Semantics of Structured Nodes 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 [21] has introduced significant changes and additions to "*the lingua franca of Software Engineering*". Within the UML, Activity Diagrams are particularly prominent, since they are the natural choice when it comes to the modeling of web-services, workflows, and service-oriented architectures. One of the most novel concepts introduced are so called structured nodes (including loops, collection parameters, and streaming). Building on [29–32], this paper explores the meaning of StructuredActivityNodes, as it is called in the metamodel, by defining them mathematically in terms of procedural colored Petri nets.

Keywords: UML 2.0, Activity Diagrams, structured nodes, loops, collection-valued parameters, streaming, procedural Petri-nets

1 Introduction

1.1 Motivation

The modeling of business processes and workflows is an important area in industrial software engineering, and, given that it crucially involves domain-experts which are usually non-programmers, 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[1] of UML 1.5 have certain shortcomings in this respect, one of which is the lack of structuring mechanisms withing ActivityGraphs. The only way to model non-linear control- and data flow is the DecisionNode, that is, a goto, with all its negative repercussions.

There are none of the structuring mechanisms in the UML 1.5 that have replaced gotos in programming languages, like different kinds of loops, properly nested if/then/else or case/otherwise expressions, exceptions, transactions or similar. Consequently, it is quite understandable that some people consider the ActivityGraphs of UML 1.5 as "*spaghetti-diagrams*".

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

The OMG has addressed this problem by adding so called StructuredActivityNodes ("*structured nodes*", for simplicity) in the new version of the UML. However, these constructs are totally new, not just in the UML, so there is little to no experience yet, concerning both their concrete and abstract syntax. Also, their semantics is described only in a very superficial way, without the formal rigor necessary for practical model exchange, or enactment, or verification tools. This paper strives to improve in this respect, providing formal definitions for the various types of Structured Activity Nodes, and exploring the new notions both syntactically and pragmatically.

1.2 Approach

Since the standard stipulates that Activities "*use a Petri-like semantics*" (cf. [21, p. 292]), it is natural to use Petri nets as the semantic domain. For various reasons, plain PT-nets are not sufficient, however. Building on my previous work, we use Procedure-call Colored Petri-nets.

In [29–32], I have shown how control-flow, procedure calling, data-flow, and exceptions in UML Activity Diagrams can be mapped to higher-order variants of Petri Nets. The question now is, how the remaining constructs—LoopNodes, ConditionalNodes, and ExpansionRegions—can be embedded: are they just syntactic sugaring, or do they require semantic additions? Also, the notion of streaming put forward in the UML obviously raises questions related to concurrency. And, putting all this together: is it possible to stretch the approach presented in [29–32] a bit further to also cover structured nodes?

2 Activity Diagrams

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, and exceptions of Activities is found in [29], [30], and [31, 32]. For brevity, we only give a short summary of the intuition here.

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 (see Figure 3).

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 them: "*Activities are redesigned to use a Petri-like semantic*" (cf. [21, p. 292]). The intuition of the semantic mapping is presented in the first six compartments of Figure 2.

The whole area of Activity Diagrams has been structured into a graph of packages (see Figure 1) of increasing expressive power. This paper deals with the constructs of the package CompleteStructuredActivities.

For elementary 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.

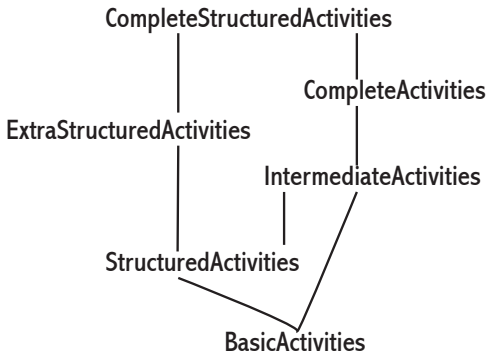

Fig. 1. Levels of expressiveness in Activity Diagrams.

As Actions may call Activities, transitions should be able to call nets like procedures. Obviously, this goes beyond traditional P/T-nets, so that [29] resorts to procedural Petri-nets (first described in [18]). We require that each Activity is represented by a separate boxed and named Activity Diagram similar to UML 2.0 Interaction Diagrams (cf. Figure 6 and [33, 28]), each of which may then be transformed separately into a plain Petri-net. These individual nets are held together by a refinement relationship exploited at run-time, i.e., when a family of Petri-net is executed.

For the intuition of Activities without StructuredActivityNodes see Figure 2. There, each field shows the intuition of the translation for one area of Activities. Due to the restricted space in this paper, the mapping is given only intuitively, with Activity Diagram fragments (left column) to Petri-net fragments (right column). The details are defined in [29, 30]. Exceptions are left out (see [31, 32]).

3 Abstract syntax and semantic domain

In UML, the abstract syntax is defined by the metamodel. Figure 3 shows a small portion of the metamodel concerned with StructuredActivityNodes. Obviously, the metaclass StructuredActivityNodes is a subclass of Action[2] Every StructuredActivityNode may contain ActivityNodes and ActivityEdges. StructuredActivityNode has three subclasses, ConditionalNode, LoopNode, and ExpansionRegion, which are examined in sections 4, 5, and 6 in turn.

The semantic domain consists in a rather arcane Petri-net dialect, so it is worth to explain it, and discuss its choice in the first place. First of all, the standard more or less prescribes the use of Petri-nets as the semantic domain by stating that "*use a Petri-like semantics*" (cf. [21, p. 292]) (see e.g. [20, 26] for introductions). However, many

[2] StructuredActivityNode is also a subclass of ActivityGroup, but since it is only "*a generic grouping construct for nodes and edges*" and has "*no inherent semantics*" (cf. [21, p. 301]), they may be safely ignored in this paper.

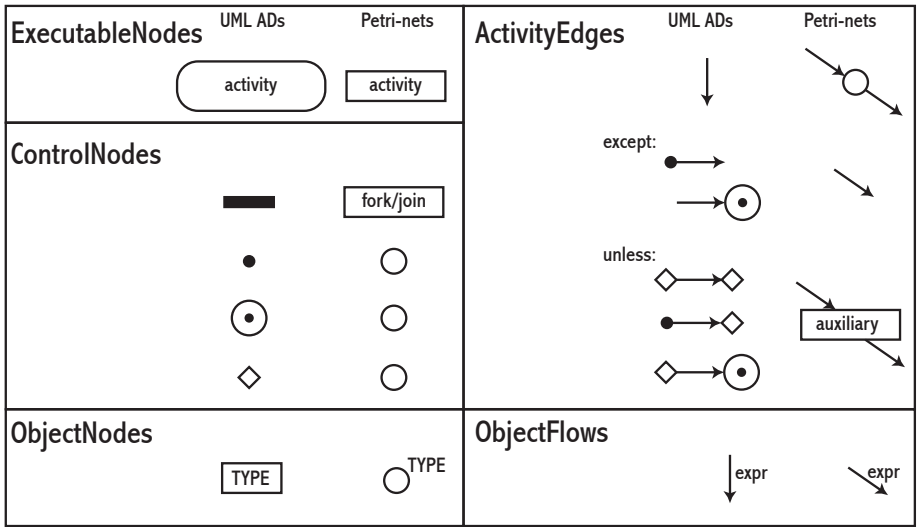

Fig. 2. The intuition of the semantic mapping for Activities. Actions that call Activities are represented as Petri-net transitions with a double outline. They are translated into refined transitions of a procedural Petri-net.

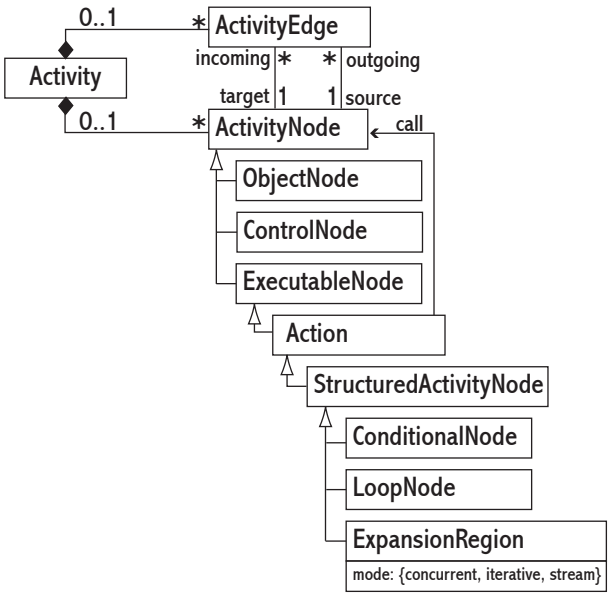

Fig. 3. The UML 2.0 metamodel as far as ExpansionNodes are concerned (simplified).

of the constructs introduced in the standard are not present in elementary P/T-nets. Fortunately, most of these questions have already been addressed in the world of Petri-nets, e.g., high-level inscriptions occur in colored petri nets (cf. [17]), procedure calling has been treated in procedural nets (cf. [18]). All that is missing is a slight extension of procedural nets to cover exception handling and a fusion of the isolated features. This has been done in [31, 32], and due to lack of space, it is not possible to repeat the complete formal definition here.

A procedural net is a set of simple nets with initial and terminal markings, and a refinement function from transitions into the set of nets. Whenever a refined transition t fires, a t_{call} event is recorded and a new instance of the refinement net are created. When this instance reaches its terminal marking, a t_{return}-event is recorded, and the instance is removed.

4 ConditionalNodes

The standard gives no hint on the concrete syntax of a ConditionalNode. I propose to represent it similar to an ExpansionRegion as a dashed line with ObjectNodes for the input- and output-parameters and one compartment for each pair of condition and consequence, separated by dashed lines (see Figure 4, left).

A conditional nodes is a kind of set of guarded commands: "*a conditional node is a structured activity node that represents an exclusive choice among some number of alternatives.*" (cf. [21, p. 313]). Each consists of a set of "*clauses, [each consisting] of a test section and a body section*". When executing a ConditionalNode, all tests are executed. Then, the body section of one of those that yielded true is chosen nondeterministically and executed. Alternatively, "*sequencing constraints may be specified among clauses*".

In order to reduces complexity, I propose that the test sections be side-effect free. Thus, it is probably the easiest to use the guards introduced with DecisionNodes. Also, I suggest that the body section consist of single Actions. These may call upon other Activities, of course. Then, ConditionalNodes are just syntactic sugar, and their meaning is best defined by an expansion into more basic constructs. Figure 4 (right) shows how this may be done.

5 LoopNodes

A LoopNode has "*setup, test, and body sections*" (cf. [21, p. 341]). The standard gives no hint on the concrete syntax of a LoopNode. I propose to represent it similar to an ExpansionRegion (and my proposal for ConditionalNodes) as a dashed line with ObjectNodes for the input- and output-parameters and compartments for setup, test, and body regions separated by dashed lines (see Figure 5 (a)).

There are several possible interpretations of LoopNodes. The first interpretation considers the setup and body compartments as individual Actions (that might be refined by Activities). The test compartment becomes a guard on a DecisionNode, similar to the treatment of guards in ConditionalNodes (cf. Figure 5 (b)). A second interpretation might translate each of the compartments into a single action (cf. Figure 5 (c).

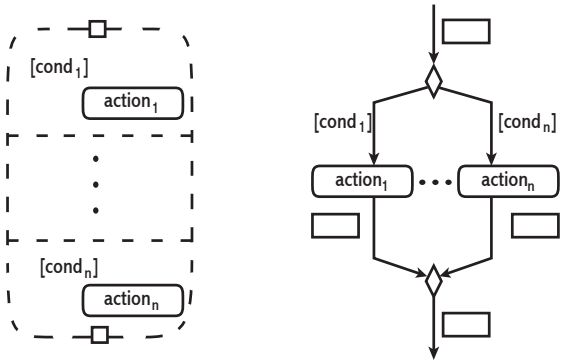

Fig. 4. ConditionalNodes are syntactic sugar: sugared form (left) meaning (right).

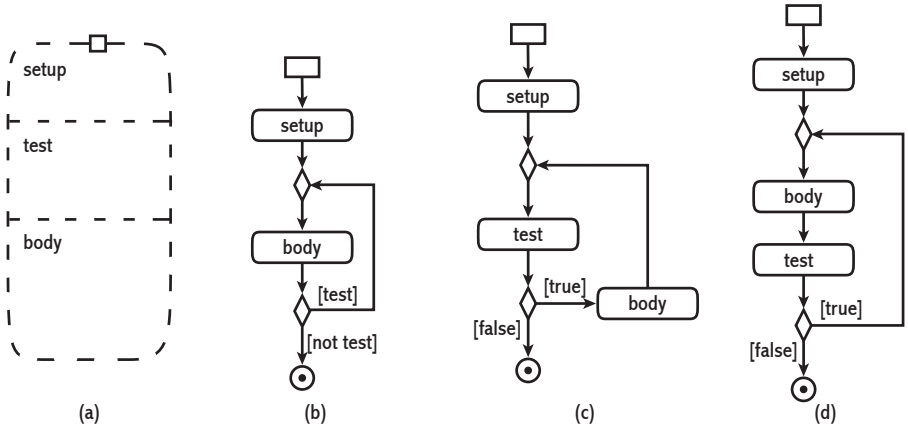

Fig. 5. LoopNodes as syntactic sugar: sugared form (a) and possible expansions (b–d).

The standard provides a facility to implement both while-do and repeat-until loops, that is, loops where the condition to continue execution is tested before or after executing the loop body (cf. (c) and (d) of Figure 5). If the former case is intended, the isTestedFirst-attribute of LoopNode is set to true, in the latter case, it is set to false. Again, there is no syntactic feature in the standard to distinguish these two variants. Thus, I propose to use the sequence of the body and test sections in a LoopNode, that is, if the test section stands above the body section as in Figure 5 (a), the test is executed first, and we have a while-loop. For until-loops, the sections are simply interchanged. This would avoid an additional inscription, and make the type of loop very obvious even for people that are not familiar with the underlying concepts.

There are more questions concerning the interpretation of LoopNodes, however. The standard declares that a LoopNode "*is a costructured activity node that represents*

a loop with setup, test, and body sections." (cf. [21, p. 341]). Unfortunately, the meaning of the word "*costructured*" remains opaque. The standard goes on by saying that "*each section is a well-nested subregion of the activity whose nodes follow any predecessors of the loop and precede any successors of the loop.*" (cf. [21, p. 341]). This seems to indicate that a LoopNode is just a kind of additional structure on top of an Activity. That is, the Activity diagram fragment shown in Figure 6 (left) is identical to the Activity diagram fragment of Figure 6 (right), and the LoopNode structure shown there has no meaning at all, and is just a kind of reading guide that overlays the structure of the Activity. But then, why do we have LoopNodes in the first place?

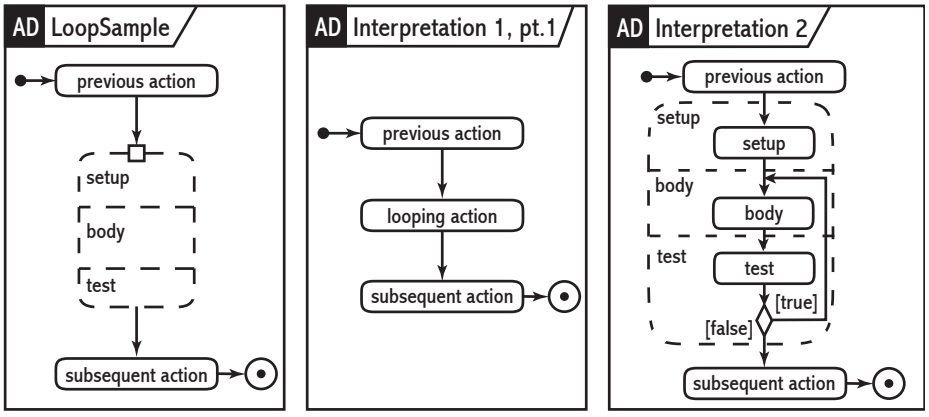

Fig. 6. LoopNodes in context: a fragment of an Activity containing a LoopNode (left), interpreting a LoopNode as an Action calling an Activity as described in Figure 5 (middle), and an alternative interpretation where the LoopNode is expanded in its context (right).

Also, consider the interaction of loops with exceptions. A natural construction one would expect to be admissible (and simple enough semantically) is the situation shown in Figure 7 (left). What is the scope of the exception—is the body section also an InterruptibleActivityRegion? Or should it be possible that the exception is raised in the test and/or setup, too?

Actually, both situations make sense. So I propose to follow the interpretation shown in Figure 6 (middle) where the LoopNode is a plain Action refined by the net of, say, Figure 5 (d). This node is now also the protected node. So, it is possible to raise an exception anywhere in the loop. If the scope is supposed to be more restricted, the Activity that refines the "*looping action*" node raises the exception.

Another interesting consideration is which result executing a LoopNode yields. The standard declares that "*the test and body sections are executed repeatedly until the test produces a false value. The results of the final execution of the test or body are available after completion of execution of the loop.*" (cf. [21, p. 341]). This does not make much sense, however, since the last thing executed before leaving the loop must always be the test, and the test must finish with value false, so the result of a LoopNode is the constant

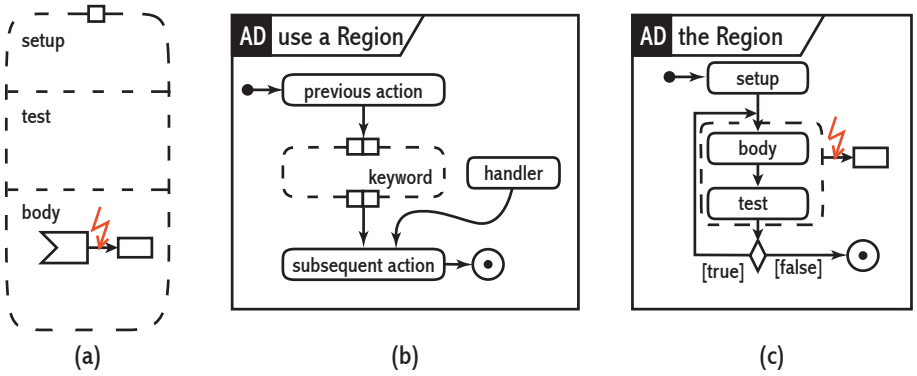

Fig. 7. Interactions between structured nodes and exceptions (a); the context of an ExpansionRegion (b) may call upon an Activity defined, say, like (c). In this example, the raising scope of the exception (the InterruptibleActivityRegion) has been set to encompass both the body and the test section.

false. It is not clear, how a result of the last execution of the body or an accumulated result may be passed on outside the LoopNode. This issue definitely needs clarification in the standard.

6 ExpansionRegions

The description of ExpansionRegions in the UML standard exhibits far more errors and inconsistencies, and is much less detailed than the descriptions of LoopNodes and ConditionalNodes. Also, there is a fairly straightforward intuition in terms of their practical value for the latter two, even if this is not documented in the standard. For ExpansionRegions, however, such a pragmatic intuition is not obvious. So, to some degree, this section is speculative in exploring possible intentions of the standard.

ExpansionRegions are ActivityNodes that process collections of elements as a unit. The standard declares that when "*an execution of an activity makes a token available to the input of an ExpansionRegion, [it] consumes the token and begins execution. The ExpansionRegion is executed once for each element in the collection*" (cf. [21, p. 326]). Thus, an ExpansionRegion can be viewed as a kind of map-function from one collection to another. Such a functionality may be implemented in four different ways (not counting mixtures of these).

iterative that is, in a loop where each of the elements is treated in turn, but processing starts only on the complete set of arguments;

streaming is similar to iterative in that only one element is processed at once, but processing of the first element may start even though further elements have not yet arrived;

parallel meaning that all elements are processed in lockstep, starting, proceeding, and ending together, irrespective of the actual time needed to process individual elements, and so possibly creating idle times;

concurrent is similar to parallel in that all arguments are treated potentially at the same time, but independent of each other rather than in parallel.

Following the UML standard, it is possible to specify three of these variants by setting the mode-attribute of ExpansionRegion. It may carry the values iterative, stream, and concurrent. Beware of the last mode setting, however: due to a spate of serious printing mistakes in the standard, the word parallel is used in all but one place instead of the word concurrent. The behavior explained is definitely concurrency, not parallelism, though: "*the execution[s of the elements in the collection] may happen in parallel, or overlapping in time, but they are not required to*" (cf. [21, p. 326]). To achieve parallelism, the standard would have to specify the execution as "*parallel in lockstep, beginning, proceeding, and ending simultaneously*" or similar.

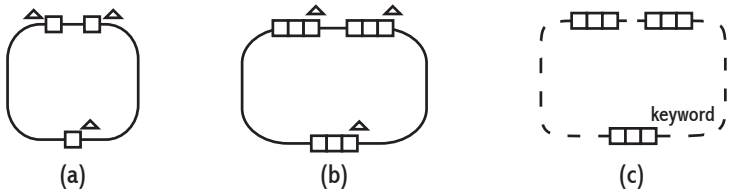

Fig. 8. Syntax of (a) plain ExecutableNode with Pins of stream-value, (b) ExecutableNode with collection Pins of stream-value, and (c) ExpansionRegion (necessarily with collection-valued Pins).

At this point, we hit on an important feature of Activities. The standard states that "*the expansion region is executed once for each element in the collection (or once per element position, if there are* **multiple** *collections).*" (cf. [21, p. 326, emphasis added]) This seems to imply that an Activity is a kind of dataflow-computer with different parameters flowing through it—similar to a Petri-net, in fact[3], and not like an individual run. Observe also, that if there are several computations going on at the same time, these must be isolated from each other to avoid interactions. Thus, a macro-like expansion strategy, as has often been proposed for high-level Petri-nets, is out of the question here. In [29], this problem has already been solved silently by the very definition of procedural Petri-nets. So, using the procedure call semantics for LoopNodes and ExpansionRegions, this problem is avoided altogether.

Thus, similar to the treatment of LoopNodes as proposed in Figure 6 (middle), ExpansionRegions should be translated as refined Actions (see Figure 7 b and c), where various calls to the same refinement transitions executes in its own state space. Depending on the mode of the ExpansionRegion, different refinement nets must be used (see Figure 9).

[3] Also, this raises again the question how well Activity Diagrams are suited for, say, workflow modeling, and whether Activities represent a workflow type/schema or an instance.

6.1 Iterative ExpansionRegions

In an iterative ExpansionRegion, "*the executions of the region must happen in sequence, with one finishing before another can begin. [. . .] Subsequent iterations start when the previous iteration is completed. During each of these cases, one element of the collection is made available to the execution of the region as a token during each execution of the region.*" (cf. [21, p. 326]). This mode is treated semantically as shown in Figure 9 (left).

There, for simplicity, the collections have been implemented as lists. In the net, the region is represented by a transition. It may be refined to call another net in the sense of procedural Petri-nets (cf. [18, 29]). The region may start processing the first element of the collection right away by taking it from the list and, after processing it, adding it to a result list. When all elements have been processed, the Input collection is depleted (i.e. has become the empty list), and the resulting list is reversed to achieve the original order again (this last step may be omitted of course, if the collection is unordered).

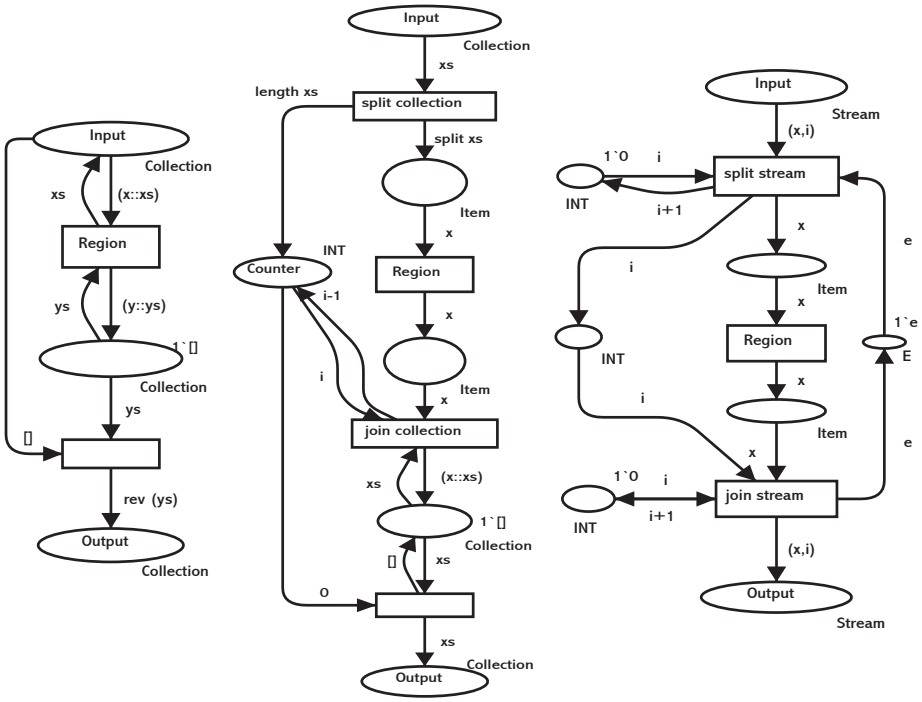

Fig. 9. Semantics of ExpansionRegions: iterative mode (left), streaming mode (middle), and concurrent mode (right).

6.2 Concurrent ExpansionRegions

In a concurrent ExpansionRegion, "*the execution may happen in parallel, or overlapping in time, but they are not required to.*" (cf. [21, p. 326]). This mode is treated semantically as shown in Figure 9 (right). There, the collection is first split up into its elements. The number of elements is tracked in the place Counter. Then, each element may be processed by the Region. Observe, that a transition may fire concurrently to itself as often, as there are tokens activating it. As soon as the first results are produced, they may be collected again by the join collection transition. If all results are processed, the Counter has been decreased to zero, and the output may be produced.

Note that it would be not easy at all to realize a truly lockstep-parallel execution mode of an ExpansionRegion, since we could not use the procedure call mechanism, but would have to resort to a kind of net folding.

At this point, a small digression concerning the tool used is in place. The nets in Figure 9 have been drawn and simulated using CPN Toolset (see [7]). The inscriptions are Standard ML code (the programming language used for inscriptions in the CPN Toolset, cf. [22]), with some special conventions. For instance, `E` is the color of plain tokens ("*black dot tokens*"), and the only value of this type is `e`. Multisets are represented like `1‘true++2‘false`, which means: one token of value `true` and two tokens of value `false`. Tuples are written as (x, y), and lists as `(head::tail)`.

6.3 Streaming ExpansionRegions

In an ExpansionRegion with mode stream, "*there is a single execution of the region, but its input place receives a stream of elements from the collection. The values in the input collection are extracted and placed into the execution of the expansion region as a stream [...]. Such a region must handle streams properly or it is ill defined. When the execution of the entire stream is complete, any output streams are assembled into collections of the same kinds as the inputs.*" (cf. [21, p. 326]). Thus, at any given time during the stream processing, some elements of a stream may have been processed already, some may be being processed in the very instant, and some may still be awaiting processing. This is in contrast to other collection-valued ExpansionRegions, where all elements of a collection must be present *before* the processing starts, and all elements of the collection must be processed *before* the Region is terminated.

Starting with the simple case of an individual stream, this behavior may be captured by the net shown in Figure 9 (middle). Collections may be represented by tagging the elements with sequence numbers, i.e. `Element STREAM` $=$ `Element`$\times$`SequenceNumber`.

First of all, the stream is split into sequence numbers and elements proper. While the element is processed by the Region, the sequence number is passed by. The complement place ensures proper synchronisation.

As a side remark, the standard also proposes stream-valued Pins that are not collection-valued (cf. [21, Fig. 283, p. 358], reproduced in Figure 8 a and b), but it remains unclear, how this may be interpreted.

7 Conclusion

7.1 Summary and contribution

In this paper, the concepts related to StructuredActivityNodes are examined by defining a straightforward semantics based on Petri nets. Some problems concerning the concrete and abstract syntax and the semantics have been uncovered in the standard (concrete syntax of Conditional- and LoopNodes, scope of exceptions, single/multiple inputs), and possible solutions have been proposed.

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] and [29–32]. The current paper covers structured nodes. Together with [29–32], now all of UML 2.0 Activities have been covered with formal semantics (consider again Figure 1).

7.2 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 (see [1, 2, 4–6, 8–15, 19, 23, 24, 27]) has become obsolete. See Figure 10 for a comparison.

In particular, structured nodes which have not been there in the UML 1.5 have not been addressed so far. Also, it seems that so far, only very little has been published on the UML 2.0 Activity Diagrams: [3] examines expansions and streaming in a intuitive way, focusing on shared input pins and some aspects of streaming. [29, 30, 32, 31] provide formal definitions of the semantics of control-flow, procedure call, data-flow, and exceptions in UML 2.0 Activities, respectively. This paper builds on the latter four.

7.3 Open questions

We may thus now turn to questions like what refinement and composition means for Activities, or how the new constructs work in the field. Also, 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. Also, examples soon become too complex for manual treatment, and so we need tool-support.

Furthermore, if an ExpansionRegion is indeed a kind of map, then how would a fold-operation be accomplished? It requires several flows to be joined together, but neither are there specific constructs for this, nor are there any hints how several instances of one Activity might interact. Note that it is not sufficient to simply inject a fork right at the start of an ExpansionRegion. Embedding parameters in control-objects might be a way out, but seems to raise more questions than it answers.

References

1. Thomas Allweyer and Peter Loos. Process Orientation in UML through Integration of Event-Driven Process Chains. In Pierre-Alain Muller and Jean 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.

authors, references	UML version	semantic domain	control flow	data flow	hierarchy	exceptions	streaming	structured nodes	rigor
Allweyer et al. [1]	0.9	–	wf	√	–	–	–	–	low
Apvrille et al. [2]	1.x	LOTOS	wf	–	–	–	–	–	medium
Börger et al. [6]	1.x	ASM	wf	–	√	–	–	–	medium
Bolton & Davies [5, 4]	1.3	CSP	wf	–	–	–	–	–	low
Eshuis & Wieringa [9, 10]	1.x	algorithm	wf, nwf	–	–	–	–	–	high
Eshuis & Wieringa [12, 11]	1.x	LTS	wf, nwf	–	–	–	–	–	high
Gehrke et al. [15]	1.0	PN	wf, nwf	(–)	–	–	–	–	medium
Pinheiro da Silva [24]	1.x	LOTOS	wf, time	–	–	–	–	–	low
Rodrigues [27]	1.x	FSP	wf	–	–	–	–	–	low
Li et al. [19]	1.x	LTS	wf	(–)	–	–	–	–	high
Störrle [29]	2.0	PPN	wf, nwf	–	√	–	–	–	high
Störrle [30]	2.0	CPN	wf, nwf	√	–	–	–	–	high
Störrle [31, 32]	2.0	ECPN	(wf, nwf)	(√)	(√)	√	–	–	medium

PN = simple P/T-Petri-nets ECPN = exception colored Petri-nets FSP = finite state processes
PPN = procedural Petri-nets LOTOS = Language of Temporal Ordering Specifications LTS = labeled transition systems
CPN = colored Petri-nets CSP = communicating sequential processes ASM = abstract state machines

Fig. 10. Comparative categorization of the previous work leading to this article (in column "*control-flow*", wf means well-formed, and nwf means non well formed. The degree of rigour is approximate, where a completely formal definition is high rigour, examples and some formalism is medium, and mere text is low.

2. L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, and J.-P. Courtiat. A New UML Profile for Real-Time System Formal Dessigne and Validation. In Gogolla and Kobryn [16], pages 287–301.
3. João P. Barros and Luís Gomes. Actions as Activities as Petri nets. In Jan Jürjens, Bernhard Rumpe, Robert France, and Eduardo B. Fernandey, editors, *Proc. Ws. Critical Systems Development with UML*, pages 129–135, 2003.
4. Christie Bolton and Jim Davies. Activity graphs and processes. In W. Griesskamp, T. Santen, and W. Stoddart, editors, *Proc. Intl. Conf. Integrated Formal Methods (IFM)*, LNCS. Springer Verlag, 2000.
5. Christie Bolton and Jim Davies. On giving a behavioural semantics to activity graphs. In Reggio et al. [25], pages 17–22.
6. Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. An ASM Semantics for UML Activity Diagrams. In Teodor Rus, editor, *Proc. 8^{th} Intl. Conf. Algebraic Methodology and Software Technology (AMAST)*, number 1816 in LNCS, pages 293–308. Springer Verlag, May 2000.
7. Design/CPN. Technical report, Univ. of Aarhus, 2004.
8. Marlon Dumas and Arthur H.M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In Gogolla and Kobryn [16], pages 76–90.
9. Henrik 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".
10. Rik Eshuis and Roel Wieringa. A formal semantics for UML Activity Diagrams - Formalising workflow models. Technical Report CTIT-01-04, U. Twente, Dept. of Computer Science, 2001.
11. Rik Eshuis and Roel Wieringa. A Real-Time Execution Semantics for UML Activity Diagrams. In Heinrich Hussmann, editor, *Proc. 4^{th} Intl. Conf. Fundamental approaches to software engineering (FASE)*, number 2029 in LNCS, pages 76–90. Springer Verlag, 2001. Also available as `wwwhome.cs.utwente.nl/~tcm/fase.pdf`.
12. Rik Eshuis and Roel Wieringa. An Execution Algorithm for UML Activity Graphs. In Gogolla and Kobryn [16], pages 47–61.
13. Rik Eshuis and Roel Wieringa. Verification support for workflow design with UML activity graphs. In *Proc. 24^{th} Intl. Conf. on Software Engineering (ICSE)*, pages 166–176. IEEE, 2002.
14. Rik Eshuis and Roel Wieringa. Comparing Petri Net and Activity Diagram Variants for Workflow Modelling - A Quest for Reactive Petri Nets. In Weber et al. [34], pages 321–351.
15. Thomas Gehrke, Ursula Goltz, and Heike 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. Martin Gogolla and Chris Kobryn, editors. *Proc. 4^{th} Intl. Conf. on the Unified Modeling Language (≪UML≫ 2001)*, number 2185 in LNCS. Springer Verlag, 2001.
17. Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. I.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.
18. Astrid Kiehn. *A Structuring Mechanism for Petri Nets.* Dissertation, TU München, 1989. appeared as Technical Report TUM-I8902 of the TU München in March, 1989.
19. Xuandong Li, Meng Cui, Yu Pei, Zhao Jianhua, and Zheng Guoliang. Timing Analysis of UML Activity Diagrams. In Gogolla and Kobryn [16], pages 62–75.
20. Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proc. IEEE*, 77:541–580, April 1989.
21. 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 11^{th}, 2003.

22. Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
23. Dorina C. Petriu and Yimei Sun. Consistent Behaviour Representation in Activity and Sequence Diagrams. In Bran Selic, Stuart Kent, and Andy Evans, editors, *Proc. 3^{rd} Intl. Conf. ≪UML≫ 2000—Advancing the Standard*, number 1939 in LNCS, pages 369–382. Springer Verlag, October 2000.
24. Paulo 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.
25. Gianna Reggio, Alexander Knapp, Bernhard Rumpe, Bran Selic, and Roel Wieringa, editors. *Proc. Intl. Ws. Dynamic Behavior in UML Models: Semantic Questions. Technical Report No. 0006 of the Ludwig-Maximilians-Universität, München, Inst. f. Informatik*, 2000.
26. Wolfgang Reisig. *Petri-Nets: an Introduction*. Springer Verlag, 1985.
27. Roberto W.S. Rodrigues. Formalising UML Activity Diagrams using Finite State Processes. In Reggio et al. [25], pages 92–98.
28. Harald Störrle. Semantics of Interactions in UML 2.0. In John Hosking and Philip Cox, editors, *Human Centric Computing Languages and Environments*, pages 129–136. IEEE Computer Society, 2003.
29. Harald Störrle. Semantics of Control-Flow in UML 2.0 Activities. In N.N., editor, *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Springer Verlag, 2004.
30. Harald Störrle. Semantics of Data-Flow in UML 2.0 Activities. 2004. submitted to VLFM'04, June, 20^{th}, available at `www.pst.informatik.uni-muenchen.de/~stoerrle`.
31. Harald Störrle. Semantics of Exceptions in UML 2.0 Activities. 2004. submitted to Journal of Software and Systems Modeling, May, 9^{th}, available at `www.pst.informatik.uni-muenchen.de/~stoerrle`.
32. Harald Störrle. Semantics of Exceptions in UML 2.0 Activities. Technical Report 0403, Ludwig-Maximilians-Universität München, Institut für Informatik, 2004.
33. Harald Störrle. Trace Semantics of Interactions in UML 2.0. 2004. submitted to J. Visual Languages and Computing, February, 13^{th}, available at `www.pst.informatik.uni-muenchen.de/~stoerrle`.
34. Michael Weber, Hartmut Ehrig, and Wolfgang Reisig. Petri Net Technology for Communication-Based Systems. In Michael Weber, Hartmut Ehrig, and Wolfgang Reisig, editors, *Petri Net Technology for Communication-Based Systems*. DFG Research Group "Petri Net Technology", 2003.