From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations

Gregor Engels[1], Anneke Kleppe[2], Arend Rensink[2], Maria Semenyak[1], Christian Soltenborn[1], and Heike Wehrheim[1]

[1] University of Paderborn, Department of Computer Science, 33098 Paderborn, Germany
{engels,semenyak,christian,wehrheim}@upb.de
[2] University of Twente, Department of Computer Science, 7500 AE Enschede, The Netherlands*
rensink@cs.utwente.nl

Abstract. Model transformations support a model-driven design by providing an automatic translation of abstract models into more concrete ones, and eventually program code. Crucial to a successful application of model transformations is their *correctness*, in the sense that the meaning (semantics) of the models is preserved. This is especially important if the models not only describe the structure but also the intended *behaviour* of the systems. Reasoning about and showing correctness is, however, often impossible as the source and target models typically lack a precise definition of their semantics.

In this paper, we take a first step towards provably correct behavioural model transformations. In particular, we develop transformations from UML Activities (which are visual models) to programs in TAAL, which is a textual Java-like programming language. Both languages come equipped with formal behavioural semantics, which, moreover, have the same semantic domain. This sets the stage for showing correctness, which in this case comes down to showing that the behaviour of every (well-formed) UML Activity coincides with that of the corresponding TAAL program, in a well-defined sense.

1 Introduction

The concept of model-driven development (MDD) crucially depends on the possibility of generating lower-level models (and finally code) from abstract models. Originally meant as a help for structuring complex programs, models today take on a different, and much more central, role: they not only act as the primary entity for discussions with customers, but also within the development trajectory, for fixing interfaces with other systems or analysing the system with respect to requirements. Thus, it is vital to ensure that the actual system really adheres to the models. The MDD way of ensuring this is by directly generating the code from the models, possibly through intermediate steps where abstract models are refined into more concrete ones. However, this process, called model transformation, is a real solution only by virtue of the *correctness* of the

* The research by this group was financed by the GRASLAND project, funded by the Dutch NWO (project number 612.063.408).

individual transformations, in the sense that they themselves do not change the meaning (usually called the *semantics*) of the models in unintended ways. Showing that transformations are semantics-preserving is the core problem addressed by this paper. In fact, we concentrate on *behavioural* semantics, which is concerned with what the system actually does (in contrast to, for instance, structural semantics, which is concerned with the system architecture).

The problem is aggravated by the fact that model transformations usually go between different meta-models. Quite often, the abstract source model is developed using a *visual* modelling language (e.g. the UML), whereas the target model is *textual* (e.g., a program).

A lot of research has been devoted to finding appropriate languages for describing model transformations in the first place [1,2], a quest that has recently resulted in the QVT language proposed by the OMG group (see [3]). In this context, various notions of "correctness" have already been studied. Correctness can for instance refer to the *syntactical* correctness of the generation algorithms (e.g. of transformation rules in a rule-based setting [4]), it can be *termination* of the generation [5] or the uniqueness of the generated model (*confluence* of rules) [6]. Behaviour preservation, addressed here, is different from all these — in fact it presupposes that the transformations are already correct in the above senses — and is particularly challenging. An area where behaviour preservation has received some interest is *refactoring*, a specific kind of model transformation improving the structure of models [7,8]. Contrary to our interest here, transformations during refactorings do not operate on different meta-models but stay within one language.

Showing behaviour preservation of model transformations between different meta-models first of all requires a *formal* definition of the behavioural semantics of source and target model. Moreover, the semantic domains should be the same, to avoid yet another transformation on semantic domains. Given a formal semantics, a comparison of the behaviour of source and target model is a matter of selecting an appropriate notion of equivalence over the semantic domain.

In this paper, we show that this ideal of showing behavioural correctness of model transformations is indeed attainable. As an example, we define a transformation from UML Activities to TAAL [9] programs. An overview of the approach is depicted in Fig. 1. UML Activities are used to model the orderings of actions within, for instance, business processes (see [10]). They are frequently employed in workflow modelling and constitute a very high level, visual description of workflows. They are defined as a subset of UML, which we will denote UMLA in the sequel. On the other hand, TAAL is a simple (Java-like) object-oriented programming language, featuring class definitions, object instantiation and concurrency. The transformation thus has to bridge the gap between a *visual* model on the one side and *program code* on the other side. We achieve this by defining the model transformation on the (MOF-compliant) abstract syntax meta-models of the two languages (MT in Fig. 1). The model transformation is thus a transformation of one graph into another, and consequently we employ graph transformation rules for their definition. This gives us a model transformation that is both formally defined and executable, employing the graph-transformation tool Groove [11] for rule execution.

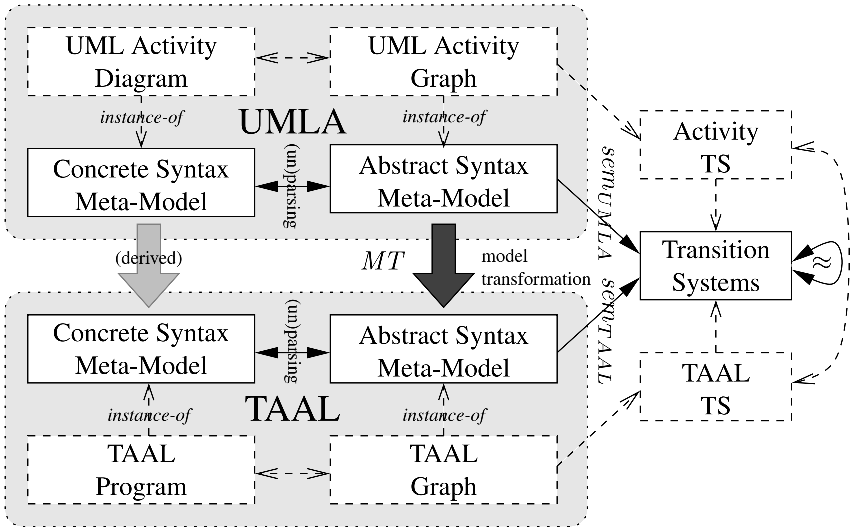


Fig. 1. Overall approach of behaviour-preserving model transformation

This choice of example model transformation has the great advantage that both languages (UMLA and TAAL) are already equipped with a formal semantics, which, moreover, is defined on the same basis, viz. (again) through graph transformation. On the UMLA side, we use a semantics defined with *Dynamic Meta Modelling* [12]; for TAAL, the semantics were developed together with the language [9]. These formal semantics (sem_{UMLA} and sem_{TAAL} in Fig. 1) first add run-time specific structure to the meta-models (e.g. a program counter representation on the TAAL side) and then define the behaviour as the possible changes in instances of that enhanced meta-model. Again, meta-models being graphs leads to a graph-rule based definition of the semantics, and we also use Groove to automatically derive the semantics of both UML Activities and TAAL programs. The underlying common semantic domain are *transition systems* ($\mathcal{TS}$), in which transitions represent applications of graph rules, in particular also those corresponding to executions of *actions* (in the UML Activity) or *operations* (in the TAAL program). Our semantics thus generates a transition system out of a meta-model instance of a UML Activity (TS_{Act}) and TAAL program (TS_{TAAL}). On these transition systems we can compare the execution behaviour of UML Activity and generated TAAL program, and can show that the ordering of actions in the Activity coincides with the ordering of corresponding methods (with the same name) in the TAAL program.

Similar approaches to evaluating the correctness of model transformations have been presented in e.g. [13], where different variants of Statecharts are transformed into each other, and a bisimilarity check is carried out (on particular instances). In a sense, our technique also resembles certification techniques for compilers [14,15], where one particular instance of compilation is afterwards checked for correctness using a generated certificate. Nevertheless, our ultimate aim is a general proof of correctness of the *transformation*. Once this task is done, we do not have to check the behaviour of transformation results any more. This paper presents the first steps in this direction, giving the model transformation itself, its tool-support, the two semantics and a comparison of the behaviour, viz. semantics, on examples.

In Section 2, we start with a short introduction to UML Activities and TAAL, and we define the graph-based transformations as a set of transformation rules over the meta-models of the two languages. In Section 3 we then argue that the semantics of Activity and generated TAAL program coincide with respect to *trace equivalence*, when comparing the execution traces of the UML model and the object-oriented program. Section 4 describes the tool support for transformation and semantics generation.

2 Transformation

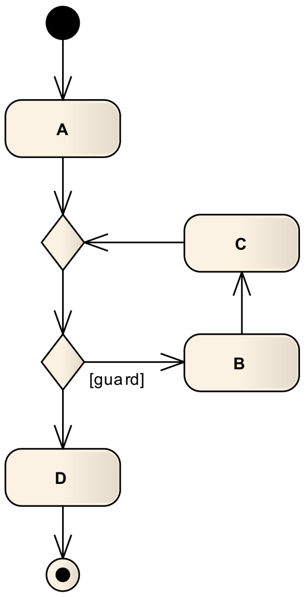
This section presents the model transformation from UML Activities to TAAL. We start by discussing the general idea of the transformation and give a first example.

UML Activities are an expressive tool for expressing the order of execution of so-called `Actions`, i.e., atomic behavioural units. The ordering is specified by a directed graph with different sort of nodes: `Actions` themselves form nodes, the start and end of an execution is marked with a special `InitialNode` and `FinalNode`, and `MergeNodes` and `DecisionNodes` regulate the flow of control. Figure 2 shows an example Activity (plus its corresponding TAAL program). The semantics of the Activity is as follows: The first `Action` to be executed is A, indicated by the fact that the `InitialNode` (the filled circle) points to it. The A `Action` is followed by a `MergeNode` and a `DecisionNode`; if the guarding condition is true, `Actions` B and C will be executed, otherwise D is executed, and the Activity ends (indicated by the `FinalNode`).

Due to the `Merge-` and `DecisionNodes`, UML Activities allow for an unstructured flow of control which is hard to translate into a structured programming language without GOTO statements. Therefore, we restrict our model transformation to *well-formed* Activities which have a structured control flow.

Well-formedness is inductively defined (similar approaches to well-formedness can be found in [16]). For this, we introduced the concept of *building blocks*. Every building block has exactly one incoming and one outgoing edge connecting it to the rest of the Activity.

- An `Action` itself constitutes a building block (see Fig. 3a).
- A sequence of two building blocks, connected by an `ActivityEdge`, is a building block (see Fig. 3b).
- A `DecisionNode`, followed by two building blocks and a closing `MergeNode`, is a building block (see Fig. 3c). Note that one of the outgoing `ActivityEdges` of the `DecisionNode` must be equipped with a guard (i.e., a `ValueSpecification`).
- A `MergeNode` followed by a `DecisionNode` and a building block which is itself connected to the `MergeNode` is a building block (see Fig. 3d). Here, the `DecisionNode` has an additional outgoing `ActivityEdge` which is taken if the guarding condition is false.



```
program ActivityExecution
{
    new ActivityExecutionClass().main()
}

   class ActivityExecutionClass
        main() {
            A();
            while [guard] do
                B();
                C();
            endwhile
            D();
        }
        A() {}
        B() {}
        C() {}
        D() {}
    endclass

endprogram
```

Fig. 2. A well-formed UML Activity and the corresponding TAAL program

- A ForkNode, followed by two building blocks and a closing JoinNode, is a building block (see Fig. 3e).
- Finally, an InitialNode followed by a building block followed by a FinalNode forms a well-formed Activity (see Fig. 3f).

Such well-formed Activities are the starting point for our transformation, which follows the inductive definition of well-formedness:

- Actions are mapped to TAAL operations.
- A sequence of two Actions is mapped to a sequential execution of the corresponding operations.
- A DecisionNode and its MergeNode are mapped to an if-then-else expression.
- A MergeNode followed by a DecisionNode is mapped to a while-do expression.
- A ForkNode followed by a JoinNode is mapped to a forking of methods (i.e., parallel execution of the parts in between the nodes).

The generated code is then embedded into a TAAL program skeleton, i.e., a main() method which is owned by a class ActivityExecutionClass. This class is instantiated, and the main() method is invoked. In the right of Fig. 2, we see the TAAL program corresponding to the UML Activity on the left. The MergeNode-DecisionNode structure of the Activity is translated into a TAAL while loop.

Now that we have given the general idea of our transformation, we want to look into the details of the transformation's realization. Before we can do so, we need to provide

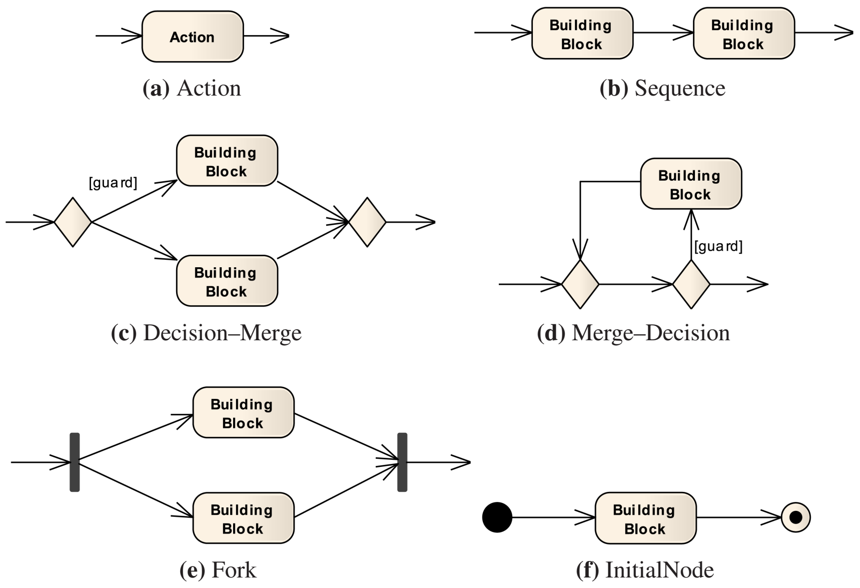


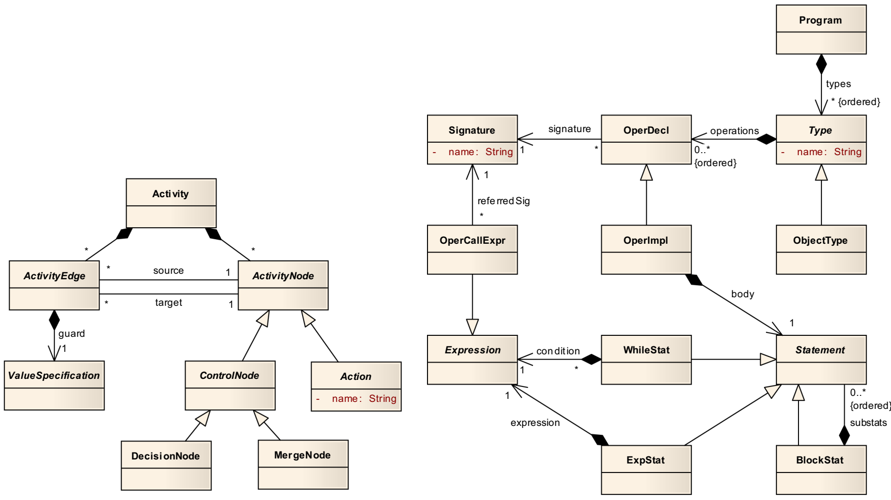
Fig. 3. Different building blocks (a–e) and one well-formed Activity (f)

a couple of prerequisites. As we will see below, the transformation is defined on the abstract syntax level, i.e. the meta-models.

Meta-models. The abstract syntax of the UML is defined by means of a *meta-model*, i.e., a set of class diagrams describing the structure of valid diagram instances. Figure 4a shows the Activity concepts relevant for this paper. The class diagram basically looks as expected: An `Activity` consists of a number of `ActivityNodes` which are connected by `ActivityEdges`. `Actions` are atomic units of behaviour, and `ControlNodes` are used to introduce decisions etc. into the modelled flow of execution.

The abstract syntax of the TAAL language is more complex. A TAAL `Program` consists of a number of `Types`, one of which is the `ObjectType` (representing the concept of a class). A `Type` owns a number of operations, which have a `Signature` and a `Statement` representing the body of the operation. There are a number of `Statements`, including a `WhileStat`, an `ExprStat` and a `BlockStat` used as a container for an arbitrary number of `Statements`. An operation call is represented by the `OperCallExp` expression. Figure 4b shows the most important concepts of the TAAL language. Note that for the sake of simplicity, we have omitted a huge number of concepts (including everything related to variables, literals etc.).

Transformation. As we have seen, the abstract syntax of both languages is described by means of meta-models, i.e., class diagrams. Therefore, a valid instance of a UML Activity or a TAAL program can be described as an object diagram which is consistent to the according class diagram. Since object diagrams can be treated as (labelled) graphs, we decided to use *graph transformation rules* (GTRs, [17]) for the specification



(**a**) UML Activity (**b**) TAAL

Fig. 4. Excerpt of the meta-models

of our transformation. This approach is a common one for defining model transformations [18], and has—in particular for our undertaking—a number of advantages: First of all, GTRs are specified completely formally; this is important for our final goal of *proving* that our transformation is behaviour preserving. Second, due to their visual appearance, GTRs are relatively easy to understand, and third, the semantics of Activities as well as TAAL programs is specified with GTRs, which will allow us to work with the same formalism for finally proving the correctness of our transformation. Moreover, due to the availability of GTR tools, our transformation is executable.

GTRs performs changes on a so-called *host graph*. They consist of a left-hand and a right-hand graph; if a subgraph similar to the left-hand graph can be found in the host graph, it is replaced by the right-hand graph. In our case, the start graph is the object diagram representing the Activity to be transformed. After a couple of applications of our GTRs, that object diagram is transformed into an object diagram representing the target TAAL program.

GTRs can be presented in two ways: by explicitly showing the left-hand and right-hand graph or by merging them into one graph. In this paper, we have chosen the latter, one-graph approach. This implies that nodes and edges have to be annotated according to their function within the rule. There are 4 types of elements:

- Nodes and edges which remain unchanged are depicted with black, solid, thin lines.
- Nodes and edges created by the rule are depicted with green, solid, fat lines.
- Nodes and edges deleted by the rule are depicted with blue, dashed, thin lines.
- Nodes and edges which must not exist in the host graph for the rule to match are depicted with red, dashed, fat lines.

Having said all that, let us now dive into the details. To specify our transformation, we had to implement 8 main transformation rules. Table 1 shows these rules and briefly

Table 1. Transformation rules and their tasks

Rule name	Task
Activity	Creates TAAL skeleton (`Program`, class `ExecutionClass` etc.)
Action_Implementation	Creates (empty) TAAL operation definitions for every `Action`, adds it to `ExecutionClass`
Action_Invocation	Creates TAAL operation call for every `Action`, “wraps” it in `BlockStat`
Sequence	Merges two `BlockStats` into one, according to two sequential building blocks
Decision	Creates TAAL if-then-else structure from according `DecisionNode` and `MergeNode`
While	Creates TAAL while-do structure from according `MergeNode` and `DecisionNode`
Fork	Creates TAAL fork structure from according `ForkNode` and `JoinNode`
Initial	Sets the remaining `BlockStat` as `Statement` of `main()` method

states their task within the transformation process. In the following, we will first explain the general idea of our transformation, and we will then by way of example show one of our transformation rules.

The transformation follows the inductive definition of well-formedness of UML Activities. A building block can be translated as soon as its included building blocks have been transformed. On the Activity side this is achieved by *reducing* translated structures to simple, structure-less building blocks. While the UML Activity thus gets simpler and simpler during the transformation, we at the same time build up the corresponding structures on the TAAL side which grows. To remember which building block belongs to which part of the TAAL constructs, we use *correspondence nodes*: Each correspondence node is connected to a UML Activity building block (depicted on the left side) and to the corresponding TAAL construct (depicted on the right side). Note that this approach is inspired by *Triple Graph Grammars* (TGGs, [19]), which explicitly consist of a left-hand graph, a right-hand graph and a correspondence graph associating constructs from the left side with their pendants on the right side. Note also that the concept of building blocks and correspondence nodes are only used within the GTRs; consequently, they do not appear in the meta-models of the two languages.

We want to illustrate this approach with an example transformation rule. Its task is to transform a certain Activity structure into a `while` loop, and it is depicted as Fig. 5. The rule can be applied if the graph to be transformed contains the structure which can be seen on the left side of the rule. Note the two `ActivityEdges` at the top and at the bottom of the Activity structure: They are the connection to the rest of the Activity and are therefore not deleted.

The part within the two `ActivityEdges` is the actual loop: A `MergeNode` is followed by a `DecisionNode` which has two outgoing `ActivityEdges`: the bottom one is the edge mentioned before, the left edge leads to the body of the loop. This body is in fact a building block – it represents some arbitrary (but well-formed) structure which

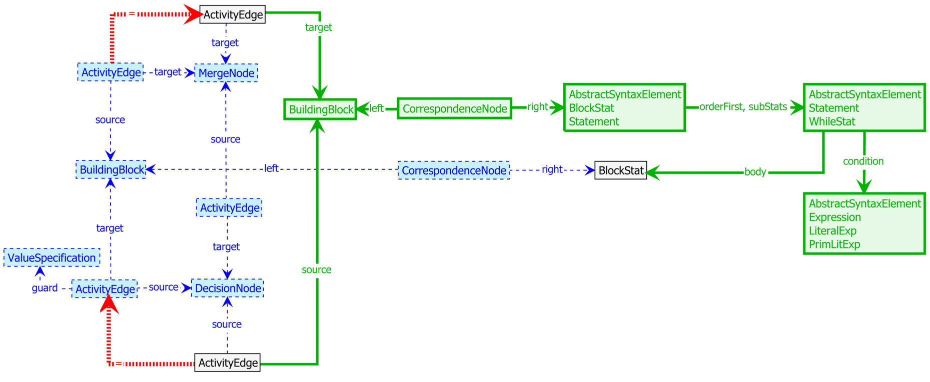


Fig. 5. Rule creating a While loop

has already been transformed. For instance, the building block could have been a single `Action`; in this case, it would have been a result of applying the Action_Invocation rule. It may also have been a more complex structure which has been reduced to one building block by applying a sequence of transformation rules.

Now, note that the building block has an association to a correspondence node, which has a `BlockStat` node as its right side. That `BlockStat` node is the result of the one or more transformation steps described above (the ones which finally resulted in the single building block on the correspondence node's left side).
The rule basically performs three changes on the host graph:

1. It creates the TAAL elements forming a `while` loop, i.e., the `WhileStat`, a wrapping `BlockStat` and some elements representing the loop's condition.
2. It sets the `BlockStat` corresponding to the building block as the body of the `while` loop.
3. It deletes the loop structure on the Activity side, replaces it with a single building block and creates a new correspondence node associating that building block with the wrapping `BlockStat` mentioned above.

Similar rules are used to treat simple `Actions`, sequences of building blocks and the Decision-Merge structure. In addition, we employ the Action_Implementation rule to create operation definitions on the TAAL side, the Activity rule to create the execution infrastructure like e.g. class definition, and the Initial rule to fill the `main`-method. Together, these rules perform a transformation of a meta-model instance of UML Activities into a meta-model instance of TAAL programs, from which we can then derive the concrete syntax TAAL program.

3 Behaviour Preservation

Recall from the introduction that our final goal is to prove that our transformation is *behaviour preserving*. In this section, we explain the notion of semantic equivalence we have in mind, and we argue on our example that our transformation fulfils this requirement. To this end, we first of all need to explain the formal semantics of the two languages, and most importantly, fix the notion of *equivalence* used in the comparison ($\approx$ in Fig. 1). As models of behaviour, we use the standard notion of *transition system*.

Definition 1. *A transition system* $(Q, \rightarrow, q_0)$ *over some alphabet* A *consists of a set of* states Q*, a transition relation* $\rightarrow \subseteq Q \times A \times Q$*, and an initial state* $q_0 \in Q$*. The set of transition systems with alphabet* A *is denoted* $TS[A]$.

Some related notation:

$$q \xrightarrow{a} q' \iff (q, a, q') \in \rightarrow$$
$$q \xrightarrow{a_1 \cdots a_n} q' \iff \exists q_1, \ldots, q_{n+1}. q = q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_{n+1} = q'$$

A transition system captures the behaviour of a model (or program) if it comprises precisely the execution steps that the model specifies (or the program executes). A single execution step is captured by a transition. A run of the system is captured by a connected sequence of transitions, or in other words, a path through the transition system. Note that the individual transitions, or execution steps, are thought of as atomic; this imposes a limit on the size of the steps that can be captured by a single transition, since on too coarse a level of granularity, the atomicity assumption is not justified (large execution steps may overlap, interfere or be aborted). In consequence, as we will see, we end up with a rather "small-step" semantics.

The mechanism for extracting a transition system from a model is what we call the behavioural semantics of the model (or, more precisely, of the modelling language). In the case of UMLA and TAAL, this mechanism uses the same graphs as the model transformation, and again works by means of graph transformation systems: see [9,12], respectively. In a first step, the static graphs are enhanced to incorporate run-time specific aspects (e.g., a token in the case of UMLA and a program counter in the case of TAAL). A graph transformation system, combined with the start graph that is given by the abstract syntax graph of the model, gives rise to a transition system in the following way:

– Each state is a graph;
– Each transition is the application of a transformation rule, where the label of the transition is given by the name of the applied rule;
– The initial state is given by the start graph;
– Whenever a graph transformation rule is applicable to a state, the corresponding rule application is a transition and the resulting graph is a state.

Thus, every well-formed UML Activity gives rise to a transition system, as does every TAAL program. When, then, do two transition systems describe the *same* behaviour? This is a question that has received much attention, especially in the context of *process algebra* (see [20]). It has become clear that there is no single answer that is satisfactory in all cases; rather, "sameness" can be captured by one of a range of so-called *equivalence relations* over transition systems; see, e.g., [21]. The *weakest* (most liberal) notion of "sameness" is that of *trace equivalence*, which is defined as follows.

Definition 2 (trace equivalence). *Assume that the alphabet A is partitioned into a set of internal and external actions, A_{in} and A_{ex}. A* trace *of a transition system $T \in TS[A]$ is a sequence $a_1 \cdots a_n$ with $a_i \in A_{\mathsf{ex}}$ for all $1 \leq i \leq n$, for which there is a path*

$$q_0 \xrightarrow{w_0\, a_1\, w_1\, a_2 \cdots w_n\, a_n\, w_{n+1}} q'$$

such that $w_i \in A^*_{\mathsf{in}}$ for all $0 \leq i \leq n+1$. The traces of T are collected in $\mathit{Traces}(T)$.

Two transition systems $T_1, T_2 \in TS$ are trace equivalent*, denoted $T_1 \approx T_2$, if $\mathit{Traces}(T_1) = \mathit{Traces}(T_2)$.*

In our case, the issue of equivalence is more complicated yet: despite the similarity of the semantic definitions of UMLA and TAAL, we cannot directly apply the existing theory, since the transition systems under comparison do not have the same alphabets. The reason for this is simple: as labels we have the rule names of the graph transformation system, and these are different for both languages. Furthermore, and more subtly, the granularity of the execution steps is not the same: in UMLA, executing an activity is based on the movements of tokens, whereas in TAAL it is based on a program counter; these mechanisms obey different rules, and hence moving a token from one activity to the next comprises different steps, in a different order, than moving a program counter from one method invocation to the next.

Our solution to this problem is to identify *one* rule in the graph transformation system for UMLA as well as the one for TAAL that we take to represent the "actual" execution of the action (on the one hand) or method (on the other).[1] For this rule, instead of using the rule name as label in the transition system, we use the name of the action. Thus, the functions sem_{UMLA} and sem_{TAAL} in Fig. 1 map each UMLA resp. TAAL abstract syntax graph to the transition system constructed as per the algorithm above, with the modified transition labelling. All other rule names are interpreted as *internal* in the sense of Def. 2.

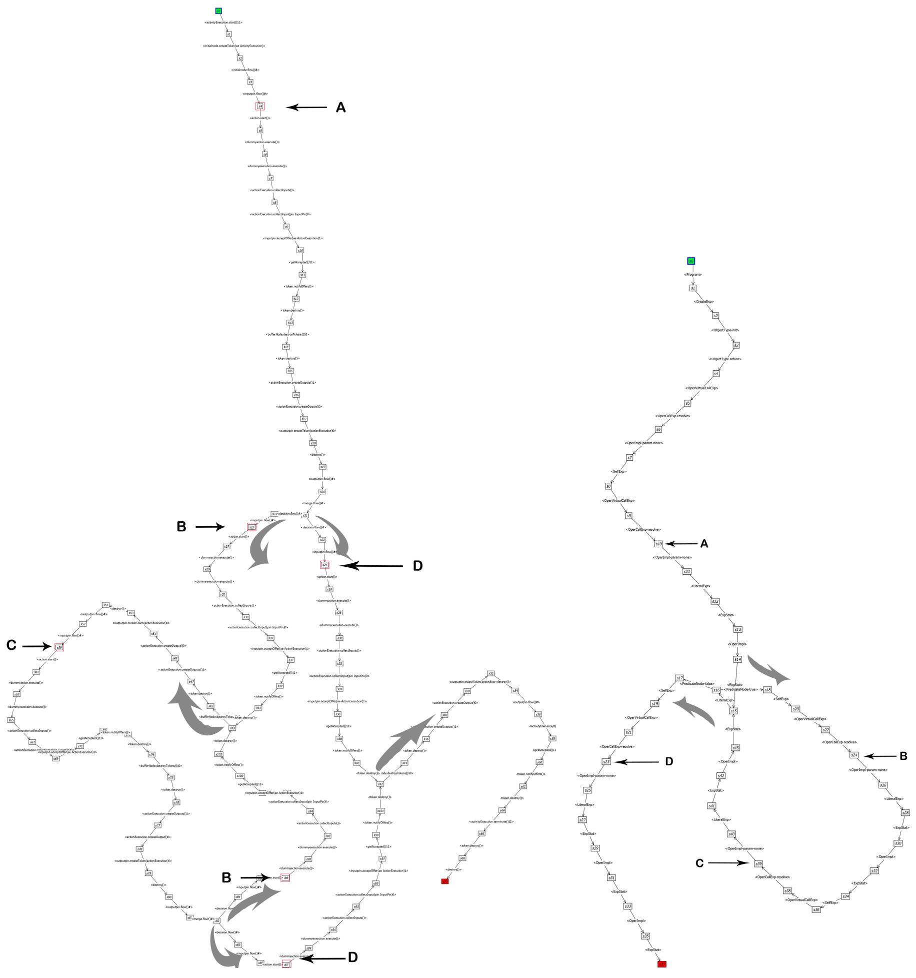
The core challenge of our approach is then to prove that for all UMLA graphs G, the following holds:

$$sem_{TAAL}(MT(G)) \approx sem_{UMLA}(G) \ . \tag{1}$$

Our claim is that (1) indeed holds. As an example, Fig. 6 shows the transition systems derived from the example Activity and the resulting TAAL program of Fig. 2. We start with the TAAL transition system (2.b), which exactly looks as expected: The loop can immediately be identified. A closer investigation reveals that the set of traces is also as expected: First, the `A()` operation is executed, followed by an arbitrary number of executions of the sequence `B()`–`C()`, and finally the `D()` method is executed. Note that all this happens within the execution of the `main()` operation, i.e., we do not take that operation into account. The set of traces of the TAAL program can thus be described by the regular expression $\mathtt{A}(\mathtt{B}\ \mathtt{C})^*\mathtt{D}$.

The Activity's transition system (2.a) looks different, though: It seems to contain two loops. These loops are due to the *traverse-to-completion* semantics of UML Activities [22]. Still, this does not affect the correctness in our chosen criterion: the UMLA transition system gives us exactly the same set of traces over Actions, namely $\mathtt{A}(\mathtt{B}\ \mathtt{C})^*\mathtt{D}$.

[1] For UMLA this is a rule called `action.start()`; for TAAL it is `OperVirtualCallExp`.



(a) UML Activity (b) TAAL

Fig. 6. The transition systems of the example

We have carried out this comparison on a large number of examples, involving different structures of the UML Activity, in particular also with more complex nestings of Decision and Merge nodes. In all of these examples, the resulting transition systems were trace equivalent. Nevertheless, we see this only as a first step towards showing behaviour preservation, and our ultimate aim is a general proof of correctness for the transformation, in the sense of 1.

4 Tool Support

The preceding sections have shown that in our setting, the semantics of UML Activities and TAAL programs as well as our transformation is specified by means of graph transformation rules. Therefore, we rely on a tool which supports the creation and application of graph transformation rules. As we have mentioned before, we use the tool Groove [11] for this purpose.

For the transformation tool, our main requirement is that the transformation can be specified formally – otherwise, we would not be able to perform a formal proof of correctness. Since most of us already were experienced Groove users, it was an obvious choice to also use that tool for defining the transformation itself. This section will detail our reasons for that choice, and it will point out some particular strengths of Groove.

Despite the "standard" graph transformation features like creation and deletion of nodes and edges, Groove supports some more advanced concepts which allowed us to specify our transformation as desired. First, Groove supports *attributed graphs*, which e.g. allowed us to create TAAL operations having the same name as their corresponding `Actions`. Additionally, we were able to specify the Action_Implementation rule in such a way that one TAAL operation is created for every *name* of an `Action`; for an Activity that contains two `Actions` named A, this results in a TAAL program with one A operation, but two invocations of that operation.

Second, a powerful notion of *universal quantification* has been implemented in Groove. In a nutshell, this means that rules can be written which manipulate *all* occurrences of a node in a certain context. While implementing our transformation, this was of particular importance for the Sequence rule: Recall from Sect. 2 that this rule merges two `BlockStats` into one, and part of this is to add all sub statements of one `BlockStat` to the resulting `BlockStat`. Universal quantification allowed us to implement this behaviour within one rule.

Another reason for choosing Groove was that the transformation rules we defined basically relate parts of a UML Activity with their corresponding parts on the TAAL side, in contrast to an operational transformation specification (e.g. in Java). Since relating elements of source and target models will probably be an important part of our proof, we hope to reuse the transformation rules for this purpose.

Figure 7 shows a screenshot of Groove. On the left side, the names of the transformation rules can be seen. Note that at the bottom of the rule's compartment, a couple of rules are shown whose names start with "Failure". These rules match if certain structures exist in a state which would indicate that the transformation has failed. Note also that these rules have a priority of 0: This makes sure that the failure rules can only match if none of the transformation rules matches any more (i.e., after the complete transformation has been carried out).

The big compartment on the right shows the start state representing the Activity as introduced in Fig. 2. Note that Groove allows to hide parts of the displayed graph; we have hidden the Activity node and its edges to the Activity's element to reduce the complexity of the graph's visualisation. Note also the `DMMSystem` node to the left of the graph: This node and the associated `Invocations` are needed for the graph transformation rules describing the Activity's semantics. They are deleted by our transformation.

In order to use Groove, we translated the UML Activity under consideration into a suitable format. For this, we have written an Eclipse [23] plugin which takes a UML Activity model as input and generates a Groove state graph out of it. The Activity is given in the XMI format which is then read and processed using the API of the Eclipse

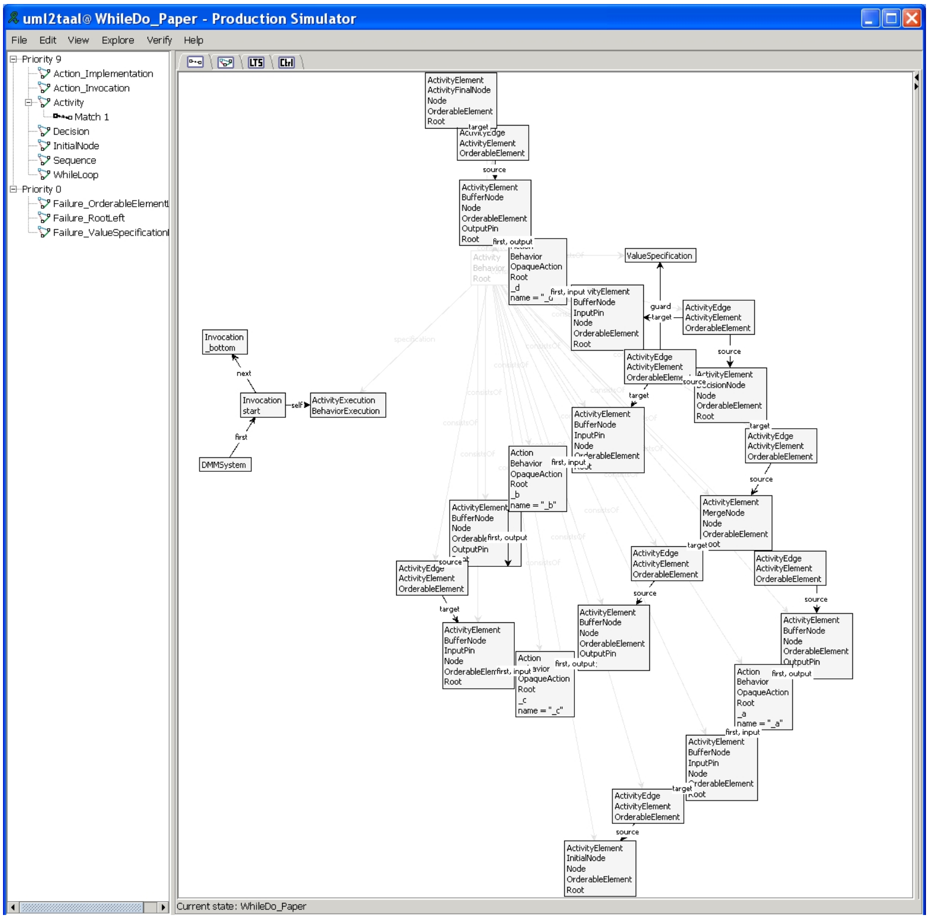

uml2taal@ WhileDo_Paper - Production Simulator
File
Edit
View
Explore
Verify
Help
Priority 9
Action_Implementation
Action_Invocation
Activity
Match 1
Decision
InitialNode
Sequence
WhileLoop
Priority 0
Failure_OrderableElement
Failure_RootLeft
Failure_ValueSpecification
LTS
Ctrl
Invocation _bottom
next
Invocation start
self
ActivityExecution BehaviorExecution
first
DMMSystem
ValueSpecification
guard
target
source
first, output
first, input
specification
consistsOf
ActivityElement ActivityFinalNode Node OrderableElement Root
ActivityEdge ActivityElement OrderableElement
ActivityElement BufferNode Node OrderableElement OutputPin Root
ActivityElement BufferNode InputPin Node OrderableElement Root
ActivityElement DecisionNode Node OrderableElement Root
ActivityElement MergeNode Node OrderableElement Root
Action Behavior OpaqueAction Root _a name = "_a"
Action Behavior OpaqueAction Root _b name = "_b"
Action Behavior OpaqueAction Root _c name = "_c"
Behavior OpaqueAction Root _d name = "_d"
ActivityElement InitialNode Node OrderableElement Root
Current state: WhileDo_Paper

Fig. 7. Screenshot of the tool Groove

UML2 project. Since the Activity's model basically *is* the graph to be processed, the generation is straight-forward.

On the TAAL side, a similar Eclipse plugin exists: It transforms TAAL programs into Groove-format abstract syntax graphs and back again. Being able to generate a TAAL program's concrete syntax from a graph turned out to be very helpful for the validation and debugging of our model transformation.

5 Conclusion

In this paper, we have developed a model transformation from UML Activities to TAAL, defined a notion of correctness of the transformation and argued that—based on the formal semantics of the two languages—the transformation is indeed correct. Transformations from UML models to object-oriented programming languages are frequently employed in a model-based development, and thus their correctness constitutes an important part of MDD. Our ultimate goal and future work is a formal proof of correctness.

The contribution of this work does, however, go beyond this specific transformation. Although the two modelling languages are conceptually very different, a comparison can be carried out. There are a number of important issues which helped towards this goal. First of all, it is the existence of meta-models (of the same language) which facilitated the definition of the transformation. Secondly, indispensable for a correctness proof is (a) a formal definition of the transformation (here given because of the use of graph transformation systems) and (b) a formal definition of the semantics of the languages. Crucial is also the (formal) definition of the employed notion of equivalence; for this, a *common* semantic domain of the languages is important. Last but not least, such a comparison would not have been possible without a tool for executing the model transformation.

The method proposed in this paper for the comparison of behavioural semantics is obviously only applicable if the modelling languages in question are indeed behavioural. Moreover, it should be possible to express their semantics by means of transition systems. Fortunately, the transition system formalism is itself quite general, so we do not expect this to be a limiting factor.

References

1. Akehurst, D.H., Kent, S.: A Relational Approach to Defining Transformations in a Metamodel. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 243–258. Springer, Heidelberg (2002)
2. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: Workshop on Generative Techniques in the Context of Model-Driven Architecture (2003)
3. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2007), `http://www.omg.org/cgi-bin/doc?ptc/2007-07-07`
4. Küster, J.M., Abd-El-Razik, M.: Validation of Model Transformations - First Experiences Using a White Box Approach. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007)
5. Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination Criteria for Model Transformation. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 49–63. Springer, Heidelberg (2005)
6. Lambers, L., Ehrig, H., Orejas, F.: Conflict Detection for Graph Transformation with Negative Application Conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 61–76. Springer, Heidelberg (2006)
7. Mens, T., Demeyer, S., Janssens, D.: Formalising Behaviour Preserving Program Transformations. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 286–301. Springer, Heidelberg (2002)
8. Ruhroth, T., Wehrheim, H.: Refactoring Object-Oriented Specifications with Data and Processes. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 236–251. Springer, Heidelberg (2007)
9. Kastenberg, H., Kleppe, A., Rensink, A.: Defining Object-Oriented Execution Semantics Using Graph Transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)

10. Object Management Group: Business Process Modeling Notation V1.1 (2008), http://www.omg.org/spec/BPMN/1.1/PDF
11. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
12. Hausmann, J.H.: Dynamic Meta Modeling. PhD thesis, University of Paderborn (2005)
13. Karsai, G., Narayanan, A.: On the Correctness of Model Transformations in the Development of Embedded Systems. In: Kordon, F., Sokolsky, O. (eds.) Monterey Workshop. LNCS, vol. 4888, pp. 1–18. Springer, Heidelberg (2006)
14. Glesner, S.: Using Program Checking to Ensure the Correctness of Compiler Implementations. J. UCS 9(3), 191–222 (2003)
15. Denney, E., Fischer, B.: Certifiable Program Generation. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 17–28. Springer, Heidelberg (2005)
16. Peterson, W.W., Kasami, T., Tokura, N.: On the Capabilities of While, Repeat, and Exit Statements. Commun. ACM 16(8), 503–512 (1973)
17. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1. World Scientific, Singapore (1997)
18. Klar, F., Königs, A., Schürr, A.: Model Transformation in the Large. In: Crnkovic, I., Bertolino, A. (eds.) ESEC/SIGSOFT FSE, pp. 285–294. ACM, New York (2007)
19. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
20. Bergstra, J.A., Ponse, A., Smolka, S.A.: Handbook of Process Algebra. Elsevier, Amsterdam (2001)
21. van Glabbeek, R.J.: The linear time – branching time spectrum I: The semantics of concrete, sequential processes. In: [20], pp. 3–100
22. Bock, C.: UML 2 Activity and Action Models, Part 4: Object Nodes. Journal of Object Technology 3(1), 27–41 (2004)
23. Eclipse Foundation, http://www.eclipse.org