Analysis of UML Activities using Dynamic Meta Modeling

Gregor Engels, Christian Soltenborn, and Heike Wehrheim

Universität Paderborn, Institut für Informatik,
33098 Paderborn, Germany
{engels,christian,wehrheim}@upb.de

Abstract. Dynamic Meta Modeling (DMM) is a universal approach to defining semantics for languages syntactically grounded on meta models. DMM has been designed with the aim of getting highly understandable yet precise semantic models which in particular allow for a formal analysis. In this paper, we exemplify this by showing how DMM can be used to give a semantics to and define an associated analysis technique for UML Activities.

Key words: UML, semantics, behavior, verification, DMM

1 Introduction

Dynamic Meta Modeling (DMM) [1, 2] has been introduced as a general concept for defining the behavioral semantics of languages syntactically based on *meta models*. Meta models are formalisms for specifying the correct *syntax* of programs (or more generally, models). They allow for a high-level description of syntax abstracting from the concrete way of writing models. This is of particular importance for tool-independent model descriptions and for transformations between models. Meta models have thus become the core instrument in the MDA initiative of the OMG [3]. DMM extends meta models for defining syntax of languages with concepts for describing their dynamic *semantics*. While the primary target of DMM was the UML, the method was designed as to work for any meta model based formalism, thus being universally applicable. The designers set out to define a method which is highly understandable yet formal and precise. The former property was ment for keeping the advantages of visual modeling in the semantics (non-experts should be able to understand semantics); the latter was particularly important for a formal analysis of models.

In this paper, we exemplify the DMM's ability of allowing for a formal analysis by means of defining an automatic analysis technique for UML Activities. For doing so, we first give a DMM semantics to UML Activities following [2]. DMM is conceptually based on *graph transformations* [4], which fits well to the visual appeal of Activities themselves, and more generally of meta models. In contrast to previous approaches to giving semantics for UML Activites [5–7], DMM is able to precisely formalise the intricate traverse-to-completion semantics of Activities [8]. The semantic domain for Activities are *transitions systems*

whose states are graphs representing the Activities and their current runtime states. The use of the general domain of transition systems allows for a direct application of concepts for comparing models (using notions of equivalence on transition systems) as well as specifying properties of models (e.g. via temporal logics interpreted on transition systems).

The definition of the semantics is the basis for the subsequent development of an automatic analysis technique. Rather than analyzing for individual properties of particular models, we are interested in defining a general quality criterion for Activities. To this end, we identify properties of Activities which characterise "good" models in the main application area of Activities, namely *workflow modeling*. Workflows describe business processes in companies. Activities modeling workflows have to adhere to particular requirements, some of which can be syntactically checked (e.g. whether there is a unique initial and a unique final action) but others referring to the execution of Activities (viz. their semantics). Following an approach of van der Aalst [9] we develop a correctness criterion called *soundness* covering several crucial properties of Activities modeling workflows. Soundness is defined on the particular form of transition systems generated by the DMM semantics for UML Activities.

Our objective is then to get a *fully automatic* check for soundness. Starting from an Activity modeling a workflow, the soundness analysis should essentially be carried out by tools. Instead of building a new tool from scratch, we choose an existing tool (GROOVE [10]) as the basis for our analysis. GROOVE allows for the construction, simulation and verification of transition systems specified via graphs and graph transformations. Verification currently includes CTL model checking [11]. The use of GROOVE thus necessitates a transformation of our soundness criterion into CTL formulas which are then checked on the generated transition systems. We prove correctness of this transformation as to ensure analysis of the correct property.

The paper is structured as follows. The next section gives an introductory example of an UML Activity modeling a workflow. On this we will informally discuss our soundness criterion in general, and already formally define those parts referring to the syntax. Section 3 explains the approach of Dynamic Meta Modeling, and defines the semantics for UML Activities. Section 4 is concerned with the verification of soundness: we give the formal definition of sound Activities by means of their DMM specification. The transformation of soundness into CTL formulas is the main topic of section 5: we show how to perform the transformation and prove its correctness. Additionally, the section explains the usage of tools, in particular GROOVE. The last section concludes and discusses related work.

2 The Idea of Soundness

The purpose of this section is to introduce our notion of workflow modeling using UML Activities, and to discuss the soundness property in the context of that definition. Recall from the introduction that we have chosen soundness as

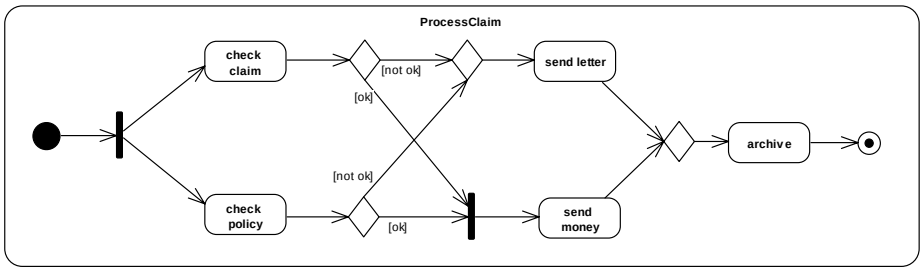

Fig. 1. Workflow "Process claim" as a UML Activity.

a generic indication of quality: according to van der Aalst [9, 12], every workflow should be sound, regardless of its concrete semantic domain.

We use a workflow which will serve as a running example for the rest of our paper; it describes the processing of an insurance claim in a strongly simplified way and is depicted as a UML Activity in figure 1. The meaning of figure 1 is supposed to be as follows:

> If a claim arrives at an insurance company, two things need to be checked: does the customer have an appropriate policy, and is the claim itself valid? To speed up processing of the claim, the checks are performed in parallel (**check policy** and **check claim**). Only if both checks succeed, money will be sent to the customer (**send money**). If at least one of the checks fails, a letter will be sent to the customer explaining why the claim has been rejected (**send letter**). At the very end, the claim is archived (**archive**).

Before we present our criteria for sound UML Activities, we need to give a basic idea of their semantics (we will look into this in more detail in section 3). The UML specification [13] states that "Activities have a Petri-like semantics", i.e., the semantics is based on token flow. When an Activity is executed, the `InitialNode` (solid circle) creates a token, which corresponds to a case to be handled by the workflow. That token is then routed through the Activity. Tasks are depicted by rounded rectangles (they are called `Actions` in the UML terminology). `ForkNode` and `JoinNode` (vertical bars) represent parallelity, i.e., they copy respectively join the arriving tokens. `DecisionNode` and `MergeNode` (diamonds) are used to route tokens. `ActivityFinalNodes` (dotted circle) consume arriving tokens.

The reader not familiar with UML Activities should note the ease of understanding figure 1. The expert might notice that we only use a subset of UML Activities, i.e., the `FundamentalActivities`, `BasicActivities` and `IntermediateActivities` packages, since the elements of these packages suffice to model many kinds of workflows.

At first glance the presented workflow seems reasonable. But what about its objective quality? Or, more generally: what properties should an arbitrary

workflow at least have to be considered high-quality? In the following, we discuss the soundness property suggested by van der Aalst [9, 12]. In his opinion, every workflow should fulfill some basic requirements:

1. The workflow should have well-defined pre- and postconditions.
2. The workflow should not contain any useless elements.
3. If the end condition is reached, no more tasks should be processed.
4. The end condition should finally be reached.

Requirements 1, 2, and 4 are self-explanatory. For requirement 3, assume that tasks are still processed after the end condition has been reached: these tasks obviously do not contribute to the result of the workflow. The work involved in performing these tasks is therefore wasted.

Taking the semantics described above into account, it is straightforward to translate van der Aalst's soundness definition into the world of UML Activities. A UML Activity is considered to be sound if the following conditions hold:

1. The Activity must have exactly one `InitialNode` and `ActivityFinalNode`.
2. Any `Action` must be executed under at least one possible execution of the Activity.
3. If a token arrives at the `ActivityFinalNode`, no more tokens are left in the Activity.
4. A token finally arrives at the `ActivityFinalNode`.

Note that in practice, requirement 1 does not restrict the modeler: more than one `InitialNode` can be modeled equivalently by one `InitialNode` and a `ForkNode` producing the desired number of tokens (`ActivityFinalNode` and `JoinNode` accordingly).

The requirements formulated above put restrictions on both the syntax and the semantics of a sound Activity: requirement 1 restricts the structure, and the other requirements restrict how the Activities must behave to be considered sound.

Since structural restrictions are usually easy to verify, their verification will not be discussed further. The behavioral restrictions are more interesting: to verify them, we need a formal semantics of the behavior of UML Activities. In the next section, we will dicuss the definition of such a semantics by means of Dynamic Meta Modeling (DMM). Section 4 will then show how this semantics can be used to formalize the behavioral restrictions, and section 5 will show how to verify the restrictions in an automatic way.

3 Dynamic Meta Modeling

The most important prerequisite for automatically analyzing the behavior of models is that the behavior is specified formally. Moreover, to allow advanced language users to understand the precise semantics of their models, the specification should be as easily understandable as possible. Dynamic Meta Modeling aims at fulfilling these seemingly contradictory requirements by combining two

different approaches into one semantics description technique: *denotational modeling* and *operational rules*.

DMM is targeted at languages having an abstract syntax which is defined by means of a *meta model* as suggested by the OMG, i.e., a model describing the elements the language itself consists of. Sentences of the language must then be consistent with the meta model. Often MOF [14] is used for the specification of meta models, which is basically a subset of UML class diagrams. To follow the OMG layered model, the language's meta model is level M2, the level of the concrete syntax (i.e., an object diagram consistent to the class diagram of level M2) is M1, and the visualization of the concrete syntax (in our case, the picture of the UML Activity) is level M0.

In DMM, the static semantics of a language is specified using *Denotational Meta Modeling*. This means that the semantic domain has its own meta model, to which the meta model describing the Visual Modeling language is mapped. The meta model of the semantic domain often is an enhanced version of the meta model of the language itself. For example, we will see below that the Activity's semantic domain meta model has additional elements like `Token` and `Offer`, which allow to express certain states of execution of the Activity under consideration.

The dynamic semantics is then specified by developing a set of operational rules which describe how instances of the semantic domain meta model change in time. For this, the instances are mapped to *typed graphs* [15], i.e., graphs whose nodes are typed over the semantic domain meta model. The operational rules are then defined as *graph transformation rules*, working on the derived typed graphs.

Since the typed graphs represent states of execution of the Activity, the described specification technique allows for the computation of transition systems representing the precise behavior of the investigated models. The operational rules result in transitions between these states. The resulting transition systems can then be verified for certain properties, as we will see in section 4. The overall concept of DMM is depicted in figure 2.

In the following, we give insight into our DMM semantics specification of UML Activities. Note that we did not specify a semantics for all Activity elements as defined in the UML 2.0 specification [13] yet. We basically implemented the the `FundamentalActivities`, `BasicActivities` and `IntermediateActivities` packages.

Figure 3 shows an excerpt of the semantic domain meta model we have developed to express the behavior of Activities (elements depicted in bold are enhancements to the original meta model). While developing that meta model, we have followed the textual description of the Activity's semantics provided as part of the UML specification [13]. Most importantly, the following concepts have been implemented:

- The execution of an Activity is controlled by the class `ActivityExecution`, which is a composition of the elements needed to describe the states of execution (see below).

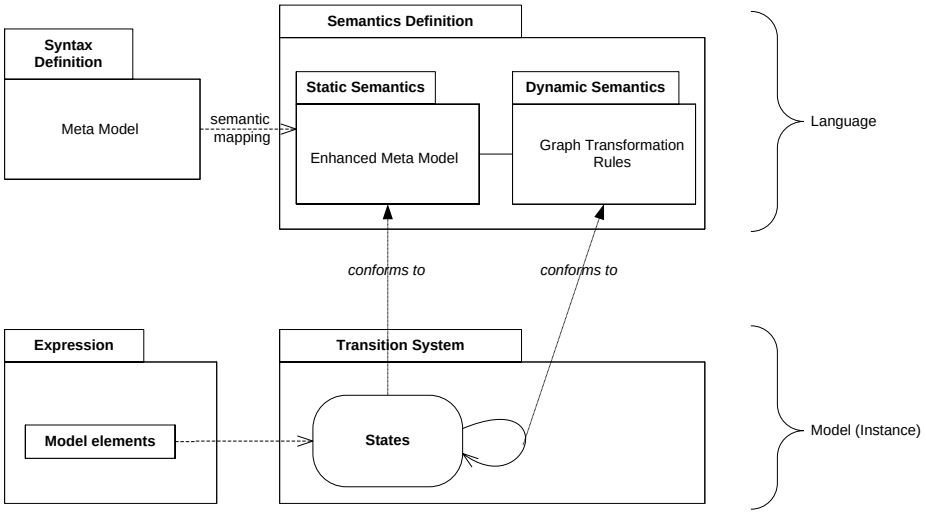

Fig. 2. Overview of the DMM approach

- As expected, the token flow is realized by introducing a `Token` class. Since according to the UML specification, a token can only rest at a subset of the Activity elements, an abstract class `BufferNode` is added to the type hierarchy of those elements.
- The token flow within Activities follows the concept of *traverse-to-completion*. In a nutshell, this means that tokens are only *offered* to edges. An offer traverses the Activity up to the next `BufferNode`, moving its token only if such a node is found. In this way, tokens can not get "stuck" within the Activity in some sense. This behavior is implemented by the `Offer` class and a couple of other constructs.

Figure 4 shows an example DMM rule implementing the semantics of the `DecisionNode`. A DMM rule consists of a signature, a number of pre- and postconditions and an optional number of invocations of other DMM rules (note that the presented rule does not have invocations). Slightly simplified, the rule matches an instance graph if a morphism from the preconditions into the instance graph can be found. If this is the case, the graph will be modified: elements marked {new} are created, and elements marked {destroyed} are deleted. In our case, the offer on the incoming edge will be deleted, and a new offer will be created on the outgoing edge, corresponding to the fact that the offer has passed the `DecisionNode`. Figure 5 illustrates this process: the left part shows a visualization of the Activity's behavior, the right part shows the matching part of our example model before and after applying the rule of figure 4.

The derived graph represents the next state of execution. Since a `DecisionNode` has only one incoming, but several outgoing edges, an arriving offer will be routed to all of them: the rule matches every combination of the only incoming and one of the outgoing edges, and produces several new states. From the

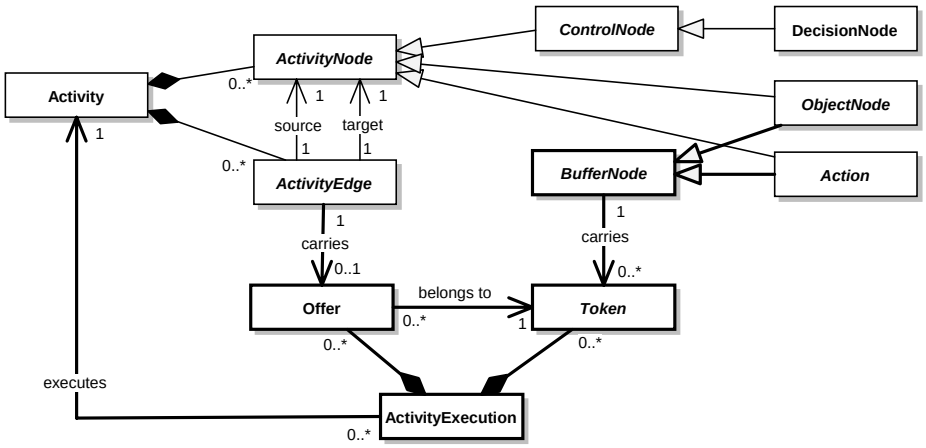

Fig. 3. Enhanced UML Activity meta model

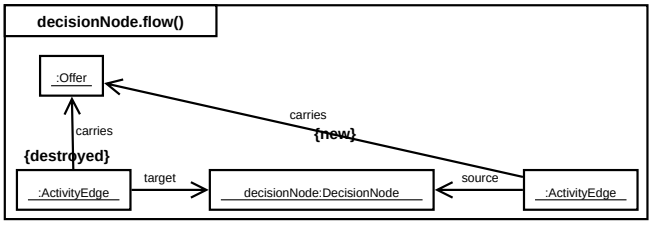

Fig. 4. DMM rule decisionNode.flow()

transition system's point of view, the result is a branch, representing all possible executions of the Activity at that point. Figure 5 shows one possible application of rule decisionnode.flow(). In this case, the offer is routed along the top edge of the `DecisionNode`. The right part of that figure shows two consecutive states of the Activity.

The resulting transition system represents the complete behavior of the Activity under consideration. It will be the basis for analysis of the Activity, using standard techniques such as model checking. In the next section, we show how to verify a transition system representing a concrete Activity for soundness.

4 Sound UML Activities

Up to now, we have informally defined soundness for UML Activities in section 2, and we have introduced our formal semantics of Activity's behavior in the last section. In the following, we will use this semantics to formally define soundness, and we will then translate the soundness conditions into CTL formulas in order to be able to verify the formula's validity with a model checker.

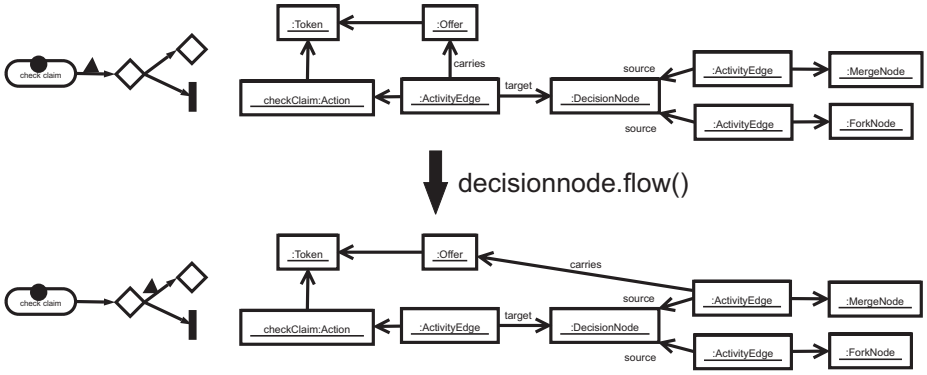

Fig. 5. Application of rule decisionNode.flow()

Our goal is to be able to make statements about states of execution of the Activity under investigation. We do this in an indirect way: we do not speak about the states themselves, but about rules which match states. As we have seen in the last section, a rule only matches a state (i.e., a graph) if a morphism between the preconditions of the rule and the state can be found. In other words: if a rule matches a state, we know that the preconditions of that rule hold within the state, which means that we have knowledge about the state itself. As we will see in section 5, the model checker provided by GROOVE [16] works exactly this way: it verifies CTL formulas where the atomic propositions are applications of the graph transformation rules used to calculate the transition system under investigation.

Note that the described approach does not restrict the verification process: every property p which can be formulated as a precondition of a rule can be verified by adding a special rule r which has p not only as its pre-, but also as its postcondition (i.e., its application does not change the state). A state s fulfilling p will result in a self-transition (s, s) labeled r. Therefore, if we assume some reasonable kind of *fairness* (see e.g. [17]), checking for the application of r is equivalent to finding a state for which p holds.

Before we present our definition of soundness, we need to introduce the idea of some DMM rules from our semantics definition, and we need to define some predicates. Note that from now on, we slightly simplify our original results (see [18] for a more comprehensive coverage of the content of this section).

First, recall from section 2 that for an Activity to be considered sound, a token must finally arrive at the `ActivityFinalNode`, and at that moment, no other tokens must be left in the Activity. Recall also that `ActivityFinalNodes` consume all arriving tokens. To implement this behavior, we have defined two rules whose task is to destroy arriving tokens as desired: finalnode.destroyToken1() and finalnode.destroyToken2(). Both rules match if a token arrives at an `ActivityFinalNode`; the difference between them is that the former only matches if ex-

actly one token is flowing within the (whole) Activity, and the latter matches if two or more tokens are flowing. We can use this difference to define the desired behavior: an Activity is sound if, under all its possible executions, rule finalnode.destroyToken1() matches at some point in time, and rule finalnode.destroyToken2() never matches. Note that the rules' implementation guarantees that whenever these rules are applied, no other rules can be applied afterwards. This is in compliance with the UML specification which says that a token arriving at an `ActivityFinalNode` immediately ends the Activity.

The other requirement for soundness is that a sound Activity does not contain any useless elements. Since `Actions` are the elements of Activities where actual work is performed, we slightly relax that requirement by only requiring no useless `Actions`. In our semantics, the execution of an `Action` is mainly implemented by the rule N.start(), where N is the name of that `Action` (note that every `Action` has its own rule). We therefore define an Activity to be sound if, under all its possible executions, the rule N.start() matches at some point in time for every `Action` of the Activity.

Having said that, we need to define some predicates which will prove helpful when formalizing our soundness definition:

Definition 1 Let r be a DMM rule, s a state of a UML Activity as described in section 3, i.e., a graph which is typed over the enhanced meta model. Let v be a vertex of that graph.

1. If r matches the state s, then $matches(r, s)$ is true.
2. If v's type is `Action` or a subtype of `Action`, then $isAction(v)$ is true.
3. Let $isAction(v)$ be true. Then $name(v)$ represents the name of the `Action` represented by v.
4. If s is the state derived from an application of rule finalnode.destroyToken1(), then $isFinal(s)$ is true.

Now we are ready to present our formal definition of soundness for UML Activities.

Definition 2 (Sound Activity) Let A be a UML Activity with exactly one `InitialNode` and `ActivityFinalNode`, s_0 the state of A with only a token on the `InitialNode`, and V_{s_0} the vertices of the graph s_0. Let $TS = (S, \rightarrow, s_0)$ be the transition system induced by the DMM rule set as described in section 3 (S contains exactly those states reachable from s_0). A is *sound* if and only if the following conditions hold:

1. $\forall s \in S : (\exists s' \in S : s \rightarrow^* s' \wedge matches(\text{finalnode.destroyToken1}(), s')) \vee isFinal(s)$
2. $\forall s \in S : \neg matches(\text{finalnode.destroyToken2}(), s)$
3. $\forall v \in V_{s_0} : isAction(v) \wedge name(v) = N \Rightarrow \exists s \in S : matches(N.\text{start}(), s)$

Let us briefly discuss the relation between the informal soundness definition from section 2 and definition 2. Condition 1 ensures that from all states s, a state s' is reachable such that rule finalnode.destroyToken1() can be applied to it. If this

is the case, we know that a token will finally reach the `ActivityFinalNode`. As rule finalnode.destroyToken2() is never applied, we also know that at this point in time, no other token is left in the Activity. The predicate $isFinal(s)$ of condition 1 takes care of the state derived from applying rule finalnode.destroyToken1(): it is needed because since no other rule can be applied to s (see above), the first part of the condition does not hold. Condition 3 makes sure that for every `Action`, a state $s \in S$ exists where that `Action` is executed. Since S contains all states reachable from s_0 (and no more), we know that all `Actions` are executed under at least one of the possible executions of A.

5 Utilizing the GROOVE Toolset

Our final goal is to have an automatic check for soundness. Hence we need a tool which, given a set of graph transformation rules, can generate the transition system according to our semantics and inspect it with respect to our conditions. For this, we have chosen to use the tool GROOVE. GROOVE is a shortcut for "GRaphs for Object-Oriented VErification" and has been developed by Arend Rensink at the University of Twente [19]. It offers a collection of tools for handling graph transformations: the Generator computes a transition system out of a start graph and a set of graph transformation rules, the Editor allows to edit the graphs and rules, and the Imager visualizes them. The Simulator integrates these tools, and the Model Checker allows for the verification of CTL formulas over the generated transition systems. As expected, we mainly utilize GROOVE's Generator and Model Checker (see figure 6 for the complete workflow).

To use the model checker for checking soundness, we first need to translate the conditions of definition 2 into CTL, i.e., the notion of temporal logic GROOVE understands. Note that the Model Checker works as described in the last section: it verifies CTL formulas over the *application* of graph transformation rules. Before we can translate our soundness definition into the language GROOVE understands, we need some prerequisites. We start by defining the *computations* of a transition system:

Definition 3 (Computations) Let $TS = (S, \rightarrow, s_0)$ be a DMM transition system as defined in definition 2. The set of *computations* $Comp(s_0)$ is defined as follows:

$$Comp(s_0) := \{s_0 s_1 s_2 \ldots : (s_i, s_{i+1}) \in \rightarrow\}$$

$Comp(s_0)$ contains all possible computations starting with state s_0.

Now we briefly define the CTL formulas we will use to express our conditions. Note that this is only a subset of CTL.

Definition 4 (CTL formulas) Let $TS = (S, \rightarrow, s_0)$ be a DMM transition system as defined in definition 2. Let $Comp(s_0)$ be the set of computations as in definition 3, and let p be some atomic proposition. Then

$$TS \models \mathbf{AF}(p) :\Leftrightarrow \forall s_0 s_1 \cdots \in Comp(s_0) \exists k \in \mathbb{N} : p \text{ holds in } s_k$$
$$TS \models \mathbf{AG}(p) :\Leftrightarrow \forall s_0 s_1 \cdots \in Comp(s_0) \forall k \in \mathbb{N} : p \text{ holds in } s_k$$
$$TS \models \mathbf{EF}(p) :\Leftrightarrow \exists s_0 s_1 \cdots \in Comp(s_0) \exists k \in \mathbb{N} : p \text{ holds in } s_k$$

In case of finite computations, k accordingly has to be restricted to the length of the computation. **AF** stands for "On **A**ll paths **F**inally...", **AG** stands for "On **A**ll paths **G**lobally..." and **EF** stands for "There **E**xists a path such that **F**inally...".

We are now ready to formulate our theorem.

Theorem 5 Let A be a UML Activity with exactly one `InitialNode` and `ActivityFinalNode`, s_0 the state of A with only a token on the `InitialNode`. Let $N_1, \ldots, N_k$ be the names of the `Actions` contained in A. Let $TS = (S, \rightarrow, s_0)$ be the transition system induced by the DMM rule set as described in section 3 (S contains exactly those states which are reachable from s_0). A is sound if and only if the following CTL formulas hold for TS:

1. $TS \models \mathbf{AF}(\mathsf{finalnode.destroyToken1()})$
2. $TS \models \mathbf{AG}(\neg \mathsf{finalnode.destroyToken2()})$
3. $TS \models \mathbf{EF}(N_1.\mathsf{start()}) \wedge \cdots \wedge \mathbf{EF}(N_k.\mathsf{start()})$

Proof. We start by showing the equivalence of the first condition of definition 2 and theorem 5. First, we can also write

$$\forall s \in S : (\exists s' \in S : s \rightarrow^* s' \wedge matches(\mathsf{finalnode.destroyToken1()}, s')) \vee isFinal(s)$$

as

$$\forall s \in S : \ (\exists k \in \mathbb{N} : s \rightarrow s_1 \rightarrow \cdots \rightarrow s_k \ \wedge \ matches(\mathsf{finalnode.destroyToken1()}, s_k)) \\ \vee isFinal(s)$$

Since the above holds for all states $s \in S$, S contains all states reachable from the initial state, and rule $\mathsf{flowfinal.destroyToken1()}$ is always the last rule in a computation, we can write this as

$$\forall s_0 s_1 \cdots \in Comp(s_0) \exists k \in \mathbb{N} : matches(\mathsf{finalnode.destroyToken1()}, s_k)$$

Following definition 4, we can now formulate our property as a CTL formula over the application of rule $\mathsf{flowfinal.destroyToken1()}$:

$$TS \models \mathbf{AF}(\mathsf{finalnode.destroyToken1()})$$

Now it is easy to see how conditions 2 and 3 of definition 2 can be translated into temporal logic: since condition 2 holds for all states, it must also hold for all computations. We can therefore write the condition as

$$\forall s_0 s_1 \cdots \in Comp(s_0) \forall k \in \mathbb{N} : \neg matches(\mathsf{finalnode.destroyToken2()}, s_k)$$

As above, this can be translated into temporal logic:

$$TS \models \mathbf{AG}(\neg \mathsf{finalnode.destroyToken2()})$$

The last condition of definition 2 states that for all `Actions`, there is some state $s \in S$ such that the start rule of that very `Action` matches. Let $N_1, \ldots, N_k$ be the names of the `Actions`. Since we do not know anything about state s except that it exists, we can only conclude:

$$\exists s_0 s_1 \cdots \in Comp(s_0) \exists k \in \mathbb{N} : matches(N_i.\mathsf{start}(), s_k)$$

$(i = 1, \ldots, k)$. We can again translate this into CTL:

$$TS \models \mathbf{EF}(N_i.\mathsf{start}())$$

□

Now we have everything needed to verify our running example introduced in section 2 for soundness. As it turns out, the example is not sound: the GROOVE model checker reports that conditions 1 and 2 of theorem 5 do not hold. In other words: there are paths within the transition system where rule finalnode.destroyToken1() is never applied, and also states where rule finalnode.destroyToken2() matches. Intuitively, this means that there is at least one situation where a token arrives at the `ActivityFinalNode`, but there are more tokens left in the Activity.

A further investigation of the example shows the reasons for this: first, if one of the checks succeeds and the other fails, we end up in a state where a token is stuck at the `JoinNode`. For this situation, both conditions do not hold: the arriving token is consumed by applying rule finalnode.destroyToken2(), and since the other token is stuck, it will not be consumed (in particular not by applying rule finalnode.destroyToken1()).

Second, if both checks fail, two tokens will arrive at the `ActivityFinalNode`, again violating condition 2 of theorem 5. Note that in this case, two letters will be sent to the customer, which is obviously not the desired behavior. Note also that the second token will be consumed by applying rule finalnode.destroyToken1() (i.e., condition 1 is not violated).

It remains to discuss the chain of tools we developed for the automatic verification of Activities. As mentioned in the introduction, DMM has been developed mainly having the UML in mind. This is reflected in figure 6 by using the UML meta model both as input for the semantics editor and the modeling tool: the purpose of the former is to develop operational rules by means of graph transformation rules which are typed over an extension of that meta model (as we have done for UML Activities). The latter utilizes the meta model to verify the syntactical correctness of the models. Note that this might be a typical use of DMM, but basically every meta model can be incorporated into a DMM-based semantics definition.

The semantics editor delivers a complete semantics to the property checker, including a mapping from the original into the enhanced meta model. This mapping is then used to transform a model instance (in our case: a UML Activity) into a start graph typed over the enhanced meta model.

The start graph as well as the graph transformation rules serve as input for the GROOVE generator. As the name suggests, a transition system is generated,

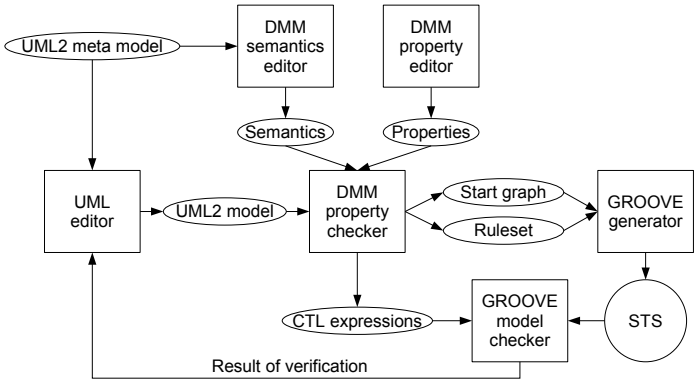

Fig. 6. Tool chain

having graphs as states and transitions between these states. The transitions are labeled with the name of the applied rule.

The generated transition system represents the complete semantics of the input model. It can either be investigated manually by visualizing it with the GROOVE simulator (e.g. to understand the precise semantics of a certain part of a model), or automatic verification techniques can be used.

For the latter, the first step is to identify the properties to be modified, and to formulate them with the help of the DMM property editor. The property checker then generates a set of rules and a start graph, from which the GROOVE generator will compute the transition system. Next, the GROOVE model checker is utilized to verify whether the properties hold on the computed transition system (for the example of figure 1, the whole procedure took 13.7 seconds). The result of the verification process can then be used by the modeler to improve her model.

6 Conclusion

In this paper we have shown how to use dynamic meta modeling for 1) defining a semantics for a modeling formalism based on a given meta model, and 2) carrying out an analysis on the resulting semantics. As application, we used UML Activities which pose a particular challenge for semantics definitions due to their traverse-to-completion behavior introduced in UML 2.0. For the analysis, we have chosen a general quality criterion (soundness) for workflows which are a typical modeling domain for UML Activities. As a result, we now have a tool chain which allows for the automatic analysis of soundness for workflows (using the GROOVE toolset for the generation of the transition system as well as for model checking).

Related work. There have been several apporaches to define a formal semantics for UML Activities. Störrle et al. [20, 21, 5] try to use Petri nets as the semantic

domain for Activities: they conclude that due to the traverse-to-completion semantics introduced in UML 2.0, a mapping from Activities into Petri nets is not possible. Eshuis [22] translates Activity Diagrams from UML 1.5 into the input language of the model checker NuSVM [23], giving Activities a statechart-like semantics as stated by the UML 1.5 specification, and uses that semantics for verification of certain properties. Similarly, [6] and [7] do not treat UML 2.0 Activities, but their 1.5 predecessors.

Our approach is different to the ones described above in two ways: first, we use DMM to define the dynamic semantics of UML Activities. The resulting specification is formal *and* easily understandable and can therefore not only be used for automatic analysis of models at design-time, but also as reference for advanced language users. Note that our specification implements the traverse-to-completion concept for token flow as suggested by the UML 2.0 specification.

Second, our analysis technique for DMM-based semantics allows to easily formulate requirements on the models under consideration: all needed is an understanding of UML object diagrams as well as a basic knowledge in CTL. For instance, to make sure that a certain object structure does never occur when executing a model, that object structure is formulated as an object diagram which serves as the pre- and the postcondition of a DMM rule R (as described in section 4). Using a basic CTL construct, the whole requirement can then be formulated like this: **AG**(**NOT** R).

Outlook. Since DMM can be used as a semantics specification technique for every visual modeling language based on meta-models, we plan to explore different applications of the described analysis technique. First, the integration of different languages will be explored, e.g. for consistency checks. Second, we are interested in the definition of domain specific languages as extensions of already existing languages. In this area, we plan to use our analysis technique to ensure that extensions of languages describing behavior do not *break* the behavior of the base languages.

References

1. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In Evans, A., Kent, S., Selic, B., eds.: UML 2000 - The Unified Modeling Language, Berlin, Springer-Verlag (October 2000)
2. Hausmann, J.H.: Dynamic Meta Modeling. PhD thesis, University of Paderborn (2005)
3. OMG: Model Driven Architecture. `http://www.omg.org/mda/`
4. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools. World Scientific Publisher (1999)
5. Störrle, H.: Semantics of Control-Flow in UML 2.0 Activities. In: VL/HCC, IEEE Computer Society (2004) 235–242
6. Bolton, C., Davies, J.: On Giving a Behavioural Semantics to Activity Graphs. In: UML 2000 - Online proceedings. (2000)
7. Börger, E., Cavarra, A., Riccobene, E.: An ASM Semantics for UML Activity Diagrams. In Rus, T., ed.: AMAST. Number 1816 in LNCS (2000) 293–308
8. Hausmann, J.H., Störrle, H.: Towards a Formal Semantics of UML 2.0 Activities. Software Engineering 2005 **P-64** (2005) 117–128
9. van der Aalst, W.: Verification of Workflow Nets. In: ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets, London, UK, Springer-Verlag (1997) 407–426
10. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz, J.L., Nagl, M., Böhlen, B., eds.: AGTIVE. Volume 3062 of Lecture Notes in Computer Science., Springer (2003) 479–485
11. Clarke, E., Emerson, E., Sistla, A.: Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In: Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, ACM (1983) 117–126
12. van der Aalst, W., van Hee, K.: Workflow Management - Models, Methods, and Systems. The MIT Press (2002)
13. Object Management Group: UML Specification V2.0. `http://www.omg.org/technology/documents/modeling_spec_catalog.htm` (2005)
14. Object Management Group: The MOF Specification. `http://www.omg.org/cgi-bin/doc?formal/00-04-03` (2004)
15. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Padberg, J.: The Category of Typed Graph Grammars and its Adjunctions with Categories. In Cuny, J.E., Ehrig, H., Engels, G., Rozenberg, G., eds.: TAGT. Volume 1073 of Lecture Notes in Computer Science., Springer (1994) 56–74
16. Kastenberg, H., Rensink, A.: Model checking dynamic states in GROOVE. In Valmari, A., ed.: Model Checking Software (SPIN). Volume 3925 of Lecture Notes in Computer Science., Springer-Verlag (2006) 299–305
17. Kindler, E., van der Aalst, W.: Liveness, Fairness, and Recurrence in Petri Nets. Inf. Process. Lett. **70**(6) (1999) 269–27
18. Soltenborn, C.: Analysis of UML Workflow Diagrams with Dynamic Meta Modeling techniques. Master's thesis, University of Paderborn (2006)
19. Rensink, A.: GROOVE: A Graph Transformation Tool Set for the Simulation and Analysis of Graph Grammars. Available at `http://www.cs.utwente.nl/~groove` (2003)

20. Störrle, H., Hausmann, J.H.: Towards a Formal Semantics of UML 2.0 Activities. In Liggesmeyer, P., Pohl, K., Goedicke, M., eds.: Software Engineering. Volume 64 of LNI., GI (2005) 117–128
21. Störrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. Electr. Notes Theor. Comput. Sci. **127**(4) (2005) 35–52
22. Eshuis, R.: Symbolic model checking of UML Activity diagrams. ACM Trans. Softw. Eng. Methodol. **15**(1) (2006) 1–38
23. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An Opensource Tool for Symbolic Model Checking. In Brinksma, E., Larsen, K.G., eds.: CAV. Volume 2404 of Lecture Notes in Computer Science., Springer (2002) 359–364