UML Action Semantics for Model Tranformation Systems*

Dániel Varró and András Pataricza
Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1521 Budapest Magyar tudósok körútja 2.
{varro,pataric}@mit.bme.hu

Abstract

The Action Semantics for UML provides a standard and platform independent way to describe the behavior of methods and executable actions in object-oriented system design prior to implementation allowing the development of highly automated and optimized code generators for UML CASE tools. Model transformation systems provide visual but formal background to specify arbitrary transformations in the Model Driven Architecture (the leading trend in software engineering). In the current paper, we describe a general encoding of model transformation systems as executable Action Semantics expressions to provide a standard way for automatically generating the implementation of formal (and provenly correct) transformations by off-the-shelf MDA tools. In addition, we point out a weakness in the Action Semantics standard that must be improved to achieve a stand-alone and functionally complete action specification language.

Keywords: Action Semantics, model transformation, MDA, UML, graph transformation.

1 Transformations in the Model Driven Architecture

1.1 Towards the Model Driven Architecture

Recently, the main trends in software engineering have been dominated by the **Model Driven Architecture (MDA)** [17] vision of the Object Management Group (OMG). According to MDA, software development will be driven by a thorough modeling phase where first (i) a *platform independent model* (PIM) of the business logic is constructed from which (ii) *platform specific models* (PSMs) including details of the underlying software architecture are derived by *model transformations* followed by (iii) an automatic generation of the target application code.

*This work was supported by the Hungarian Information and Communication Technologies and Applications Grant (IKTA 00065/2000), the Hungarian National Scientific Foundation Grant (OTKA 038027) and the Timber Hill Foundation

The PIMs and PSMs are defined by means of the Unified Modeling Language (UML) [21], which has become the *de facto* standard visual object-oriented modeling language in systems engineering. Moreover, the recent inclusion of an action specification language to the UML standard (Action Semantics for UML [13, 16]) seems to become a breakthrough for tool vendors to develop highly automated and optimized code generators for UML CASE tools (such as [11, 19]) with an executable action specification language.

However, UML still lacks a precise formal semantics, which hinders the formal verification and validation of system design. Moreover, several shortcomings of the language have been revealed in domain specific applications as well. To provide formal semantics, UML models are frequently projected into various mathematical domains (Petri nets, transition systems, process algebras, etc.), and the results of the formal analysis are back–annotated to the UML-based system model to hide the mathematics from designers [3, 10, 28].

1.2 Model Transformations in the MDA Environment

As the upcoming UML 2.0 standard aims at rearchitecturing UML into a family of individual languages built around a small kernel language, different kinds of model transformations will play a central role for UML as well as for the entire MDA approach.

- *model transformations within a language* should control the correctness of consecutive refinement steps during the evolution of the static structure of a model, or define a (rule-based) operational semantics directly on models;
- *model transformations between different languages* should provide precise means to project the semantic content of a diagram into another one, which is indispensable for a consistent global view of the system under design;
- a visual UML diagram (i.e., a sentence of a language in the UML family) should be transformed into its (individually defined) semantic domain, which process is called *model interpretation*.

The VIATRA model transformation system VIATRA (VIsual Automated model TRAnsformations [5, 28]) is a prototype tool that provides a general and automated framework for specifying transformations between arbitrary models conforming to their metamodel.

The main characteristics of VIATRA are the following:

- The precise theoretical background of the transformations is formalized by means of graph transformation systems [20].
- From visual model transformation rules defined in a UML notation, VIATRA automatically generates a Prolog program for the implementation [25].
- Moreover, the semantic correctness of transformations can be proven by model checking techniques [26].

Related work Other existing model transformation approaches can be grouped into two main categories:

- *Relational (or bidirectional) approaches*: these approaches typically *declare a relationship* between objects (and links) of the source and target language. Such a specification typically based upon either (a) a metamodel with OCL constraints [1, 2, 14], (b) textual mappings [9], or (c) triple graph grammars [22].
- *Operational (or unidirectional) approaches*: these techniques *describe the process* of a model transformation from the source to the target language. Such a specification mainly combines metamodeling with (d) graph transformation [6, 10, 24, 27], (e) term rewriting rules [29], or (f) XSL transformations [7, 18].

Unfortunately, none of these approaches provide a general solution for model transformation when evaluated according to their (i) expressive power, (ii) precise mathematical background, (iii) efficient implementation and (iv) relatedness to industrial standard.

- For instance, relational approaches providing bidirectionality might be convenient for many simple practical transformations but transformations with deliberate loss of information (such as abstractions) cannot be expressed.
- XSLT based solutions do not have a precise mathematical background, moreover, XSLT is very inefficient as a transformation language if our models are not trees but complex graph structures.
- Graph transformation (and rewriting) based approaches fulfill requirements (i) – (iii) but their concepts are far from the industrial standards which hinders their acceptance in the UML environment.

Problem statement In order to facilitate the widespread use of model transformations with graph transformation as the formal background, we have to integrate them into existing MDA standards and tools. In other words, academic tools (like VIATRA) are useful for the experimentation and verification of model transformations, but *the final (product quality) implementation should be integrated into the MDA approach and the UML standard as much as possible*.

Our contribution In the current paper, we aim at transforming model transformation systems into standard Action Semantics descriptions to allow the automatic generation of transformation scripts for various software architectures by off-the-shelf UML tools. As a result, our visual but formal specification technique [28] based on metamodeling and graph transformation may become the first approach to fulfill all the four requirements of general purpose model transformations in the MDA environment.

The structure of the paper The rest of the paper is structured as follows. Section 2 provides a brief introduction to the concepts and formal treatment of model transformation systems while Sec. 3 gives and overview of the UML Action Semantics standard.

Then, in Sec. 4, which is the key part of the paper, we describe a general encoding of model transformation systems into Action Semantics expressions. Finally, Sec. 5 concludes our paper.

2 Formalizing Model Transformations

2.1 Models and metamodels

The *abstract syntax* of visual modeling languages is defined by a corresponding metamodel in a UML notation (i.e., a simplified class diagram), which conforms to the best engineering practices in visual specification techniques. Models (defined in the form of UML object diagrams) are sentences of those languages, thus each well-formed model has to conform to its metamodel. Typically, models and metamodels are represented internally as typed, attributed and directed graphs.

- On the metamodel-level (or class level), classes can be mapped into a graph node and all associations are projected into a graph edge in the **type graph** (denoted as TG). The inheritance hierarchy of metamodels can be preserved by an appropriate *subtyping relation* on nodes (and possibly, on edges). Class attributes are derived into graph attributes where the latter may be treated mathematically as (possibly partial) functions from nodes to their domains.

- On the model level (or object level), objects and links between them are mapped into nodes and edges, respectively, in the **model (instance) graph** (denoted as M). Each node and edge in the model graph is related to a corresponding graph object in the type graph by a corresponding *typing homomorphism* [4].

A sample metamodel and a simple model of finite automata are depicted in Figure 1.

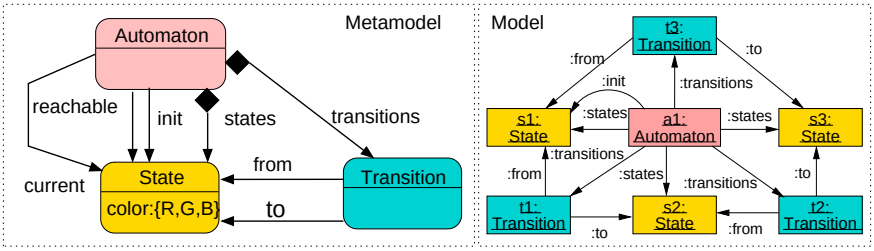

Figure 1: A metamodel and model of finite automata

Example 1 According to the metamodel, a well–formed instance of a finite *automaton* is composed of *states* and *transitions*. A transition is leading between its *from* state and *to* state. The initial states of the automaton are marked with *init*, the active states

are marked with *current*, while the reachable states starting from the initial states are modeled by *reachable* edges.

A sample automaton a1 consisting of three states (s1, s2, s3) and three transitions between them t1 (leading between s1 and s2), t2 (leading between s2 and s3), and t3 (leading between s2 and s3) is also depicted. We can notice that the initial state of a1 is s1.

2.2 Model Transformation Systems

The *dynamic operational semantics of a modeling language* as well as *transformations between modeling languages* can be formalized by *model transformation (transition) systems* (introduced in [28]), which is a variant of graph transformation systems with a predefined set of control structures.

A **graph transformation rule** is a 3-tuple $Rule = (Lhs, Neg, Rhs)$, where Lhs is the left-hand side graph, Rhs is the right-hand side graph, while Neg denote the (optional) negative application conditions.

The **application of a rule** to a model graph M (e.g., a UML model of the user) rewrites the user model by replacing the pattern defined by Lhs with the pattern of the Rhs. This is performed as follows.

1. *Find a match* of Lhs in M (graph pattern matching).
2. *Check the negative application conditions* Neg which prohibit the presence of certain nodes and edges in the model graph. Informally, if the match of the Lhs pattern can be extended to include the match of the Neg pattern then the original match of the Lhs is eliminated. Negative application conditions are denoted by graph objects with a cross.
3. *Remove* a part of the graph M that can be mapped only to the Lhs but not to the Rhs graph (i.e., to $Lhs \setminus Rhs$) in order to yield the *context graph.*
4. *Glue* the image of the Rhs and the context graph to obtain the derived model M', which means the creation of certain model elements (nodes and edges).

The entire model transformation process is defined by an initial graph manipulated by a set of graph transformation rules (micro steps) executed in a specific mode in accordance with the semantics (macro steps) of a hierarchical control flow graph.

A **model transformation (transition) system** $MTS = (Init, R, CFG)$ with respect to (one or more) type graph TG is a triple, where $Init$ defines the **initial graph**, R is a set of **graph transformation rules** (both compatible with TG), and CFG is a set of a **control flow graphs** defined as follows.

- There are five types of nodes of the CFG: **Start**, **End**, **Try**, **Forall**, and **Loop**.
- There are two types of edges: **succeed** and **fail**.

The control flow graph is evaluated by a virtual machine which traverses the graph according to the edges and applies the rules associated to each node.

1. When a *Try* node is reached, its associated rule is tried to be executed. If the rule was applied successfully then the next node is determined by the *succeed* edge, while in case the execution failed, the *fail* edge is followed.

2. At a *Loop* node, the associated rule is applied as long as possible (which may cause non-termination in the macro step). Only a *succeed* edge may lead from a *Loop* node.

3. When a *Forall* node is reached, the related rule is executed parallelly for all distinct (possible none) occurrences in the current host graph. Only a *succeed* edge may lead from a *Forall* node.

Note that this CFG model follows the control flow concepts of the VIATRA tool. However, the use of "as long as possible" kind of control conditions (and additional negative application conditions) instead of *forall* nodes would almost directly yield the appropriate control conditions for many existing graph transformation tools.

Example 2 A pair of rules describing how the reachability problem on finite automata can be formulated by graph rewriting rules is depicted in Figure 2. Rule *initR* states that all states of the automaton marked as initial are reachable (if the state has not been marked previously). Rule *reachR* expresses that if a reachable state S_1 of the automaton is connected by a transition T_1 to such a state S_2 that is not reachable yet then S_2 should also become reachable as a result of the rule application.

Note that without the negative application condition (the crossed reachable edge in the left-hand side of the rule), the transformation might generate more than a single reachable edge between an automaton and a state, which contradicts our intuitive requirements.

According to the control flow graph, first we have to apply *initR* in *forall* mode, then *reachR* should be executed as long as possible. Since all edges in the control flow graph are *succeed* edges, it is not explicitly depicted in the figure.

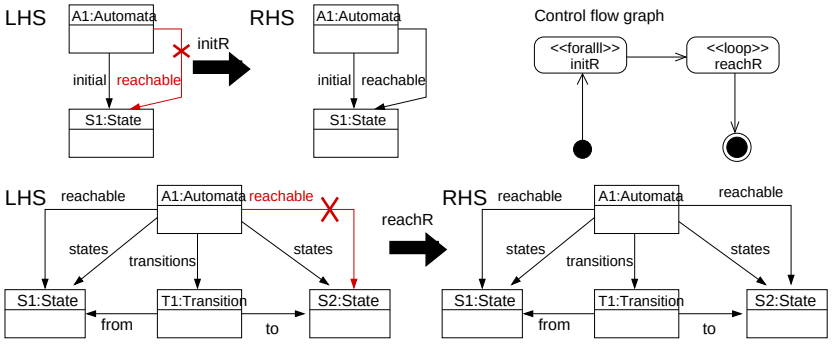

Figure 2: Calculating reachable states by graph transformation

3 Action Semantics for UML: An Overview

The **Action Semantics for UML (AS)** provides a standardized and platform (and implementation) independent way to specify the behavior of objects in a distributed environment. Basically, the user can describe the body of methods and executable actions in an abstract language prior to the implementation phase by constructing a dataflow like model.

Action specification An action specification consists of the following elements

- **Pins**: the input and output "ports" of an action having a specific type and multiplicity (a pin may hold a collection of values at a time if it is allowed by its multiplicity),
- **Variables**: an auxiliary store for results of intermediate computations,
- **Data flow**: connects the output pin of one action to the input pin of another thus providing an implicit ordering of action execution,
- **Control flow**: imposes an explicit ordering constraint for action pairs having no connecting data flow,
- **Actions**: for object manipulation, memory operations, arithmetic, message passing, etc.,
- **Procedures**: provides the packaging of actions with input and output pins, e.g., for method body.

Action execution The execution of an action has the following stages in its life-cycle.

- **Waiting.** An action execution may be created at any time after the procedure execution for its containing procedure is created. On creation, an action execution has the status waiting and no pin values are available.
- **Ready.** An action execution with status waiting becomes ready on the completion of the execution of all prerequisite actions (that is all actions that are the sources of data flows or predecessors of control flows into the action becoming ready). The values of the input pins of the target action execution are determined by the values of the output pins from the prerequisite action executions for actions via data flows.
- **Executing.** Once it is ready, an action execution eventually begins executing (the action semantics does not determine the specific time delay (if any) between becoming ready and actually executing).
- **Complete.** When it has finished execution, the action becomes complete. The action execution then has pin values for all output pins of the action computed according to specific semantics of different actions. After the output values from a completed execution have been copied, there is no longer any way for another execution to access the completed execution.

Actions have no default orderings (like sequential execution as in traditional programming languages); actions that are not implicitly ordered by data flow or explicitly ordered by control flow can be executed either parallelly or in an arbitrary order.

Types of actions Specific semantics of different kind of actions can be grouped into the following main categories (only actions relevant for the encoding of Sec. 4 are enlisted).

- **Computation actions** are primitive actions for mathematical functions (not defined in the standard in details)
- **Composite actions** are recursive structures that permit complex actions to be composed of simpler ones providing means for basic control flow actions (e.g., LoopAction, ConditionalAction, GroupAction),
- **Read and write actions** access, navigate, and modify model-level constructs (such as objects, links, attribute slots, and variables)
- **Collection actions** (such as FilterAction, MapAction, or IterateAction) apply a subaction to a collection of elements to avoid overspecification of control caused by explicit indexing and extracting of elements.

Syntax of actions The Action Semantics standard only defines a metamodel (and some well-formedness constraints) for the language without any restrictions on concrete syntax. In this respect, a well-formed action expression itself is a rather complex object diagram, which is easy to process for CASE tools but extremely hard to read and write system engineers. In fact, existing UML CASE tools with an integrated action specification language have their own textual notations for describing actions.

Therefore, the encoding of model transformation systems will be presented in the sequel on two levels: (i) first, in an own, self-explanatory pseudo action language to understand the overall idea of the encoding (instead of sticking to any specific existing dialects of AS tools), (ii) and then in a standardized way, by using object diagrams (coping with AS technicalities).

Example 3 In order to provide an overview of Action Semantics, a simple fragment of an AS expression is presented in Fig. 3, which captures the behavior of a boolean function testing whether the value of variable factor is equal to the constant 2.

- For this purpose, we should first read the values of the constant 2 and the variable factor into corresponding output pins by using AS actions LiteralValueAction and ReadVariableAction, respectively.
- Then both values are transported to the argument input pins of ApplyFunctionAction by corresponding DataFlow operations.
- Finally, ApplyFunctionAction applies the primitive function == supplied with the previous arguments, and stores the results in its op1 output pin.

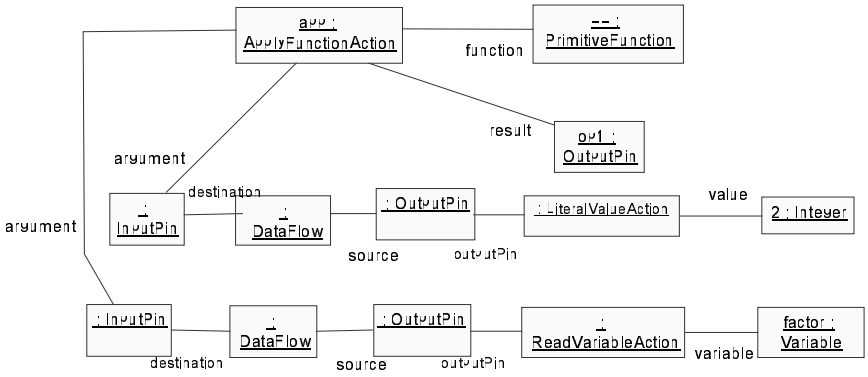

Figure 3: An Action Semantics expression for `factor==2`

4 Action Semantics for Model Transformation Systems

In this section, we provide a general way to encode model transformation systems into Action Semantics (AS) descriptions to provide a standard and platform independent way to implement transformations in the MDA environment.

Our *generation technique* takes the metamodel(s), and a model transformation system as the *input*, and generates a set of actions as the *output*. The *results of the transformations* are obtained afterwards in the form of *an object/collaboration diagram*.

A natural *correctness criteria* of the generation process would be that the execution of actions according to the Action Semantics standard should yield identical result with (at least one trace of) the formal generation process driven by the model transformation system itself starting from a given instance model. However, as the AS standard completely lacks any formal semantics, unfortunately, we cannot formally reason about the correctness of our approach.

The overall idea basically follows the graph pattern matching techniques implemented in the PROGRES [23] and FUJABA [15] tools in procedural and object-oriented languages. The encoding consists of the following main steps (which will be introduced in details as “Solutions” later on in this section):

- *implementing graph pattern matching* by local searches in the user model based on collection actions and navigation capabilities of AS;
- *checking non-existence* of certain objects and links prescribed by the negative conditions by a user defined function;
- *adding and deleting graph objects* by using actions for object and link manipulation;
- *implementing rule application modes* by various corresponding collection actions;

- *simulating the execution of the control flow graph* by conditional actions and explicit control flow restrictions.

The encoding will be introduced on our running example of the reachability problem of finite automata, which includes the handling of all major problems.

4.1 Encoding the control flow graph

The handling of the control flow graph consists of modeling rule applications in a certain mode and defining the sequence of consecutive transformation steps.

Solution 1 For each rule applied in *loop* or *forall* mode, a GroupAction is generated, while a ConditionalAction is generated for a rule applied in *try* mode.

Solution 2 The sequence of rule applications are defined by explicit ControlFlow restrictions set upon the sequence of corresponding rule actions.

As only *succeed* edges may lead from *loop* and *forall* nodes of a CFG such rules are modeled by GroupActions, which is simply a collection of subactions. However, in case of *try* rules, the CFG branches depending on the success of rule application, thus the corresponding action of a *try* rule must return whether the application of the rule was successful or not. After that, the composite actions of rules can be simple connected by ControlFlow objects in accordance with the CFG.

Example 4 The control flow graph of our reachability example is depicted in the AS notation in Fig. 4 stating that the execution of GroupAction *initR* should precede the execution of *reachR* action.

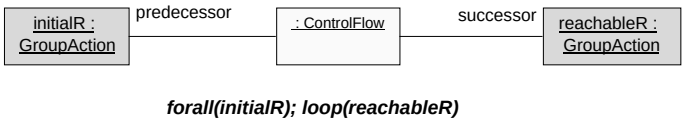

Figure 4: Control flow restrictions

4.2 Action semantics for pattern matching

The implementation of a graph pattern matching algorithm within Action Semantics is the central part of the entire encoding. The main challenge relies in the fact that graph transformation tools are traditionally control-oriented with global (constraint-based) graph pattern matching algorithms while Action Semantics provide a data flow based specification technique allowing only local navigations for pattern matching algorithms.

In addition, the encoding of pattern matching depends on the rule application mode, thus, the same rule may have different AS representations when applied in different application modes. *Loop* and *try* modes are handled almost similarly (*loop* mode is based upon *try* mode, since a rule is tried to be applied once as long as possible and the next application depends on the applicability of the current one), and they differ essentially from the behavior of *forall* mode (where rule applications are executed parallelly for each matching).

In the sequel, we discuss the encoding of a rule applied in *forall* mode on the demonstrative example of Algorithm 1 for *initR*, while the pseudo encoding of *reachR* is shown in Algorithm 2 with detailed explanations given later in the current section. (Note that node identifiers in rules correspond to variables to ease the comparison of graph transformation rules with their pseudo AS representation.)

Algorithm 1 Encoding *initR* in a pseudo action specification language

```
 1: GroupAction Automaton::initR() =
 2: Variable a1, s1, S1;
 3: a1 = ReadSelfAction();
 4: if ReadIsClassifiedObjectAction(Automaton, a1) then
 5:    {S1} = ReadLinkAction(a1, initial) ;
 6:    for all s1 ∈ {S1} do {MapAction; parallel execution}
 7:       if ReadIsClassifiedObjectAction(State, s1) then
 8:          if ¬ testLink(a1, reachable, s1) then
 9:             CreateLinkAction(a1, reachable, s1);
10:          end if
11:       end if
12:    end for
13: end if
```

Starting point of pattern matching The first step, which is to find the starting point for pattern matching is, in fact, identical for all modes.

Solution 3 The starting point of the pattern matching is identified by the instance retrieved by a ReadSelfAction executed on an instance of the model class (i.e., Automaton in our example) and stored in a variable (see Line 3 in Algorithm 1).

As for the AS representation, a data flow is required to connect the output pin of ReadSelfAction with the input pin of AddVariableValueAction action. We must also specify that the previous value stored in the variable should be overwritten by setting the isReplaceAll variable to true. Note that matching instances of LHS graph nodes will be stored in AS variables (later on as well).

Example 5 The AS representation of Line 3 in Algorithm 1 is depicted in Fig. 5. We expect to retrieve an Automaton instance stored in variable *a1*.

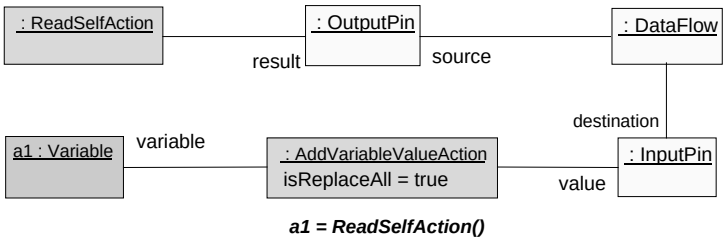

Figure 5: Starting point of pattern matching

Type checking of objects Our next problem to be solved is to visit only type conforming objects when matching patterns.

Solution 4 When a new object is obtained at any time during pattern matching (i.e., matched to a corresponding node in the graph transformation rule), we immediately test whether it has a conforming type (conformant to the type of the graph node). See Lines 4 and 7 in Algorithm 1 as examples.

This testing is performed by a ConditionalAction with a test clause consisting of a single ReadIsClassifiedObjectAction. The test action ReadIsClassifiedObjectAction (checking whether an object is an instance of a certain class) has to return a boolean value on its output pin, which serves as the test output for the test clause in the meanwhile. If value retrieved by the test subaction of a clause is true then the body action of the clause can be executed (which consists of further actions of pattern matching in our case).

Example 6 The AS representation of Line 7 in Algorithm 1 is depicted in Fig. 6. We check whether the value stored in variable *s1* is an instance of the class Automaton. Naturally, we first have to get this value from the variable by a ReadVariableAction connected to the input pin of ReadIsClassifiedObjectAction by a data flow.

Navigating links The core operation of graph pattern matching in a UML environment is the navigation of links.

Solution 5 When a certain object is matched, the neighbors of the object (connected by links corresponding to the edge types in the graph transformation rule) are obtained by navigating links (Line 5 in Algorithm 1). This navigation results in a single object or a collection of objects stored in a variable.

A navigation of a link in AS (by applying ReadLinkAction) means that

- the exactly one end object (called the source object) of a link is already known (i.e., at most one LinkEndData may have an associated single value on its input pin), while the target end of the link should be unknown yet (naturally, an association can be navigated in both directions if allowed by the metamodel);

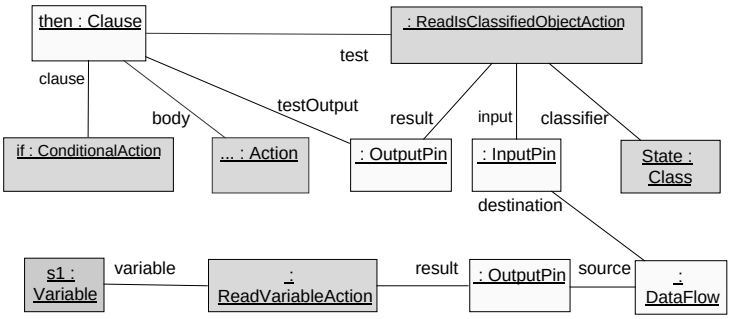

Figure 6: Checking types of objects

- the link should correspond to a certain association (also defined by the LinkEndData);
- as a result of the navigation a single object or a set of objects is retrieved (depending on multiplicities of the association and the topology of interconnected objects), and stored in a variable.

Example 7 The AS representation of Line 5 in Algorithm 1 is depicted in Fig. 7. We read the value of variable *a1* into the input pin of one LinkEndData corresponding to an association end of the initial association. When the ReadLinkAction is executed the result is written into variable *S1* (variables with a capital initial will store a collection of objects in the sequel).

Rule application mode specific processing of collections The processing of collections obtained from navigating links depends on rule application modes.

Solution 6 When processing a collection for the pattern matching of a rule applied in *forall* mode (see Line 6 in Algorithm 1), each element in the collection must be processed independently from each other, thus subsequent actions in the pattern matching process should be applied for each of them. For this reason, a MapAction is required in AS.

Solution 7 When processing a collection for the pattern matching of a rule applied in *try* or *loop* mode (see Lines 9, 12 and 15 in Algorithm 2), each element in the collection must be processed sequentially (thus an IterationAction is required in AS); however, the next element in the collection needs to be processed *only if* no complete matching has been found successfully so far (first boolean condition in Lines 10, 13 and 16).

Algorithm 2 Encoding *reachR* in a pseudo action specification language

```
1:  GroupAction Automaton::reachR() =
2:  Variable isApplicable, a1, s1, S1, s2, S2, t1, T1;
3:  AddVariableValueAction(isApplicable, T);
4:  while isApplicable do {LoopAction}
5:     AddVariableValueAction(isApplicable, F);
6:     a1 = ReadSelfAction();
7:     if ReadIsClassifiedObjectAction(Automaton, a1) then
8:        {S1} = ReadLinkAction(a1,reachable);
9:        for all s1 ∈ {S1} do {IterateAction, sequential execution}
10:          if ¬isApplicable ∧ ReadIsClassifiedObjectAction(State, s1) ∧
             testLink(a1,states, s1) then
11:             {T1} = ReadLinkAction(s1,source);
12:             for all t1 ∈ {T1} do {IterateAction; sequential execution}
13:                if ¬isApplicable ∧ ReadIsClassifiedObjectAction(Transition, t1) ∧
                   testLink(a1,transitions, t1) then
14:                   {S2} = ReadLinkAction(t1,target);
15:                   for all s2 ∈ {S2} do {IterateAction; sequential execution}
16:                      if ¬isApplicable ∧ ReadIsClassifiedObjectAction(State, s2) ∧
                         testLink(a1,state, s2) then
17:                         if ¬ (testLink(a1,reachable, s2)) then {Neg. condition}
18:                            AddVariableValueAction(isApplicable, T);
19:                            CreateLinkAction(a1,reachable, s2);
20:                         end if
21:                      end if
22:                   end for
23:                end if
24:             end for
25:          end if
26:       end for
27:    end if
28: end while
```

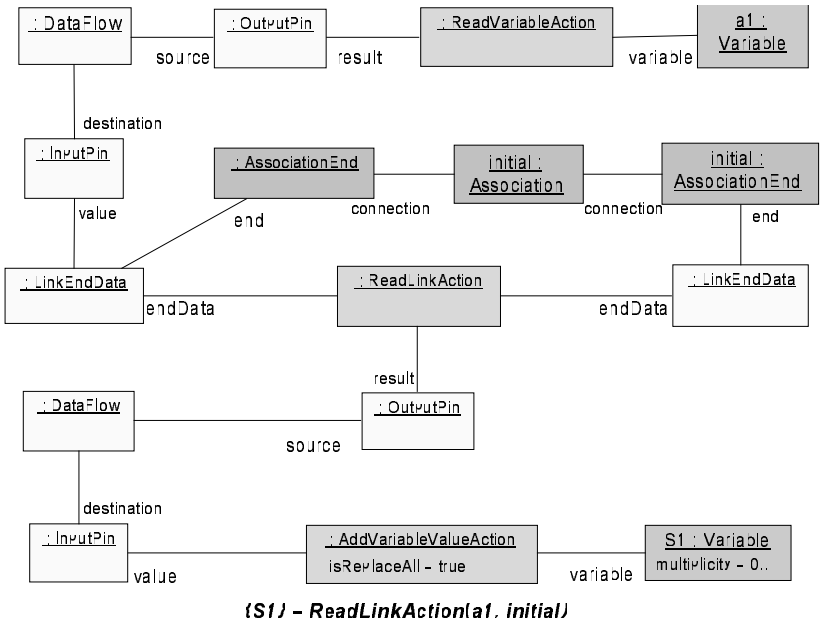

Figure 7: Navigating links

Solution 8 In both cases, the current element of the collection (corresponding to a matching of a certain node in the LHS of the rule) is stored in a variable. If the collection is empty then none of the subactions are executed thus this instantiation can be omitted.

Example 8 In the AS representation (see Fig. 8 for Line 6 in Algorithm 1), we first read the collection variable (ReadVariableAction) *S1* into the input pin of MapAction with a further constraint stating the value contained by the subinput (output) pin should be stored in the variable *s1* (AddVariableValueAction).

A weakness in the standard: Testing the existence of links At the current point, we give explanation for Line 8 in Algorithm 1, which will demonstrate a major weakness in the AS standard.

When all the nodes of the LHS are instantiated by objects, we have to test the existence of links between these objects prescribed by edges in the LHS that have not been visited by navigation (or the non-existence of links in case of negative conditions). For efficiency reasons, such tests should be performed as soon as possible.

Example 9 For instance, in Line 8 of Algorithm 1, we check the negative condition that there must not be any existing reachable link between the automaton object stored in

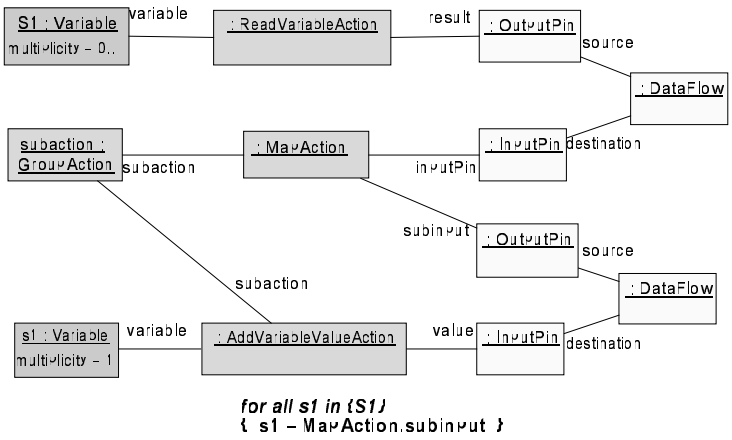

Figure 8: Implementing application modes (*forall*)

variable *a1* and the initial state object stored in *s1* by executing the boolean function testLink(a1, reachable, s1).

Problem 1 The existence of a link of a certain type relating two given objects cannot be tested within AS.

Function testLink can be *used* in UML models with AS expressions, but it cannot be implemented by predefined actions of AS since a user function can be *declared* (with a signature specification) within AS as an action, but they cannot be *defined* (no specification of semantics). Typically, this action is implemented in every UML CASE tool with an action specification language; however, it is not part of the standard, which is very unfortunate.

Solution 9 Thus, the standard should be extended with a TestLinkAction with two LinkEndData (both required to store an associated object on its input pin) as storing the input and a boolean valued output pin.

Without such an extension to the standard, the implementation of the external mathematical function a testLink will be CASE tool dependent and does not fit well to the generality of our approach.

4.3 Manipulating links and objects

Finally, after a successful pattern matching phase, objects and links are to be manipulated according to the difference of the LHS and the RHS of a rule.

Solution 10 Whenever a transformation rule prescribes

- the *addition of an object*, a CreateObjectAction is executed and the created object is stored in a new variable;
- the *deletion of an object*, a DestroyObjectAction is executed on an object retrieved from the corresponding variable;
- the *addition of a link*, a CreateLinkAction is executed to create a link of a certain type between the objects read from the corresponding variables;
- the *deletion of a link*, a DestroyLinkAction is executed to destroy a link of a certain type between the objects read from the corresponding variables.

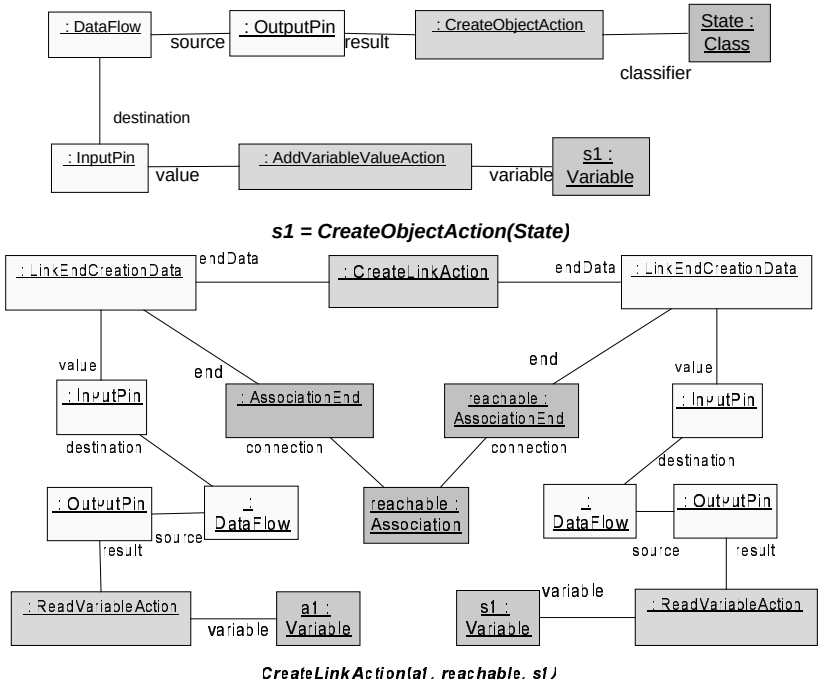

Figure 9: Creating objects and links

Example 10 The AS encoding of link creation in Line 9 of Algorithm 1, and an imaginatory creation of a State object is demonstrated in Fig. 9.

- When a new State object is created (upper part of Fig. 9) by CreateObjectAction it is directly passed to AddVariableAction by a data flow connection to store the new object in variable *s1*.
- When a reachable link is created between objects *a1* (of type Automaton) and *s1* (of type State), a CreateLinkAction is executed where *values* of both LinkEndCreationData are specified by data flows from the corresponding ReadVariableActions, while

the *ends* of both LinkEndCreationData are defined by the related association in the metamodel.

5 Conclusions

In the paper, we presented a method to automatically implement model transformations specified on a very high abstraction level by metamodeling techniques and graph transformation rules by mapping them into UML Action Semantics expressions. The main advantage of our approach is that visual but mathematically precise model transformations can be directly encoded in *the standard* action specification language of the MDA (and UML) environment thus providing a smooth integration of formal specifications and industrial standards. The entire encoding was demonstrated on a small example (reachability analysis of finite automata) which was still rich enough to cover all the basic rules of our encoding.

Our approach conceptually followed the graph pattern matching algorithms of the PROGRES [23] and (especially) FUJABA [15] systems. Instead of global, constraint satisfaction based searches (such as [12]), the matching pattern is searched locally by navigating through links between objects. Meanwhile, from a strict theoretical point of view, global strategies can be more efficient than local searches, practical experiences in the previous graph transformation systems demonstrated that pattern matching by navigation is very efficient in most practical cases, moreover it fits better to the object-oriented nature of UML and AS.

During our encoding, we also discovered a weakness in the Action Semantics (testing the existence of links of a certain type between two objects) standard that should be fixed in order to obtain a fully functional navigation language for actions.

Acknowledgments

The authors are grateful to Albert Zündorf for the fruitful discussion and tool demonstration during the ICGT 2002 conference.

References

[1] D. Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. Ph.D. thesis, University of Kent, Canterbury, 2000.

[2] D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 243–258. Springer-Verlag, Dresden, Germany, 2002.

[3] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, vol. 16(5):pp. 265–275, 2001.

[4] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, vol. 26(3/4):pp. 241–265, 1996.

[5] Gy. Csertán, G. Huszerl, I. Majzik, Zs. Pap, A. Pataricza, and D. Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In J. Richardson, W. Emmerich, and D. Wile (eds.), *Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, pp. 267–270. IEEE Press, Edinburgh, UK, 2002.

[6] J. de Lara and H. Vangheluwe. AToM3: A tool for multi-formalism and metamodelling. In R.-D. Kutsche and H. Weber (eds.), *5th International Conf. FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings*, vol. 2306 of *LNCS*, pp. 174–188. Springer, 2002.

[7] B. Demuth, H. Hussmann, and S. Obermaier. Experiments with XMI based transformations of software models. In *Workshop on Transformations in UML*. 2001.

[8] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.

[9] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: Firs International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 90–105. Springer-Verlag, Barcelona, Spain, 2002.

[10] R. Heckel, J. Küster, and G. Taentzer. Towards automatic translation of UML models into semantic domains. In *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pp. 11–21. Grenoble, France, 2002.

[11] Kennedy-Carter. Executable UML (xuml). `http://www.kc.com/html/xuml.html`.

[12] J. Larrosa and G. Valiente. Graph pattern matching using constraint satisfaction. In H. Ehrig and G. Taentzer (eds.), *GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*. Berlin, Germany, 2000.

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

[14] D. Milicev. Automatic model transformations using extended UML object diagrams in modeling environments. *IEEE Transactions on Software Engineering*, vol. 28(4):pp. 413–431, 2002.

[15] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*. ACM Press, Limerick, Ireland, 2000.

[16] Object Management Group. *Action Semantics for the UML*, 2001. `http://www.omg.org`.

[17] Object Management Group. *Model Driven Architecture — A Technical Perspective*, 2001. `http://www.omg.org`.

[18] M. Peltier, J. Bézivina, and G. Guillaume. MTRANS: A general framework, based on XSLT, for model transformations. In J. Whittle et al. (eds.), *Workshop on Transformations in UML*, pp. 93–97. 2001.

[19] Project Technology. Bridgepoint development suite. `http://www.projtech.com`.

[20] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.

[21] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[22] A. Schürr. Specification of graph translators with triple graph grammars. Tech. rep., RWTH Aachen, Fachgruppe Informatik, Germany, 1994.

[23] A. Schürr, A. J. Winter, and A. Zündorf. *In [8]*, chap. The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.

[24] J. Sprinkle and G. Karsai. Defining a basis for metamodel driven model migration. In *Proceedings of 9th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, Lund, Sweden*. 2002.

[25] D. Varró. Automatic program generation for and by model transformation systems. In H.-J. Kreowski and P. Knirsch (eds.), *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pp. 161–173. Grenoble, France, 2002.

[26] D. Varró. Towards symbolic analysis of visual modelling languages. In P. Bottoni and M. Minas (eds.), *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, vol. 72 of *ENTCS*, pp. 57–70. Elsevier, Barcelona, Spain, 2002.

[27] D. Varró and A. Pataricza. Metamodeling mathematics: A precise and visual framework for describing semantics domains of UML models. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 18–33. Springer-Verlag, Dresden, Germany, 2002.

[28] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, vol. 44(2):pp. 205–227, 2002.

[29] J. Whittle. Transformations and software modeling languages: Automating transformations in UML. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 227–242. Springer-Verlag, Dresden, Germany, 2002.