Electronic Communications of the EASST
Volume 30 (2010)

International Colloquium on Graph and Model Transformation On the occasion of the 65th birthday of Hartmut Ehrig (GraMoT 2010)

Test-driven Language Derivation with Graph Transformation-Based Dynamic Meta Modeling

Gregor Engels and Christian Soltenborn

18 pages

Guest Editors: Claudia Ermel, Hartmut Ehrig, Fernando Orejas, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Test-driven Language Derivation with Graph Transformation-Based Dynamic Meta Modeling

Gregor Engels and Christian Soltenborn

University of Paderborn
{engels,christian}@uni-paderborn.de

Abstract: Deriving a new language L_B from an already existing one L_A is a typical task in domain-specific language engineering. Here, besides adjusting L_A's syntax, the language engineer has to modify the semantics of L_A to derive L_B's semantics. Particularly, in case of *behavioral* modeling languages, this is a difficult and error-prone task, as changing the behavior of language elements or adding behavior for new elements might have undesired *side effects*.

Therefore, we propose a *test-driven language derivation process*. In a first step, the language engineer creates example models containing the changed or newly added elements in different contexts. For each of these models, the language engineer also precisely describes the expected behavior. In a second step, each example model and its description of behavior is transformed into an *executable test case*. Finally, these test cases are used when deriving the actual semantics of L_B - at any time, the language engineer can run the tests to verify whether the changes he performed on L_A's semantics indeed produce the desired behavior.

In this paper, we illustrate the approach using our graph transformation-based semantics specification technique *Dynamic Meta Modeling*. This is once more an example where the graph transformation approach shows its strengths and appropriateness to support software engineering tasks as, e.g., model transformations, software specifications, or tool development.

Keywords: language engineering, semantics, testing, DMM, graph transformation

1 Introduction

In today's world of software engineering, *domain-specific modeling languages* (DSLs) have become an important tool. A DSL is a language which has been created for the sake of being used in a certain, usually narrow domain. The language elements are abstractions of the domain's important concepts. As a result, DSLs are usually intuitive to understand and therefore well-suited as a base for the communication with the stakeholder's domain experts.

Moreover, the intuitive understandability of well-designed DSLs also results in models of a higher quality: Abstraction is always a difficult task, also for modelers. In case of DSLs, a (hopefully large) part of the necessary abstraction process has been performed at language creation time and therefore does not need to be performed by the modeler again, who can concentrate on modeling the actual business logic.

However, defining a DSL from scratch is not an easy task. The language engineer does not only need to specify the abstract and concrete syntax of the language, but also its semantics. The latter can be quite difficult, especially for languages describing behavior.

As a result, in case a language exists which has similar properties as the envisioned one, it would be desirable to reuse that language as a starting base. In this way, the language engineer can rely on an existing, proven language core, and can concentrate on performing the modifications needed for the target DSL.

In this paper, we describe a scenario of language derivation in the context of *Dynamic Meta Modeling* (DMM), a graph transformation-based semantics specification technique developed at our research group [EHS99, Hau05]. The idea is to enhance an existing language with domain-specific concepts, and to add the semantics of the new concepts to the already existing DMM semantics specification.

DMM aims at two seemingly contrary goals: DMM specifications shall be *easily understandable* and *formal*. The only prerequisite for the usage of DMM is that the language's abstract syntax is defined by means of a *metamodel*. DMM rules are (annotated) UML object diagrams, i.e., instances of the language's metamodel. As a result, due to the visual, familiar notion of DMM rules, language engineers familiar with that metamodel can easily read DMM specifications.

Behind the hoods, DMM rules are *graph transformation rules* [Roz97]. In a nutshell, this means that a DMM rule manipulates graphs. The idea is that graphs as well as rules are *typed* over the language's metamodel, and that DMM rules are used to describe changes on instances of that metamodel (and therefore behavior).

Finally, given a certain language's instance (i.e., a model) and a DMM specification, a transition system can be computed, where states are states of execution of the model, and transitions are applications of DMM rules. The transition system reflects the complete behavior of the model; it can then e.g. be analyzed with model checking techniques [ESW07]. A more detailed introduction to DMM will be given in Sect. 2.

Now, given a language equipped with a DMM specification, and provided that that language is suited as base for deriving a new language by enhancement, the language engineer has to add DMM rules to the existing DMM ruleset which define the new language element's semantics. An often occuring problem in object-oriented scenarios is that some behavior defined in the context of a certain type shall not be applied in the context of a subtype, i.e., behavior has to be *overridden* by new behavior. In [EFS09], we have introduced a notion of *rule overriding* exactly suited for this situation, which we will use to cope with that problem.

Finally, the goal must be to perform all language changes such that the semantics of the source language is not hampered, and that the new parts of the semantics correctly reflect the intentions of the language engineer.

Since a DMM specification is formal, the first and obvious idea is to formalize that notion of correctness by means of requirements the specification shall fulfill, and then to *prove* that this is indeed the case. However, the experiences from software development seem to imply that proving the correctness of a reasonable complex system is often just not feasible; therefore, the most important technique in software quality assurance is *testing*.

In [SE09] we have suggested a pragmatic approach to help creating high-quality semantics, which is inspired by the well-known approach of *Test-Driven Development* [Bec02]. It is moti-

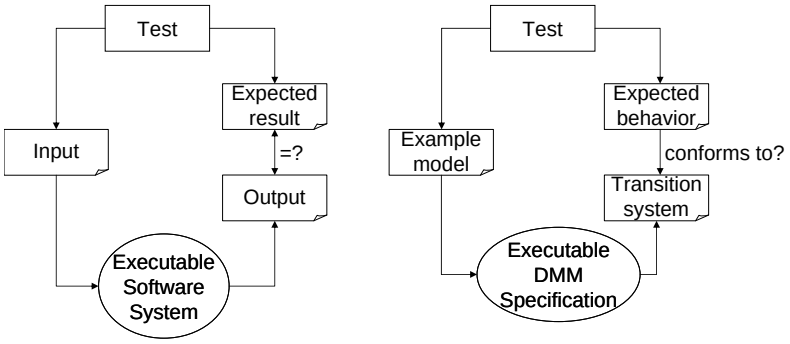

Figure 1: Comparison of testing of software systems (left) and semantics specifications (right); the test subject is depicted in an oval.

vated by the fact that a semantics specification basically follows the Input-Process-Output (IPO) model, where a certain model can be seen as the input, and the semantics of that model is the output (e.g., represented as a transition system).

Figure 1 shows our approach and its relation to the testing of software systems. In case of testing software systems, a test case consists of some input for the system and the system's expected result. The test succeeds if the actual output of the system is equal to the expected result.

In contrast to that, we want to test a DMM specification. Therefore, a test consists of an example model and its expected behavior. From that model and the DMM specification, a transition system can be computed which represents the model's behavior. The test succeeds if the actual behavior conforms to the expected behavior, i.e., if the expected behavior (and *only* the expected behavior) is contained in the transition system.

In this paper, we show how to apply the approach of test-driven semantics specification within the scenario of language derivation. For this, we will discuss a small example of language enhancement: We will enhance the language of UML Activities. While doing this, we will point out some problems, and we will show how DMM rule overriding can help to solve these problems, and how a test-driven approach can be used that the new language's semantics has a certain quality.

Structure of Paper In the next section, we will give an introduction to our semantics specification technique Dynamic Meta Modeling. Based on that, in Sect. 3 we will introduce a small example of language modification, discuss side effects of that modification, introduce our approach of test-driven semantics specification, and discuss how that approach can be used as a "safety net" against such side effects when performing language derivation. Section 4 will point out some work related to ours, and Sect. 5 will conclude and discuss our future plans.

2 Dynamic Meta Modeling

We have argued in Sect. 1 that DMM specifications are formal, but also easily understandable. This is an advantage to many other formalisms, which can only be used by experts of that formal-

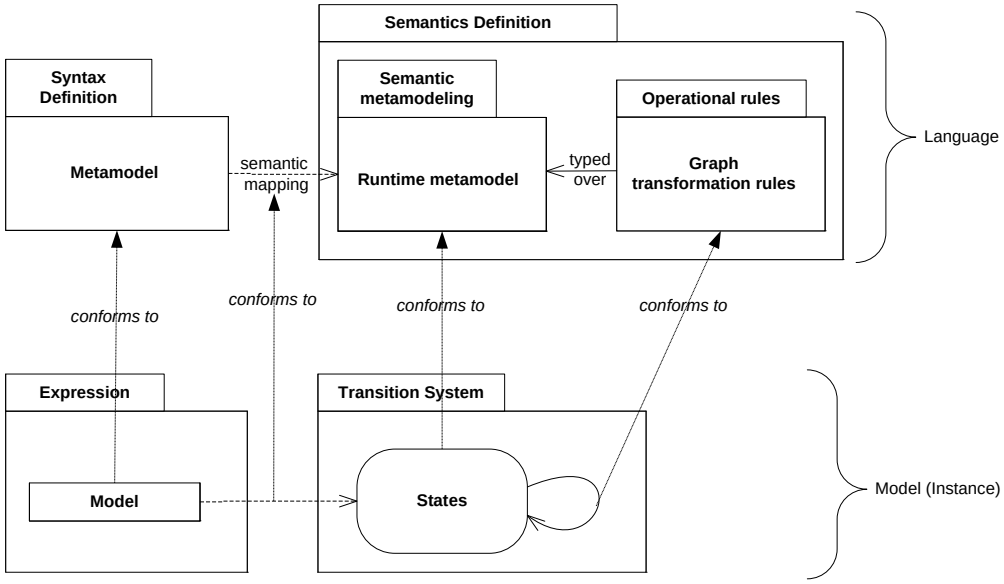

Figure 2: Overview of the DMM approach

ism. For instance, the π-calculus [MPW92] is a powerful formalism for semantics specification, but the average language user can not be expected to understand a π-calculus specification, let alone use it to specify the semantics of a language.

DMM aims at delivering semantics specifications which indeed can be understood by such users. It does that by providing a visual language for semantics specification. Additionally, a DMM specification is based on the metamodel of the according language, allowing users who are familiar with that metamodel to easily read a DMM specification.

In a nutshell, a DMM specification is created by first extending the language's metamodel with concepts needed to express states of execution; the enhanced metamodel is called *runtime metamodel*. Then, the behavior is defined by creating operational rules which modify instances of that runtime metamodel. An overview of DMM is provided as Fig. 2.

Since our goal will be to enhance the language of UML Activities, let us investigate the language's semantics specification as an example: the metamodel provided by the OMG [Obj09] only contains syntactic information, i.e., it describes the set of valid UML Activities. The language's dynamic semantics is specified using natural language: for instance, the UML specification document states that "the semantics of Activities is based on token flow". However, the language's metamodel does not contain the concept of token.

Therefore, the runtime metamodel adds that concept: A class `Token` is introduced such that instances of that class are associated to the language elements they are located at (e.g., `Actions`). As a consequence, an instance of the runtime metamodel describes a state of execution of an Activity by having `Token` objects sitting at particular elements. Figure 3 shows an excerpt of the runtime metamodel for UML Activities. The runtime class representing the concept of tokens is depicted in bold. Its `location` associations to the `ActivityNode` and `ActivityEdge` classes allow to model a concrete state of execution of an Activity: If a token

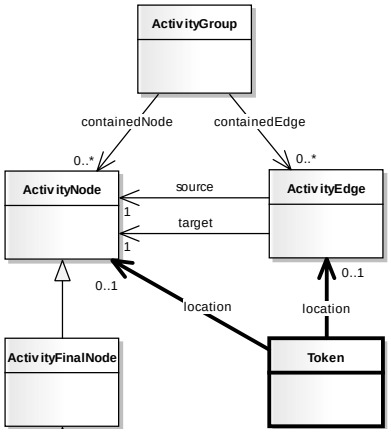

Figure 3: DMM runtime metamodel for UML Activities.

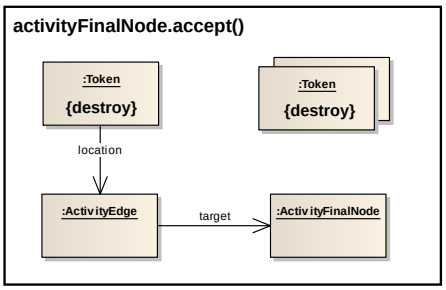

Figure 4: The original DMM rule describing the `ActivityFinalNode`'s semantics.

is sitting on a particular `Action`, the model contains a `location` link from the token to the object representing that `Action`.

Now, the operational rules come into play; a DMM rule is depicted in Fig. 4. Its semantics is as follows: The rule can be applied if an incoming `ActivityEdge` of an `ActivityFinalNode` carries at least one token. If this is the case, the rule is applied: all tokens flowing through the Activity are deleted (no matter where they are located), bringing the execution of the whole Activity to an immediate end.

The underlying formalism of DMM are *graph transformations*. Using the GROOVE toolset [Ren04], DMM specifications give rise to transition systems which describe the complete behavior of the according models. The start state of such a transition system is a model (in our case, an instance of the runtime metamodel, where e.g. `InitialNodes` are already equipped with a token). Now, every rule of the DMM specification is checked for applicability; if a rule can be applied, the application will lead to a new (and different) state (where e.g. the location of tokens has changed); the resulting transition is labeled with the applied rule. For every newly derived state, the process starts over again until no new states are found.[1]

[1] DMM specifications might give rise to an infinite transition system; in this case, standard techniques from model checking such as *bounded model checking* can be applied.

A transition system computed in that way can then be analyzed using model checking techniques. The properties to be verified need to be formulated over the applications of rules. For instance, if we want to know if a certain `Action` can ever be executed, we need to check if the transition system contains a transition which is labeled with the rule corresponding to the `Action`'s execution.

More concretely, as there is only one generic, parameterized rule defining the semantics of Actions, the rule's parameter has to be the name of this Action. For example, if we want to know whether the Action with name "A" is ever executed, we have to check whether the transition system representing the model's behavior contains a transition labeled action.start("A").

Now that we have gotten a precise idea of DMM, we are ready to perform our language enhancement in the next section.

3 Test-driven Language Derivation

In the last section, we have seen how DMM semantics specifications work in general, and we have investigated a small part of a semantics specification for UML Activities. Our next step will be to enhance the Activity language by adding a language element, and to define that element's semantics by means of an additional DMM rule in Sect. 3.1. We will discuss possible problems caused by that language modification.

To cope with such problems, we will then introduce the reader to our approach of *test-driven semantics specification* [SE09] in Sect. 3.2. Based on that, we will show in Sect. 3.3 how to reuse the concepts of that approach when deriving a new language from an existing one: In Sect. 3.4, we will fix the semantics specification (partly) by using rule overriding, and we will use test cases as guidance. Finally, Sect. 3.5 shows the tooling involved in the whole process.

3.1 Example: Enhancing UML Activities

UML Activities are a powerful behavioral language which can be used in all kinds of domains, from specifying low-level algorithms to defining high-level business processes. However, no language can contain all elements used within all kinds of contexts. As a result, the need for domain-specific languages arises.

For instance, consider the usage of UML Activities for business process modeling. The Activity language contains two language elements dedicated for the termination of (parts of) an Activity: The `ActivityFinalNode` terminates the execution of the complete Activity, whereas the `FlowFinalNode` can be used to terminate single flows of execution (e.g., in the case of concurrency).

However, the need might arise for more flexible ways to terminate flows of execution. As an example, we want to add a `GroupFinalNode` which—if it consumes a token—brings all execution within the node's `ActivityGroup` to an immediate end (but does not affect flows of execution outside the `ActivityGroup`).

The first step is to enhance the language's syntax, i.e., to add the `GroupFinalNode` to the language's metamodel. The integration of the new metaclass can be performed in different ways. Since the `GroupFinalNode`'s behavior is pretty similar to the behavior of the

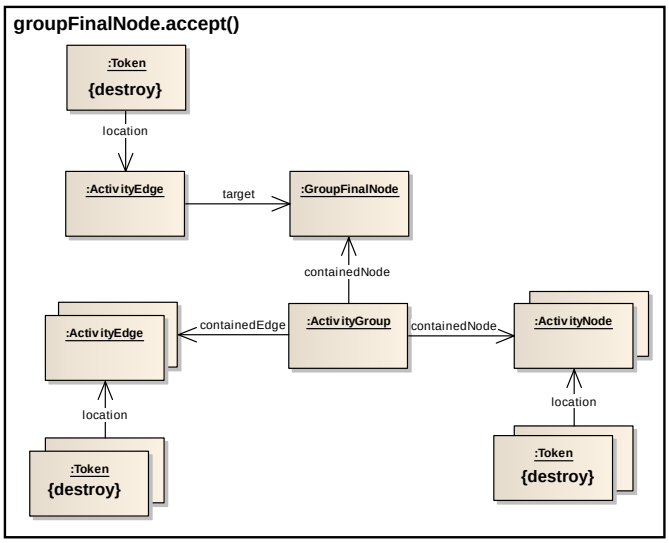

Figure 5: The modified DMM rule describing the `ActivityFinalNode`'s new behavior.

`ActivityFinalNode` (which consumes all tokens flowing in the Activity as soon as it receives a token; it has been depicted in Fig. 4 on page 5), and since the language engineer wants to reuse the concrete syntax of that node, he decides to add the element as a subclass of the `ActivityFinalNode`'s metaclass (which had been depicted in Fig. 3).

The language engineer also has a definition of the `GroupFinalNode`'s behavior in mind. An according DMM rule is depicted in Fig. 5. However, the performed language modification causes a couple of problems lying in the coordination of the old and new rules' behavior (since that behavior is heavily related). We will explain these problems in more detail below.

Now, we suggest to deal with such problems by performing modifications of semantics specifications in a *test-driven* way. In a nutshell, the language engineer will first create example models of the modified language. Such an example model contains one or more of the modified language elements in a certain context which should be related to the modifications which have been performed. For instance, the modification we have described above implies that the language engineer creates an example model which shows the new behavior of the `GroupFinalNode`.

Additionally, the language engineer will describe the expected behavior of that model in a precise, semi-formal way (more on that in the next section). Finally, executable test cases are generated from the example models and their behavior descriptions. The language engineer can now perform the modifications of the semantics specification against these test cases: If the tests all pass, he can take this as a sign that the modifications have been performed correctly. Otherwise, the failing test cases will hopefully point him to the problematic modifications he performed.

Since the idea to perform test-driven language derivation is based on our idea of test-driven semantics specification [SE09], we want to shed light on that approach in the next section.

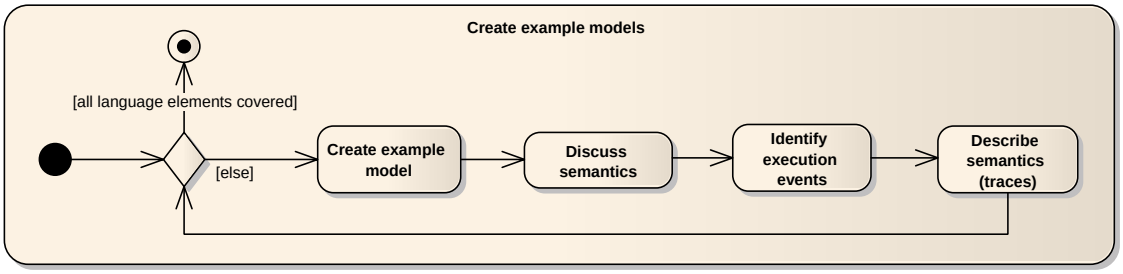

Figure 6: Create example models

3.2 Test-driven Semantics Specification

We have already seen in the introduction that a semantics specification follows the Input-Process-Output model in some sense: a model serves as input, and the semantics of that particular model is the output. In this section, we want to explain this idea in more detail.

"Test-driven semantics specification" means that the semantics of a language is developed against existing test cases: As soon as all test cases succeed, the semantics specification is finished (and we have some hope that it indeed has an appropriate quality). In our scenario of test-driven semantics specification, the input specified by a test case is an *example model*, i.e., a model which demonstrates certain behavioral properties of some language elements. Sect. 3.2.1 will deal with the creation of example models and the description of their expected behavior.

Section 3.2.2 will then show how to automatically transform each example model and its description of behavior into an executable test case. Finally, Sect. 3.2.3 will point out how to perform the actual semantics specification against the test cases.

3.2.1 Creating Example Models

The starting point is the abstract syntax of the language under consideration. It defines all language elements and their relations with each other. In the case of DMM, the abstract syntax must be given as a metamodel. Based on the abstract syntax, the example models should be created step by step, systematically going from the most basic to more complex language constructs[2].

Then, for each example model the expected behavior needs to be identified, and to be described in a semiformal way. This is done using so-called *traces of execution events*. An *execution event* in our sense is some interesting event happening during the execution of a model. For instance, in the example below (which is in the context of UML Activities), we will use execution events corresponding to the execution of a particular `Action`.

Such execution events can then be composed to *traces*. A trace is a sequence of execution events which occur when a particular model is being executed. It describes one possible way of executing a particular model. The process of creating example models is depicted in Fig. 6.

Let us illustrate the above with a simple example, which is depicted in Fig. 7. Its purpose is to demonstrate the semantics of the `DecisionNode` and `MergeNode`. This example is interesting because of the fact that it allows for more than one possible execution: a token flowing

[2] The example models can of course be created using the language's concrete syntax

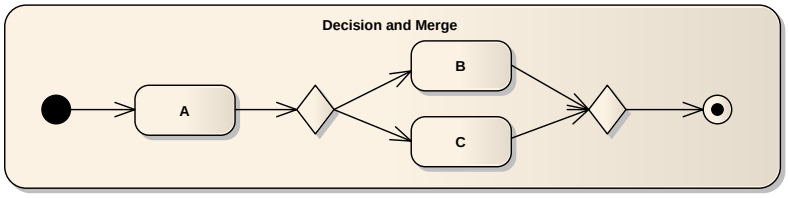

Figure 7: Example Activity containing a simple `DecisionNode`/`MergeNode` structure

through the Activity will—as soon as it has passed `Action` "A"—be routed either to `Action` "B" *or* to `Action` "C".

Obviously, the interesting execution events which occur when that model is executed are the executions of the contained `Actions`. As a result, we identify the event ActionExecutes(Name) which refers to the execution of an Action with name Name.

We have already seen above that due to the involved DecisionNode, there are two ways to execute the model. Therefore, we will describe the model's behavior by two traces of execution events:

ActionExecutes("A") ActionExecutes("B")

and

ActionExecutes("A") ActionExecutes("C")

We decided to reduce the semantics of Activities to the possible orders of execution of `Actions`, since the `Actions` are the places where the actual work will be performed. However, it would also be possible to use more fine-grained traces like InitialNode() ActionExecutes("A") DecisionNode() ActionExecutes("B") MergeNode() ActivityFinalNode().

With these traces, we have already finished the description of our example model's behavior. We can now turn to the transformation into an executable test case in the next section.

3.2.2 Deriving Test Cases

In this section, we want to investigate how to automatically verify that an example model indeed behaves as we expect it to. This is done in two steps: First, we formalize the traces of execution events of our example model by translating them into a notion of *temporal logic*. Second, we use a model checker to verify whether the transition system raising from the example model and our semantics specification contains exactly the expected behavior (and nothing else). To make our test cases executable, the described process is triggered by a small Java framework we have implemented on top of JUnit [GB].

Let us start with translating the traces of execution events into temporal logic. The idea of the translation is as follows: We want to express that the transition system contains a path starting from the start state such that all execution events occur on that path in the desired order, and that no other execution events occur in between.

We have seen in Sect. 2 that DMM specifications give rise to a transition system where each transition is labeled with the DMM rule creating that transition. As a result, we have to map

our execution events to the according DMM rules. Having done that, the translation is pretty straight-forward: From each execution event e, an expression $\mathbf{EF}(r_e)$ is generated, where r_e is the DMM rule corresponding to event e. Such an expression is true iff there **E**xists a path such that **F**inally, rule r_e occurs as a label of one of the transitions.

These expressions are then nested to express the sequence of events to occur: For instance, the sequence e_1e_2 is translated into the CTL formula $\mathbf{EF}(r_{e_1} \wedge \mathbf{EF}(r_{e_2}))$, expressing that there must be some occurence of r_{e_1}, and from that point on, there must be an occurence of r_{e_2}.

The fact that there must not be any events in between e_1 and e_2 is represented by using a CTL **U**ntil expression which makes sure that no unexpected rules occur until the next desired rule occurs. We do not show this construction here; the interested reader is pointed to [SE09].

Having computed our CTL formulas $f_1, \ldots, f_n$ (one for each trace of execution of the example model's behavior), we can check whether all these traces are indeed contained in the resulting transition system. This is done by using a model checker to verify whether the formulas hold on our transition system; if this is the case, the behavior is contained as expected.

Finally, we need to make sure that the transition system *only* contains the expected behavior. This is verified by a final CTL formula which reads as follows: $\mathbf{AF}(f_1 \vee \cdots \vee f_n)$. It makes sure that on **A**ll paths, one of the expected sequences of events takes places.

If all CTL formulas as described above hold for our transition system, we can be sure that for our example model, the semantics specification produces exactly the desired behavior. If the model checker finds out that one of the expressions does not hold, the resulting counter example will be helpful when fixing the errors of our specification.

3.2.3 Specifying the Semantics

The actual semantics specification can then be performed against the test cases we got from the last step. We start with specifying the semantics of the most simple language elements of our language. As soon as our specification contains the semantics description of all elements contained in one of our example models, that model and its description of behavior are transformed into an executable test case, which can then be verified against the current state of the semantics specification as described in the last section.

Now, if the derived test case succeeds, we can continue with specifying the more complex elements' semantics, until finally all language elements are covered. Otherwise, we need to fix the specification until the test case succeeds.

Note that all test cases are executed within every iteration of the process described above; this is to prevent regression errors (i.e., destroying the behavior of an already specified element by specifying the semantics of a still unspecified element). The whole process is depicted in Fig. 8. Now that we have seen how to create a semantics specification in a test-driven way, we can transfer the concepts used into our scenario of deriving a new language from an existing one.

3.3 Test-driven Derivation Process

Recall the language modifications we have in mind: we want to add a language element `GroupFinalNode` with the purpose of terminating the execution of a particular `ActivityGroup`. To reach this goal, we have added the according metaclass to the metamodel by subclassing an

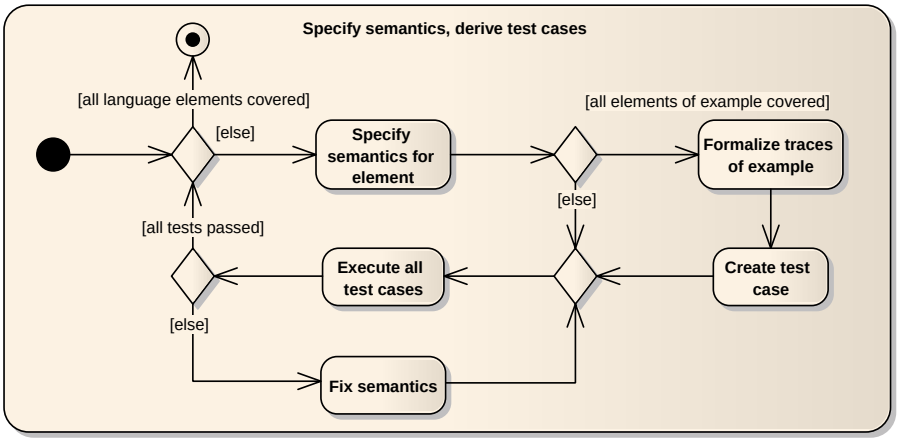

Figure 8: Specify semantics, create test cases from example models

existing one.

Now, in a test-driven setting, our next step consists of defining a test case against which we can then specify the language element's behavior, i.e., an example model.[3] How does such an example model for our language extension look like?

There is one major requirement: The example model needs to demonstrate the behavior of interest. In our case, this means that we need a UML Activity containing our `GroupFinalNode`, and the Activity's structure should be such that the `GroupFinalNode`'s existence indeed has an impact.

Despite that, the example model should be as simple as possible. This has one major advantage: In case the test case derived from our example model fails at a later stage, it will be easier to figure out the cause of the failure, since less language elements are involved.

Figure 9 shows an example model which suits our needs. Obviously, it is very simple. Additionally, it demonstrates the behavior of our new language element. To explain this, assume for the moment that the Activity does not contain the `ActivityGroup`, and that the `GroupFinalNode` is a simple `ActivityFinalNode` as the one above. Then, the semantics would be as follows: since the whole Activity's execution is terminated as soon as a token arrives at one of the `ActivityFinalNodes`, a possible behavior would be that just the execution of A (or B) takes place. In this case, one of the tokens has flown all the way down to the upper (lower) `ActivityFinalNode` before the execution of B (A) has even started (i.e., the other token is still sitting on the `ActivityEdge` in front of that `Action`). The situation is different in the case of our real example model containing the `ActivityGroup` and the `GroupFinalNode`: Since the group "encapsulates" the effect of the `GroupFinalNode`, it will always be the case that A is executed (however, B might not be executed as discussed above). We will appreciate this fact by creating our traces of execution events accordingly: There will be no trace where A is not executed.

[3] Of course, there should usually be more than one example model. For instance, in our case the example model demonstrates that the `GroupFinalNode` deletes a token within its `ActivityGroup` only, but does it really destroy *all* tokens within that group?

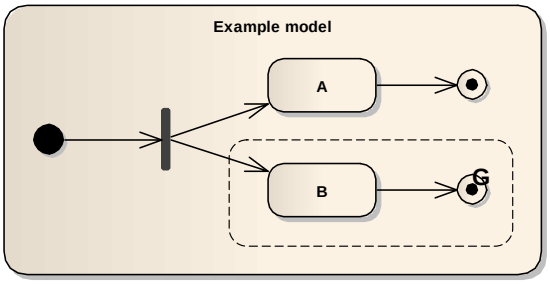

Figure 9: Example model demonstrating the behavior of the `GroupFinalNode`

Now, being equipped with an example model (or a set of example models) and its expected behavior, we can continue with specifying the according behavior. This is done by adding the rule we have seen as Fig. 5 to the DMM ruleset.

We are now ready to execute our test case. As it turns out, the test fails. This is due to an error we made: We did not take into account that the left-hand's graph of the rule activityFinalNode.accept() (see Fig. 4 on page 4) basically[4] is a subgraph of the new rule's left-hand graph. As a result, that rule matches whenever the new rule matches. The consequence is that the transition system resulting from the example model still contains traces where *A* is not executed.

This is a problem as described earlier: In our situation, it does not suffice to just add the DMM rule describing the new element's behavior. In addition, the modeler has to make sure that the old behavior does not take place in the context of the new language element `GroupFinalNode`. The next section will show how to cope with this problem using DMM rule overriding.

3.4 Using DMM Rule Overriding

We have seen above that the enhancement of our semantics specification is not finished yet: We need to prevent the `ActivityFinalNode`'s behavior from being applied in the context of the new element `GroupFinalNode`. One way to do this would be to change rule activityFinalNode.accept() such that it only matches if the `ActivityFinalNode` is not contained in an `ActivityGroup` (by adding an according negative application condition to that rule). This would indeed fix our failing test, since the `ActivityFinalNode`'s behavior would not take place any more in that situation.

Fortunately, the language's semantics has been developed in a test-driven way. In this case, our already existing test cases will hopefully tell us whether we have broken any existing behavior with our changes. And this is indeed the case: Obviously, rule activityFinalNode.accept() does not match any more in case its `ActivityFinalNode` belongs to an `ActivityGroup` (this is exactly the change we have performed above). However, the semantics of the `ActivityFinalNode` stays the same, no matter whether it belongs to an `ActivityGroup` or not. In other words: An `ActivityFinalNode` which does belong to an `ActivityGroup` shall still delete all tokens flowing through the Activity, no matter where they are located. This does not happen, since rule activityFinalNode.accept() does not match any more in such a sit-

[4] The only difference is the typing of the FinalNodes, but that doesn't affect the matching here.

uation. As a consequence, every test case in which an `ActivityFinalNode` belongs to an `ActivityGroup` will now fail, pointing us to the fact that our language modification has had some side effects.

However, the problem is easily fixable using a more sophisticated DMM construct: Rule overriding [EFS09]. Before we start to explain that construct, let us first look into DMM rules more deeply.

Every DMM rule has a so-called *context node*, and a rule is defined in the context of that node. Since every node in a DMM rule has a type, the context node implies a type for the rule itself: the context node's type can be seen to *own* the behavior described by the rule (just as a method is owned by the class it is defined in). Note that this concept strengthens the similarity of DMM and object-oriented programming languages and therefore increases the understandability of DMM specifications.

In the rule activityFinalNode.accept(), the context node is the node typed `ActivityFinal-Node`; as expected, the context node of the new rule groupFinalNode.accept() is the node typed `GroupFinalNode`. As a result, the new rule has two interesting properties: first, as we have seen above, the rule's left-hand graph is a subgraph of the left-hand graph of rule activityFinalNode.accept() (this caused our problem at the beginning), and second, the context node's type of rule groupFinalNode.accept() is a subtype of the context node's type of rule activityFinalNode.accept().

The described situation is exactly the prerequisite for using DMM rule overriding. The idea is as follows: Given two rules r_1, r_2 such that the two properties mentioned above hold, rule r_2 can *override* rule r_1 (in our example, activityFinalNode.accept() would be r_1, and groupFinalNode.accept() would be r_2).

If a rule is overridden, its matching is affected: An overridden rule r_1 only matches a host graph if there is a morphism from the rule's left-hand graph into the host graph *and* if there is no overriding rule r_2.[5] It is easy to see that this indeed fixes our problem from above: Letting groupFinalNode.accept() override rule activityFinalNode.accept() prevents the former rule from matching in the context of our new type `GroupFinalNode`, therefore leading to the desired behavior.

3.5 The Tool Chain

It remains to shed some light on the tools involved in the process of modifying an existing language (which are depicted in Fig. 10). In Sect. 2, we have already mentioned that the Groove toolset [Ren04] is used to compute the transition system used for analysis of a model's behavior. However, DMM supports quite sophisticated language features which are not directly supported by the notion of graph transformation rules Groove supports. For instance, DMM rules can explicitly invoke other DMM rules, which is not supported by common graph transformation tools.

Therefore, an own visual language for DMM specifications has been developed using Eclipse-based frameworks such as EMF [SBPM08], GMF [Ecl09a], and UML2 [Ecl09b] – part of the latter project is an EMF implementation of the UML metamodel. As expected, the syntax of

[5] Note that in [EFS09], we have in fact defined two notions of rule overriding; in [EFS09], the notion used within this paper is called *complete overriding*.

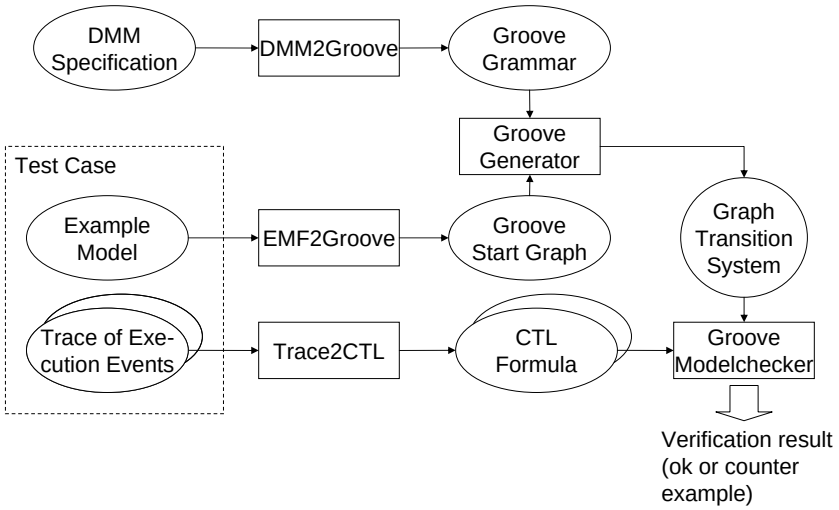

Figure 10: The DMM tool chain

the DMM language is defined by means of a metamodel. The DMM component DMM2Groove is responsible for translating an instance of the DMM metamodel into a valid Groove grammar ready to be executed on a proper graph state.

Additionally, the DMM tooling is capable of handling arbitrary EMF models. The according DMM component EMF2Groove takes such an EMF model (e.g., a concrete UML Activity as instance of the UML2 metamodel mentioned above) and transforms it into a Groove graph. In the next step, both the Groove grammar and the start graph are fed into the Groove Generator, which is then used to compute a transition system representing the complete model's behavior.

We have seen earlier in this section that a DMM test case not only consists of an example model, but also of a set of traces of execution events, each describing one possible execution of the example model. The DMM component Trace2CTL is responsible for generating CTL formulas from these traces (please refer to [SE09] for the details of the generation).

Finally, the transition system and the generated CTL formulas serve as input for the Groove model checker, which checks whether the formulas hold for the transition system (and therefore for the model from which the start graph had been generated). The model checker will either report that a formula holds, or it will provide a counter example showing under which circumstances the CTL property is violated; that counter example can then be used to understand why the CTL property is violated, and should therefore be very helpful when fixing the semantics specification.

4 Related Work

The work most closely related to ours probably is the work by Sadilek et al. [Sad08]. His goal is to quickly prototype DSLs. The comparable scenario is as follows: A language's semantics might first be specified using a formal language, e.g. *Abstract State Machines*, for the sake of proving properties of the DSL's semantics. Later on, a second, more efficient semantics spec-

ification might be created which shall be *semantically equivalent* to the first one. Since both semantics specifications allow for DSL instances to be executed, the language engineer can now create test models of the DSL, execute them and compare the resulting execution traces. The main difference to our approach is that Sadilek uses tests to compare two semantics specifications, whereas we use them to convince ourselves that the semantics specification indeed produces the behavior the language engineer had in mind.

In the area of language engineering, several approaches for defining DSLs exist. For instance, MetaCase provides MetaEdit [SLTM91], Microsoft provides the DSL Tools as part of MS Visual Studio [CJKW07], and the Eclipse foundation provides the Graphical Modeling Framework [Ecl09a]; all these approaches aim at an easy creation of visual languages. openArchitectureWare [HVEK07] provides a set of tools which allow for the easy creation of textual languages, including powerful editor support.

To our knowledge, all the above approaches focus on defining DSLs from scratch. Additionally, they do not focus at all on the definition of behavioral semantics: the approaches provide support for code generation, but they do not provide a means to systematically create high-quality code generators; the generation is pretty much done ad-hoc.

Quite some work exists on typed graph transformations, on which we defined our notion of rule overriding. For instance, in [LBE+07], de Lara et al. show how to integrate attributed graph transformations with node type inheritance, therefore allowing for the formulation of *abstract* graph transformation rules (i.e., rules which contain abstract nodes). The resulting specifications tend to be more compact, since a rule containing abstract nodes might replace several rules which would otherwise have to be defined for each of the concrete subtypes. The resulting formalism does not provide support for refinement of rules (and is therefore comparable with the expressiveness of the current state of DMM).

In [TR05], Taentzer et al. show how to formulate structural properties of type graphs with inheritance using graph constraints, and they provide a translation into standard graphs. In contrast to our work, they concentrate on structure, whereas our approach modifies the behavior of rules participating in an *overrides* relation.

5 Conclusion

In this paper, we have shown how test-driven approaches from software engineering can be reused in the field of language derivation. For this, we have first introduced our semantics specification technique Dynamic Meta Modeling, and we have explained how graph transformations serve as the backing formalism of DMM. Based on that, we have introduced a simple example of language enhancement in Sect. 3, and we have discussed the problem of overriding existing behavior.

We have then shown how to perform language derivation in a test-driven way, following the approach of test-driven semantics specification. We have also shown how rule overriding can be used to override already existing DMM graph transformation rules, and we have demonstrated all that by fixing the flaws we (intentionally) introduced within our language modification example.

Overall, we have presented an example of applying the well-established formalism of graph transformations [EEKR99] in a typical software engineering scenario. Currently, we are in-

vestigating different notions from software engineering for the sake of using them within our test-driven semantics specification approach, the most important one being *test coverage criteria*: for instance, a minimum such criterion is that over all test cases, every DMM rule has been applied at least once, but more complex coverage criteria are worth investigating.

Additionally, our focus is on developing tool support for test-driven language engineering. Our tool support has the following goals:

- Easy specification and execution of test cases.
- Back-propagation of the model checker's counter example in case a test case failed.
- Visual debugging and execution of test cases.

Finally, two of our students are working on complex semantics specifications for behavioral diagrams of the UML, using test-driven semantics specification. In the course of their work, they will also have to modify the DMM specification of an existing language. While performing their work, the students will make use of the existing DMM tooling; we plan to learn from their efforts about further needed tooling.

Bibliography

[Bec02] K. Beck. *Test-Driven Development by Example*. Addison-Wesley Longman, Amsterdam, The Netherlands, 2002.

[CJKW07] S. Cook, G. Jones, S. Kent, A. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[Ecl09a] Eclipse Foundation. Graphical Modeling Framework. http://www.eclipse.org/modeling/gmf/, 2009. online, accessed 5-5-2009.

[Ecl09b] Eclipse Foundation. UML2 Project. http://www.eclipse.org/uml2/, 2009. online, accessed 5-5-2009.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages, and Tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[EFS09] G. Engels, D. Fisseler, C. Soltenborn. Improving Reusability of Dynamic Meta Modeling Specifications with Rule Overriding. In R. DeLine (ed.), *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2009), Corvallis, Oregon (USA)*. Pp. 39–46. IEEE Computer Society, Piscataway, NJ (USA), 2009.

[EHS99] G. Engels, R. Heckel, S. Sauer. Dynamic Meta Modelling: A Graphical Approach to Operational Semantics. In *Proceedings of the workshop on Rigorous Modeling*

and Analysis with the UML: Challenges and Limitations (satellite event of the Conference on Onject-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999)), Denver, CO (USA). 1999.

[ESW07] G. Engels, C. Soltenborn, H. Wehrheim. Analysis of UML Activities using Dynamic Meta Modeling. In Bosangue and Johnsen (eds.), *Proceedings of the FMOODS 2007 Conference*. LNCS 4468, pp. 76–90. Springer, 2007.

[GB] E. Gamma, K. Beck. JUnit Homepage. http://www.junit.org/. online, accessed 1-2-2010.

[Hau05] J. H. Hausmann. *Dynamic Meta Modeling*. PhD thesis, University of Paderborn, 2005.

[HVEK07] A. Haase, M. Völter, S. Efftinge, B. Kolb. Introduction to openArchitectureWare 4.1.2. MDD Tool Implementers Forum (Part of the TOOLS 2007 conference, Zürich), 2007.

[LBE+07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Attributed Graph Transformation with Node Type Inheritance. *Theoretical Computer Science* 376(3):139–163, 2007.

[MPW92] R. Milner, J. Parrow, D. Walker. A Calculus of Mobile Processes, I. *Information and Computation* 100(1):1–40, 1992.

[Obj09] Object Management Group. OMG Unified Modeling Language (OMG UML) – Superstructure, Version 2.2. http://www.omg.org/docs/formal/09-02-02.pdf, 2 2009.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *AGTIVE 2003 – Revised Selected and Invited Papers*. LNCS 3062, pp. 479–485. Springer, 2004.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[Sad08] D. A. Sadilek. Prototyping Domain-Specific Language Semantics. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented Programming Systems Languages and Applications*. ACM, New York, 2008.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework, Second Edition*. Addison-Wesley Professional, 2008.

[SE09] C. Soltenborn, G. Engels. Towards Test-Driven Semantics Specification. In A. Schürr (ed.), *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Denver, Colorado (USA)*. Pp. 378–392. Springer, Berlin/Heidelberg, 2009.

[SLTM91] K. Smolander, K. Lyytinen, V.-P. Tahvanainen, P. Marttiin. MetaEdit: a Flexible Graphical Environment for Methodology Modelling. In *Proceedings of the third international conference on Advanced information systems engineering (CAiSE 91)*. Pp. 168–193. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[TR05] G. Taentzer, A. Rensink. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In Cerioli (ed.), *FASE*. LNCS 3442, pp. 64–79. Springer, 2005.