On the Pitfalls of UML 2 Activity Modeling

Tim Schattkowsky
C-LAB
tim@c-lab.de

Alexander Förster
University of Paderborn
alfo@uni-paderborn.de

Abstract

With the introduction of new Petri Net-like semantics for Activities in UML 2.0, these have become a complete language for modeling behavior. Thus, UML Activities are nowadays investigated for application in many areas from embedded systems to business process modeling. However, some issues have been discovered that currently seem to limit the practical applicability of Activities. In this paper, we present an overview of the identified semantic and syntactic problems, and point at possible solutions and directions for future research.

1. Introduction

The UML 2.0 [10] essentially provides two different types of behavior models. These include the UML State Machines and the UML Activities. While the first are a derivate of Harel's State Charts [6], the latter are a flow-oriented behavior model.

UML Activities model behavior in terms of computational steps connected by data and control flows. This is actually in contrast to former UML versions, where an Activity was a specialized kind of UML State Machine.

Although the concrete syntax of the basic model elements has not been changed, the consequences of this change in semantics are essential. These consequences include the fact that since UML State Machines are synchronous, UML 1.x Activities could only perform distributed computational steps in synchrony while the firing of the steps in a UML 2.0 Activity is not synchronized. Furthermore, in a UML 1.x Activity, only one step per concurrent region can be activated at a time. The new semantics allow for models where essentially all steps could be enabled.

These new semantics have opened up a number of new application areas for UML Activities, which range from modeling embedded hardware and software [16] to business process modeling [3].

For some applications, like modeling embedded systems, state-oriented modeling seems to be dominant, even in current UML-based approaches like $^{X}_{T}$UML [11] and xUML [13]. However, as we pointed out in previous work, UML Activities are often a more convenient alternative, both for hardware [15] and for software systems [14] modeling. The reason is that state-oriented modeling often cannot be efficiently applied, e.g., for modeling complex sequential algorithms or fine grained concurrency. Activities do provide concepts that help capturing such behavior much more naturally.

Furthermore, for data flow dominated behavior like in multimedia applications, state-oriented modeling is usually ineffective. Thus, flow-oriented modeling languages like Synchronous Data Flow Graphs (SDFGs) [18] are often applied for such applications. Eventually, most of these flow-oriented languages are based on or closely related to Petri Nets [12].

In business process modeling, flow-based behavior modeling is already widely applied, e.g., in the form of Event-Driven Process Chains [7] or similar notations. Again, these notations are more or less closely related to Petri Nets.

Nowadays, the UML has become the de-facto standard for modeling software systems. Thus, UML Activities are increasingly applied for modeling flow-oriented behavior, both as an alternative to state-oriented modeling and as a replacement for flow-oriented modeling, e.g., using Petri Nets.

Although the new UML 2.0 semantics for Activities seem to be close to high-level variants of Petri Nets, there are essential differences. These differences lie mainly in the fact that unlike in a Petri Net the activation of computational steps in a UML 2.0 Activity is not completely local, and that some of the model elements have quite complex semantics.

The complexity of the new Activity semantics seems to be induced both by the increased abstraction compared to other flow-oriented languages and the urge to maintain a certain syntactic and semantic compatibility with UML 1.x Activities.

International Workshop on Modeling in Software Engineering (MISE'07)
0-7695-2953-4/07 $20.00 © 2007 IEEE

In this paper we present an overview of the semantic and syntactic problems with UML 2.0 Activities that have been identified and examine the underlying causes. Furthermore, we describe how these issues can be circumvented or at least alleviated.

The remainder of this paper is structured as follows. The next section discusses related work before section three discusses the actual problems with Activity modeling. Section four outlines possible solutions to the presented problems before section five closes with a conclusion and an outlook on future work.

2. Related Work

Unfortunately, many of the existing works on UML Activities like [5] deal with UML 1.x Activities which were a variant UML State Machines. Thus, these works do not cover many of the issues raised here. In particular, considerations about UML 1.x Activity semantics have only limited applicability. Furthermore, works on the formal verification of UML 1.x Activities like [4] do not cover the interesting new features of UML 2.0 Activities.

However, there exist works highlighting the difficulties and problems when dealing with the complexity imposed by the new UML 2.0 semantics for Activities like [17] where semantic problems with exceptions, streaming and non-locality are discussed and compared to Petri Net semantics.

For the practical application of UML 2 Activities, most approaches significantly limit the set of supported Activity model elements and introduce further specializations for some of the employed elements. This is often done by defining a respective UML profile. One such example is the UML-based SysML [9] language. Other examples include the Business Process Modeling Notation BPMN [8] and our works on the specification and synthesis of embedded hardware and software [14][15][16].

3. Practical Problems with Activities

There are different kinds of problems when applying Activities. These include general, sometimes related problems like

- lack of formal semantics,
- excessive supply of concepts,
- semantic inconsistencies, e.g. in the activation of Actions
- variance in complexity of some concepts,
- semantics that are not convenient for execution
- semantic variation points, and
- lack of methods for formal verification

which seem to be at least partially caused by the desired wide applicability of the language. Furthermore, there are practical usability problems like

- scalability of the concrete syntax
- semantic inconsistencies in the syntax, e.g., for implicitly forked and joined control flows compared to object flows and the representation of token competition
- vast amount of alternative presentation options for model elements,
- inefficiency the concrete syntax, e.g., in capturing common simple control structures like loops and conditional blocks, and
- generally poor efficiency in drawing Activity Diagrams.

While some of these problems are already caused by the abstract syntax and semantics of an Activity, others seem to be solely related to the concrete syntax for Activities as defined in the UML specification.

3.1. Semantic Issues

First, the UML specification does not provide formal semantics for UML Activities. Informal semantics are provided as plain text. However, it has turned out, that these are not very precise and sometimes inconsistent [17].

The execution semantics of Activities is generally based on the concept of token flow. The idea was to define a behavior that resembles Petri nets in many respects. This has advantages for example for modeling business processes, were the old state-based semantics as in UML 1.x were not appropriate.

The basic structural simplicity of Petri nets and the fact that token flow can always be determined locally makes it feasible to generate a vast domain of theoretical results regarding their behavior. Unfortunately, these cannot be directly transferred to Activities.

The execution semantics of UML 2.0 Activities are much more complex than in Petri Nets. An important paradigm in the semantics definition of Activities is

that tokens may not remain at ControlNodes [1]. ControlNodes can generally only accept incoming tokens if the ActivityNodes subsequent to the ControlNode accept the token traversing the ControlNode. In particular if more than one ControlNode are directly connected to each other, the determination how tokens can flow involves different ControlNodes and Actions and becomes therefore non-local. There are also special ControlNodes which show an explicit non-local behavior as for example the ActivityFinalNode that destroys all tokens within the Activity.

The non-local behavior of Activities has many disadvantages. It contradicts conventional concepts as for example found in Petri nets [17]. It thwarts the generation of theoretical results about an Activity's behavior and it makes it very difficult to build execution machines for Activities that follow the exact semantics as defined in the UML 2.0 Specification.

Finally, the paradigm of tokens not remaining at ControlNodes generates not only non-locality but also severe semantic problems and finally had to be lifted for some ControlNodes as for example for the ForkNode. Figure 1 shows why that had to be done. When no token may remain at a ControlNode, and say Action "A" offers a token, the question whether "C" may execute is depended on whether "B" offers a token and also whether "D" accepts tokens. To lift the problem, the OMG introduced a special rule for ForkNodes saying that they accept incoming tokens when at least one outgoing edge can traverse a token. The tokens offered on the other edges leaving the ForkNode are buffered, contradicting the original paradigm.

Another semantically interesting but practically problematic feature is ParameterSets. While these seem to be convenient for describing alternative signatures of a behavior or just for defining optional inputs, the generality of the approach causes some problems. Since different ParameterSets may overlap, it is not clear what happens if one such sets is a superset of the other.

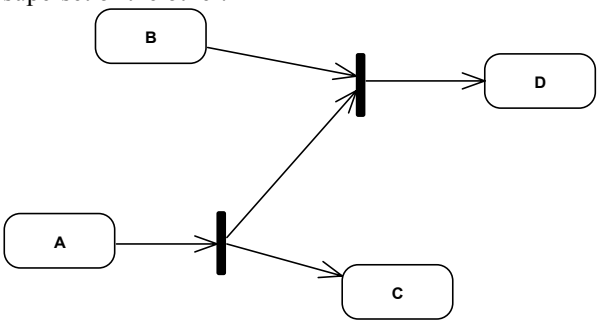

Figure 1: Non-local behavior

Consistency of the actual implementation of an Activity using ParameterSets or streaming Pins is another problem. This implementation has to be consistent with the behavior implied by these in- and outputs. Thus, only the declared outputs must be produced for a given ParameterSet.

Finally, just like streaming Pins, the use of ParameterSets limits the application of SDFG theory to Activities, as it requires the inputs and outputs of each computational step to be known a-priori. This hinders the application of the respective formal methods for verification.

A number of features of UML 2 Activities are contradictory to themselves or other features. Such contradictions in practice lead to misunderstanding of the actual semantics of a model and can finally yield communication problems or even flawed designs.

One such contradictory issue is the activation of Actions within an Activity. The general rule is that Actions gets activated, i.e. can start executing, if there are tokens on all incoming edges/input pins. But what happens if an Action has no incoming ActivityEdges? Then the Action gets activated directly with the activation of the whole Activity. This is somewhat contradictory to what an Activity developer would conceive as the normal behavior that Actions get only activated if they are triggered by incoming tokens (i.e. from preceding Actions).

To make the situation even more complicated, there are some special rules for one subtype of Action, the AcceptEventAction. The AcceptEventAction comes in two different flavors, one to receive events and one that represents the semantics of an autonomous timer. The special property of an AcceptEventAction is that, once activated, it stays activated until the surrounding container terminates; e. g. the surrounding Activity. Here again the activation rule for this kind of Actions is contradictory to that of regular Actions. Taking it all together, determining when and whether an Action is activated for how long can sometimes be quite hard.

Finally, the idea of streaming pins, although a useful feature, has several implications including the fact that it breaks the SDFG-like idea that all inputs on an Action should be consumed at once and all outputs appear at the same time.

Another contradictory feature is the token flow semantics at ForkNodes and JoinNodes when it comes to object flow. Control tokens can easily be duplicated at ForkNodes and synchronized at JoinNodes, but what happens to an object token that traverses a ForkNode? One can imagine that the token can also be duplicated, but what happens to the data object itself? There are two possible interpretations: either the object itself is also cloned, resulting in two different copies of the

object or one gets two tokens referring to the same object. Cloning objects is in general non-trivial because it has some implications about how this can be performed, if it is legal at all and whether you produce shallow or deep copies of an object and the objects associated with it. Producing additional references to an object can sometimes also lead to undesired behavior, e. g. when it comes to object manipulation or deletion. However, from a pragmatic point of view, token replication makes sense, but needs reliable semantics.

If one designs an Activity one might want to join different object flows using a JoinNode. Then the object tokes that reach a JoinNode have to be merged in some manner. In the design of the Activity, it often remains unclear what happens to the different pieces of information traversing a JoinNode, in particular how exactly the information in the object nodes are merged to produce a single output token. The suggested solution for this problem in the UML specification is to attach a join specification to the JoinNode. As the JoinSpecification actually may perform a Behavior to combine the objects, it may be very misleading to have an Activity model in which some important parts of the behavior are not modeled using Actions or SubActivities but are modeled just in side-notes to some ControlNodes.

Also the use of MergeNodes with object flows can easily lead to a contradictory situation when objects of different types or classes are reaching the MergeNode over the incoming edges. Since the object tokens basically traverse the MergeNode unchanged, objects of different types can leave the MergeNode in an uncontrolled manner.

In UML Activities, guards can be attached to any ActivityEdge to control the traversal of the edge by tokens. For object flows, there exists another way of determining/restricting the object tokens traversing the ActivityEdge which is to specify a selection behavior on an object flow. Using a note symbol with the keyword "<<selection>>", an additional rule for token selection is added to the object flow that possibly contradicts the ActivityEdge's guard expression.

It is also possible to attach "<<transformation>>" notes to an object flow modeling some sort of processing behavior for the objects while the objects traverse the ActivityEdge. This is another example of how Behaviors are modeled apart from the normal Behavior model elements like Actions, CallBehaviorActions etc. This way important behavioral steps are possible made oblique by putting them into simple note boxes.

3.2. Abstract Syntax Issues

One practically important example for an issue that very frequently leads to confusion is the use of ActivityNodes with multiple outgoing or incoming edges and guards. The most trivial, but still quite devastating problem is the fact that many users are not aware of the implicit fork and join of control flows. While this is in practice a very significant problem, there are other related issues, as we will discuss here.

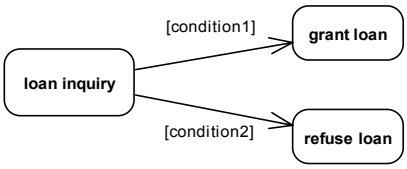

Figure 2: Implicit fork with guards

One frequently sees Activity models as in Figure 2. Intuitively, one might think that the semantics of this Activity is that after the loan inquiry there is an implicit decision whether the loan will be granted or refused. In fact, an Action with multiple outgoing edges represents in implicit fork. To model the exact behavior of a decision, an explicit DecisionNode has to be inserted between the "loan inquiry" and the "grant"/"refused" Actions. Important differences between an (implicit) ForkNode with guards and a DecisionNode with guards lie in the behavior if none or more than one guard can evaluate to true at the same time. In particular, if both conditions are true, tokens will flow over both edges in contrast to the semantics of a DecisionNode.

To make the situation even more complicated and contradictory, say the modeler just wanted to change the Activity of Figure 2 to include object flow instead of simple control flow. The modeler decides to use the pin notation for the object flow, as in Figure 3. Unfortunately, this putatively minor change alters the semantics of the token flow. Now, if both guard conditions evaluate to true, there will be a race condition for the object token such that only one subsequent Action will receive a token; which one that is, is undetermined.

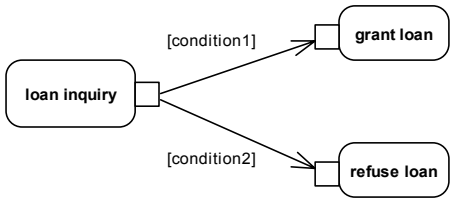

Figure 3: Token competition at Pins

3.3. Concrete Syntax Issues

UML Activities are usually represented through Activity Diagrams. The diagram elements represent elements of the abstract syntax, but the mapping is not always straightforward. Furthermore, sometimes the visual representation is actually unintuitive and ambiguous or just misleading, because of similarities in appearance with other notations for flow-oriented modeling.

As we will discuss in this section, both the creation of Activity Diagrams and their readability have proven to be practical problems under certain circumstances. However, it is important to note that these issues are at first hand a problem of the concrete syntax, but not necessarily also a problem with the abstract syntax or the semantics.

Activity Diagrams are plagued by general issues of visual notations like layout considerations. However, they sometimes even emphasize these issues, e.g., by mapping simple elements to composite visual elements. As a consequence, these require additional effort while drawing the diagram. One such example is the use of Pins, which actually forces the designer to layout several boxes and edges for each computational step. In such cases, the effort for representing a part of the respective abstract syntax is often significantly higher than desirable.

Generally, the efforts for certain common applications like representing sequential algorithms are significantly higher compared to using textual languages. This is in particular obvious when the use of Activities provides no increase in abstraction, but forces the designer to map implicit semantics into explicit model elelemnts, e.g., sequential order into a graph of boxes with control nodes.

For some model elements like ParameterSets, the visual representation reduces the expressiveness compared to the abstract syntax. In this example, it is in practice barely able to reflect all possible combinations of Pins as defined by the abstract syntax. However, the proposed visual notation for ParameterSets is hard to read anyway.

Some of the Activity model elements do not have a visual representation at all, which can prevent their application. A striking example is the LoopNode for representing loops in a block-style manner. At the abstract syntax level, this model element seems to be particularly convenient for enhancing the presentation of sequential code and could have helped with the aforementioned shortcomings. However, since the designated concrete syntax fell into the previous category by having certain limitations, no concrete syntax has been proposed at all and the model element is largely useless for practical application.

For some model elements, several presentation options have been defined. This seems to be at first hand a good thing since it widens the application areas by aligning the UML concrete syntax with common practices in different application domains. However, on the other hand it requires the reader to be aware of all of these alternatives. Though this is conceptually a minor issue, it turns out to be a relevant problem in practice, in particular when communicating the content of an Activity Diagram outside a domain.

Finally, sometimes the concrete syntax does not match the abstract syntax. Again, this can be disturbing when communicating the meaning of an Activity Diagram. One such example is when a DecisionNode and a MergeNode share the same symbol in the diagram. However, in this case the question may also be why this is not allowed by the abstract syntax at all.

4. Possible Solutions

Many of the semantic problems presented here can be addressed by the proper selection of a significant subset for the application domain, as demonstrated by existing approaches. Furthermore, some of the problems are rather academic.

For formal verification, a restriction of the concepts to subsets that map into other semantics domains like Petri Nets, SDFGs or process algebras like CSP can be employed to exploit existing works in these domains. However, this is not always desirable since it usually implies a loss of potentially interesting properties provided by Activities. Thus, fundamental work on the direct formal verification of UML 2.0 Activities themselves could be a significant contribution.

Certain changes to the abstract syntax could be useful. The separation of MergeNode and JoinNode in the abstract syntax seems to be artificial. Furthermore, support for implicit fork on data flows could be useful. However, essentially the same can be achieved just by changing the concrete syntax, as demonstrated by the UML itself in the case of the MergeNode and the JoinNode.

The productivity and scalability problems of Activity diagrams can be addressed in different ways. Providing an alternative domain specific concrete syntax reflecting the relevant subset of model elements seems to be useful. In particular the use of textual languages for sequential programs seems to be promising [2]. However, this is a significant limitation compared to the expressiveness of Activities. Thus, a more general approach providing complete coverage of

the abstract syntax is needed and could include forgotten important elements like the LoopNode.

Finally, tool support could be improved to increase the productivity in Drawing Activity diagrams. By trivial metrics, like the number of mouse clicks to draw a diagram, these are still very inefficient compared to the actual abstract syntax represented. However, tools are starting to include features that increase productivity here, from automatically attached Pins to intelligent routing of flows.

5. Conclusions and Future Work

In this paper, we have highlighted several problems that may occur when applying UML 2.0 Activities in practice. These problems lie both in syntax and semantics of the language.

There are several small conceptual misunderstandings and problems with the UML 2.0 Activity semantics. Here, the lack of a formal semantics seems to be a significant drawback, is it could clarify these problems.

The actual semantic problems are largely inherent to the wide application spectrum. Often this can be addressed by a proper selection of a relevant subset. Still, the remaining issues currently need to be resolved through better education, but could be addressed by future revisions of the UML itself.

However, formal verification of UML 2.0 Activities is currently limited to certain small subsets like subsets of UML 1.x Activities where previous works exist. These subsets do not cover the significant enhancements that UML 2.0 brought to Activities and thus are no longer appropriate. Thus, it seems to be essential to address this issue in future work to fully enable the application of UML 2.0 Activities in domains like embedded systems design where formal verification, for example of properties like liveness and boundedness, is often essential.

The syntactic problems can be reduced by the consistent use the provided presentation options. It turns out, that many problems already exist at the abstract syntax level, but can be addressed using an alternative unambiguous concrete syntax.

Future work should address the simplification of existing concepts as well as the enhancement of the concrete syntax, which seems to be the most promising way of resolving many of the presented problems. This may include the design of a usable textual concrete syntax as a complete alternative to Activity Diagrams. The main goal here should be enabling productivity similar to textual programming languages.

References

[1] Bock, C: UML 2 Activity and Action Models: Object Nodes. Journal of Object Technology, Vol. 3 No. 1, 2004.

[2] Bock, C.: UML Without Pictures, In IEEE Computer Special Issue on Model-driven Development, 2003.

[3] Engels, G., Förster, A., Heckel, R., Thöne, S.: Process Modeling using UML. In M. Dumas; W. van der Aalst, A. ter Hofstede (eds.): Process-Aware Information Systems, Wiley Publishing, New York, 2005.

[4] Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Transactions on Software Engineering Methodologlies Vol. 15, No. 1, 2006.

[5] Eshuis, R., Wieringa, R.: A Real-Time Execution Semantics for UML Activity Diagrams. In Proc. FASE 2001, 2001.

[6] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming, 1987.

[7] Hoffman, W., Kirsch, J., Scheer, A.-W.: Modellierung mit ereignisgesteuerten Prozeßketten: Methodenhandbuch; Stand, Dezember 1992. In Veröffentlichungen des Instituts für Wirtschaftsinformatik , no. 101, Universität des Saarlandes, 1992.

[8] Object Management Group, The: Business Process Modeling Notation Specification. OMG dtc/06-02-01, 2006.

[9] Object Management Group, The: OMG SysML Specification, OMG ptc/06-05-04, 2006.

[10] Object Management Group, The: Unified Modeling Language: Superstructure. OMG ad/2005-07-04, 2005.

[11] Mellor, S.J., Balcer, M.J.: Executable UML - A Foundation for Model-Driven Architecture Addison-Wesley, 2002.

[12] Petri, C.: Kommunikation mit Automaten. PhD thesis, Technische Universitat Darmstadt, Germany, 1962.

[13] Raistrick, C., Francis, P., Wright, J.: Model Driven Architecture with Executable UML. Cambridge University Press, 2004.

[14] Schattkowsky, T., Engels, G., Förster, A.: A Model-Based Approach for Platform-Independent Binary Components with Precise Timing and Fine-Grained Concurrency. In Proc. HICSS 2007, 2007.

[15] Schattkowsky, T., Hausmann, J. H., Engels, G.: Using UML Activities for System-on-Chip Design and Synthesis. In Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006), LNCS, Springer, 2006.

[16] Schattkowsky, T., Müller, W., Rettberg, A.: A Generic Model Execution Platform for the Design of Hardware and Software. In G. Martin, W. Müller (eds.): UML for SoC Design. Kluwer, 2005.

[17] Störrle, H., Hausmann, J. H.: Towards a Formal Semantics of UML 2.0 Activities. In Proc. Software Engineering 2005, 2005.

[18] Whiting, P. G., Pascoe, R. S. V.: A History of Data-Flow Languages. In IEEE Annals of the History of Computing archive, Volume 16 , pp. 38-59, 1994.

0-7695-2953-4/07 $20.00 © 2007 IEEE