \right\},$$

where A stands for an Action that calls some Activity. Recall that in the definitions of Section 3.2, the (names of the) ExecutableNodes EN of an Activity mapped by $\llbracket \cdot \rrbracket$ are used to create the transitions of the resulting net, so that $A.name$ is in fact a transition, whereas $A.call$ is an Activity.

The initial and final state of the procedural net resulting from the translation of $Spec$ are $\overline{m}_{Spec} = \langle \perp, 0, \overline{m}_{top(Spec)}, \emptyset \rangle$ and $\underline{m}_{Spec} = \langle \perp, 0, \underline{m}_{top(Spec)}, \emptyset \rangle$, assuming that there is a unique top-net in $Spec$, denoted $top(Spec)$.

As an example both for the domain of procedural nets and for the mapping, reconsider the two Activities X and Y shown in Figure 2, and take them as a specification $Spec$, that is:

$$\llbracket Spec \rrbracket = \langle \{\llbracket X \rrbracket, \llbracket Y \rrbracket\}, \{C \mapsto \llbracket Y \rrbracket\} \rangle$$

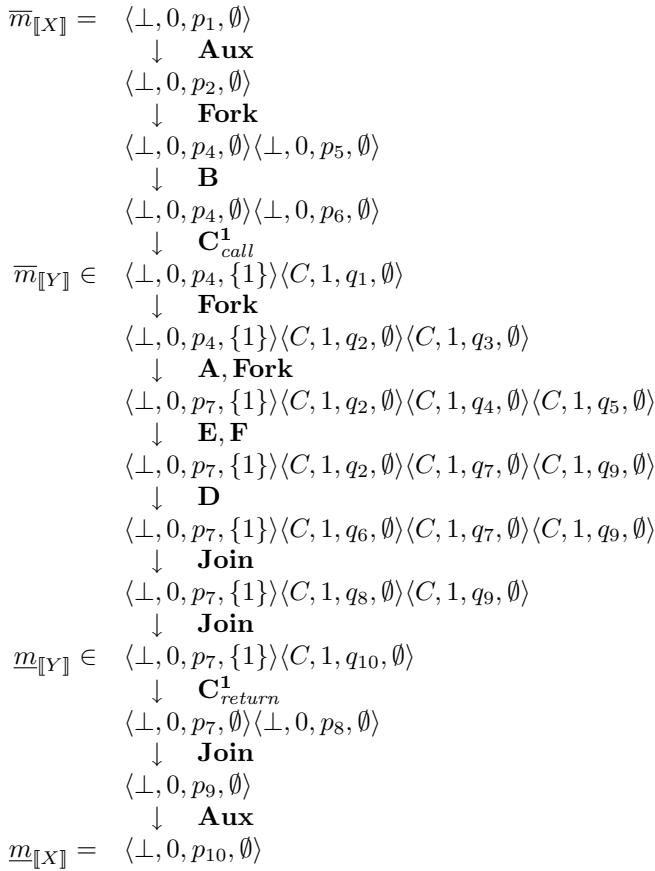


Figure 7. A run of the net in Figure 6, representing the Activity Diagram of Figure 2.

with $\overline{m}_{Spec} = \langle \perp, 0, p_1, \emptyset \rangle$ and $\underline{m}_{Spec} = \langle \perp, 0, p_{10}, \emptyset \rangle$. Figure 7 shows a sample run of the procedural net *Spec* translates into.

As a special case of invocation, the standard declares that “if a behavior is not reentrant, then no more than one invocation of it will be executing at any given time.” (cf. [23, p. 253]). This can be simulated by a run-place for each transition calling a net representing a non-reentrant ExecutableNode (see Figure 8).

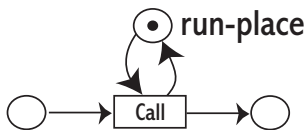


Figure 8. A run-place restricts multiple calls.

5. Conclusion

5.1. Summary

In this paper, a denotational semantics of Activity Diagrams in UML 2.0 is defined. It is based on a procedure call variant of Petri nets (adapted from [21]). The semantics covers control flow, concurrency and procedure call, but not exception handling and data flow, since these require high level Petri nets as their semantic domain.

5.2. Contribution

There have been several proposals for semantics of Activity Diagrams in UML 1.x so far, but none for UML 2.0. Some of proposals however interpreted the old standard rather liberally, and therefore still fit the new standard to some degree. These proposals, however, do not properly cover SubActivityStates, that is, the procedure call mechanism of Activity Diagrams.

5.3. Further work

The work presented here has been complemented along several lines. First of all, the remaining expressive features (data-flow/inscriptions, exceptions, and expansions) have been addressed in other papers (cf. citestoerle:ad:dataflow,stoerle:ad:exception,stoerle:ad:expansion).

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

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 [19], pages 287–301.
- [3] J. P. Barros and L. Gomes. Actions as Activities as Petri nets. In Weber et al. [31], 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. [27], 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] R. J. A. Buhr. Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.
- [9] M. Dumas and A. H. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In Gogolla and Kobryn [19], pages 76–90.
- [10] H. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, CTIT, U. Twente, 2002. Author's first name sometimes appears as “Rik”.
- [11] 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.
- [12] 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.
- [13] R. Eshuis and R. Wieringa. An Execution Algorithm for UML Activity Graphs. In Gogolla and Kobryn [19], pages 47–61.
- [14] 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.
- [15] 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. [31], pages 321–351.
- [16] R. Fehling. *Hierarchische Petrinetze. Beiträge zur Theorie und formale Basis für zugehörige Werkzeuge*. Verlag Dr. Kovač, Hamburg, 1992.
- [17] 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.
- [18] H. Genrich and K. Lautenbach. *Predicate/Transition Nets*. Springer Verlag, 1991.
- [19] M. Gogolla and C. Kobryn, editors. *Proc. 4th Intl. Conf. on the Unified Modeling Language (UML 2001)*, number 2185 in LNCS. Springer Verlag, 2001.
- [20] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. I*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.
- [21] A. Kiehn. *A Structuring Mechanism for Petri Nets*. Dissertation, TU München, 1989. appeared as Technical Report TUM-18902 of the TU München in March, 1989.
- [22] X. Li, M. Cui, Y. Pei, Z. Jianhua, and Z. Guoliang. Timing Analysis of UML Activity Diagrams. In Gogolla and Kobryn [19], pages 62–75.
- [23] 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.
- [24] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall Inc., Englewood Cliffs NJ, 1981.
- [25] 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.
- [26] 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.
- [27] G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa, editors. *Dynamic Behavior in UML Models: Semantic Questions. Workshop Proceedings*, October 2000.
- [28] W. Reisig. *Petri-Nets: an Introduction*. Springer Verlag, 1985.
- [29] R. W. Rodrigues. Formalising UML Activity Diagrams using Finite State Processes. In Reggio et al. [27], pages 92–98.
- [30] I. Suzuki and T. Murata. A method for stepwise refinement and abstraction of petri nets. *Journal of Computer and Systems Science*, 27:51–76, 1983.
- [31] M. Weber, H. Ehrig, and W. Reisig, editors. *Petri Net Technology for Communication-Based Systems*. DFG Research Group “Petri Net Technology”, 2003.

A. Petri nets

Classic references to Petri nets are [28, 24], or more recently [4] (only available in German, unfortunately).

Definition A.1 (Petri nets)

A triple $N = \langle P, T, A \rangle$ is a Petri net, iff P and T are disjoint, and $A \subseteq P \times T \cup T \times P$. 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 $N = \langle P, T, A, \bar{m}, \underline{m} \rangle$ is a complete (Petri-)net, iff $\langle P, T, A \rangle$ is a net, and \bar{m} and \underline{m} are its initial and final markings. 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 $m' = m - \bullet t + t^\bullet$. \square