Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2010) 105–120
www.elsevier.com/locate/entcs

Integrating Textual and Graphical Modelling Languages

Luc Engelen Mark van den Brand

Department of Mathematics and Computer Science
Eindhoven University of Technology (TU/e),
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

Abstract

Graphical diagrams are the main modelling constructs offered by the popular modelling language UML. Because textual representations of models also have their benefits, we investigated the integration of textual and graphical modelling languages, by comparing two approaches. One approach uses grammarware and the other uses modelware. As a case study, we implemented two versions of a textual alternative for Activity Diagrams, which is an example of a *surface language*. This paper describes our surface language, the two approaches, and the two implementations that follow these approaches.

Keywords: Grammarware, modelware, surface language

1 Introduction

Many Eclipse-based modelling formalisms focus on notations that are either mainly textual of mainly graphical. Although tools exist that transform models written in a textual language to representations of those models that can be manipulated and depicted using graphical notations, the construction and manipulation of models written using a combination of both languages is not well facilitated.

The popular modelling language UML offers graphical diagrams for the construction of models. Research has shown, however, that graphical languages are not inherently superior to textual languages [16] and that both types of languages have their benefits. Therefore, we investigate the integration of textual and graphical languages, to be able to exploit the benefits of both types of languages.

One of the problems that arise when using two or more languages to construct one model is that parts of the model written in one language can refer to elements contained in parts written in another language. Transforming a model written in multiple languages to a model written in one language involves introducing correct references between various parts of the model.

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2010.08.035

Existing tools are aimed at converting textual models conforming to grammars into models conforming to metamodels and vice versa [7,3]. These tools can not transform models that consist of parts that conform to grammars as well as parts that conform to metamodels.

We use a textual alternative for activity diagrams, a textual surface language, as a case study and have implemented two versions of this language. One alternative uses tools and techniques related to grammars, and the other uses tools and techniques related to models and metamodels. The approach related to grammars transforms UML models containing fragments of behaviour modelled using our surface language to plain UML models by rewriting the XMI representation of the model provided as input. We used the ASF+SDF Meta-Environment [20] to implement this approach. The approach related to models and metamodels extracts the fragments of surface language, converts them to metamodel based equivalents, transforms these equivalents to Activities, and uses these to replace the fragments in the original model. We used the openArchitectureWare platform [23] to implement this approach.

The remainder of this paper is organized as follows: Section 2 introduces a number of relevant concepts. A specification of the surface language we implemented, and a description of its embedding in the UML and the transformation from surface language to Activities is given in Section 3. The approach based on grammars is described in Section 4, and the approach based on models and metamodels is described in Section 5. A number of other applications involving the integration of textual and graphical languages, and the transformation of models constructed using multiple languages are discussed in Section 6. Section 7 discusses how our work relates to earlier work. We draw conclusions and discuss future work in Section 8.

2 Preliminaries

The surface language we present is a textual alternative for the activity diagrams of the UML. In this section, we give a brief description of Activities and explain what a surface language is.

We use the naming convention used by the OMG in the definition of the UML [12] when discussing concepts of the UML. This means that we use medial capitals for the names of these concepts.

2.1 UML Activities

Activities are one of the concepts offered by the UML to specify behaviour. Some aspects of an Activity can be visualized in an activity diagram. The leftmost part of Figure 1 shows an example of such a diagram.

An Activity is a directed graph, whose nodes and edges are called ActivityNodes and ActivityEdges. There are a number of different ActivityNodes, such as ControlNodes (depicted by diamonds) and Actions (depicted by rounded rectangles), and two types of ActivityEdges, namely ControlFlows and ObjectFlows.

The informal description of the semantics of Activities states that the order in

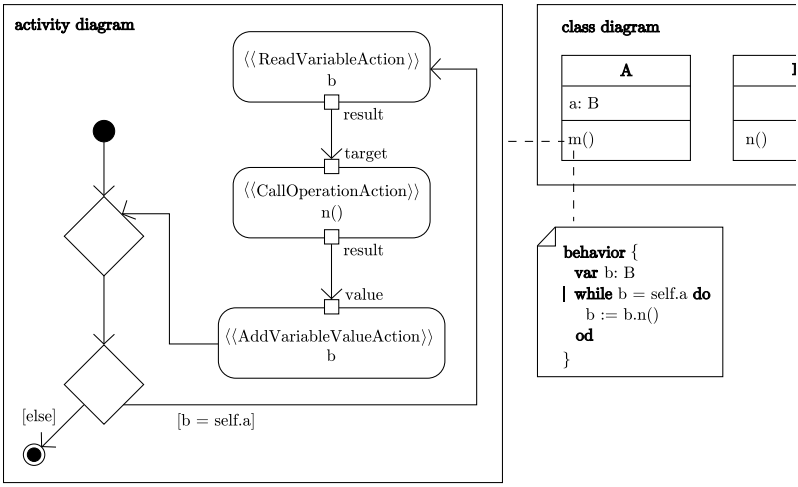

Figure 1. Two representations of the same behaviour

which Actions are executed is based on the flow of tokens. There are two kinds of tokens: control tokens and object tokens. ControlFlows, which are depicted by arrows connecting ActivityNodes, show how control tokens flow from one ActivityNode to the other. ObjectFlows, which are depicted by arrows connecting OutputPins and InputPins, show how object tokens flow from one Action producing an object to another Action that uses this object.

The ObjectFlows in Figure 1 are depicted by the arrows connecting the small rectangles on the borders of the Actions. These small rectangles are the InputPins and OutputPins of those Actions.

2.2 Surface Languages

Every model conforms to a metamodel, which defines the elements that play a role in the model. If a model conforms to a certain metamodel, each element of the model is an instance of an element in that metamodel. The UML defines a number of diagrams, which can be used to depict certain parts of a model. There are diagrams that depict the structure of a model, diagrams that depict the behaviour of parts of the model, etc. These diagrams offer a graphical representation for instances of elements in the metamodel.

A surface language offers an alternative notation for these diagrams. In our case, instead of a graphical representation, a textual representation is given for instances of elements of the metamodel. Other names for surface languages are *surface action languages* or *action languages*.

3 Specification of the Surface Language

To define our surface language, we must specify its syntax, semantics and embedding in the UML. The syntax of the surface language and its embedding in the UML are

described below. The semantics of the language is defined implicitly by describing the transformation from behaviour specified in our surface language to Activities.

3.1 Syntax

The syntax of behaviour modelled in our surface language is as follows:

$$
\begin{aligned}
SLB &::= \text{“behavior”}\ \text{“\{”}\ [MVD]\ MS\ \text{“\}”}\\
MVD &::= \text{“var”}\ VD\ \{\text{“;”}\ VD\}\ \text{“|”}\\
VD &::= VN\ \text{“:”}\ TN\\
MS &::= S\ \{\text{“;”}\ S\},
\end{aligned}
$$

where VN denotes a set of variable names and TN denotes a set of type names. A description of behaviour consists of a sequence of variable declarations and a sequence of statements. A variable declaration consists of a variable name and a type name.

The syntax of statements is as follows:

$$
\begin{aligned}
S ::= &\ \text{“if”}\ SWR\ \text{“then”}\ MS\ \text{“fi”}\\
\mid &\ \text{“if”}\ SWR\ \text{“then”}\ MS\ \text{“else”}\ MS\ \text{“fi”}\\
\mid &\ \text{“while”}\ SWR\ \text{“do”}\ MS\ \text{“od”}\\
\mid &\ \text{“return”}\ SWR\\
\mid &\ SN\ \text{“(”}\ [MSWR]\ \text{“)”}\ \text{“to”}\ SWR\\
\mid &\ SWR\ \text{“.”}\ ON\ \text{“(”}\ [MSWR]\ \text{“)”}\\
\mid &\ SWR\ \text{“.”}\ SFN\ [\text{“[”}\ \mathbb{N}\ \text{“]”}]\ \text{“:=”}\ SWR\\
\mid &\ VN\ [\text{“[”}\ \mathbb{N}\ \text{“]”}]\ \text{“:=”}\ SWR,
\end{aligned}
$$

where SN denotes a set of signal names, ON denotes a set of operation names, SFN denotes a set of structural feature names and $\mathbb{N}$ denotes the set of natural numbers.

The syntax of statements with results is as follows:

$$
\begin{aligned}
MSWR ::= &\ SWR\ \{\text{“,”}\ SWR\}\\
SWR ::= &\ \text{“create”}\ \text{“(”}\ CN\ \text{“)”}\\
\mid &\ \text{“self”}\\
\mid &\ VN\\
\mid &\ SWR\ \text{“.”}\ SFN\\
\mid &\ SWR\ \text{“.”}\ ON\ \text{“(”}\ [MSWR]\ \text{“)”},
\end{aligned}
$$

where CN denotes a set of class names, VN denotes a set of variable names, ON denotes a set of operation names and SFN denotes a set of structural feature names. Operation calls are listed both as statements and as statements with results, because both types of operation calls exist.

The note below the class diagram in Figure 1 shows an example of behaviour modelled using our surface language. The behaviour is equivalent to the behaviour represented by the activity diagram on the left of the figure.

```
<packagedElement xmi:type="uml:Class" name="C">
  <ownedBehavior xmi:type="uml:OpaqueBehavior" name="b">
    <language>SL</language>
    <body>return self</body>
  </ownedBehavior>
</packagedElement>
```

Figure 2. Embedding in the UML of behaviour modelled using a language called 'SL'

3.2 Embedding in the UML

We use a concept of the UML called OpaqueBehavior to embed our surface language in the UML. Figure 2 shows a fragment of an XMI representation of a UML model that contains an instance of OpaqueBehavior.

OpaqueBehavior uses a list of text fragments and a list of language names to specify behaviour. The first list specifies the behaviour in one or more textual languages and the second list specifies which languages are used in the first list. OpaqueBehavior can be used to specify behaviour using, for instance, fragments of Java code or natural language. In our case, the first list contains a specification of behaviour using our surface language and the second list indicates that we use this surface language.

We transform a UML model containing behaviour modelled using our surface language to a UML model without such behaviour, by replacing all these occurrences of surface language embedded in OpaqueBehavior by equivalent Activities.

3.3 Transformation

As described in Section 3.1, behaviour specified using our surface language consists of two parts: a sequence of variable declarations and a sequence of statements. The process of transforming behaviour modelled using a surface language to an Activity can be divided into two steps:

(i) The variable declarations are translated to UML Variables.

(ii) The sequence of statements is translated to an equivalent group of ActivityNodes and ActivityEdges.

Translating variable declarations to UML Variables is a trivial step, which we will not discuss. An informal description of the second step is given below.

3.3.1 Transformation Function

We describe the transformation of sequences of statements to equivalent fragments of UML Activities by means of the transformation function T_B. The function T_B uses the auxiliary transformation functions T_{MS}, T_S and T_{SWR}. Figure 3 gives a schematic representation of the transformations performed by these functions.

The clouds and the dashed arrows in the figure indicate how the fragments are joined together to create an Activity. Each cloud in a fragment is replaced by

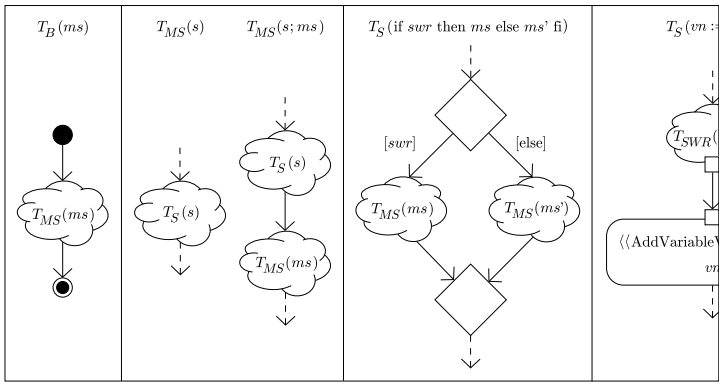

Figure 3. The transformations performed by the functions T_B, T_{MS}, T_S and T_{SWR}

another fragment of an Activity. An incoming dashed ActivityEdge shows how a fragment is connected to an outgoing ActivityEdge of the containing fragment; an outgoing dashed ActivityEdge shows how a fragment is connected to an incoming ActivityEdge of the containing fragment.

The function T_B creates a group of ActivityNodes and ActivityEdges that is equivalent to the sequence of statements provided as input, and connects this group with an InitialNode and an ActivityFinalNode using two ControlFlows.

The function T_{MS} creates an equivalent group of ActivityNodes and ActivityEdges for each of the statements in the sequence provided as input, and connects these groups using ControlFlows.

The function T_S creates a group of ActivityNodes and ActivityEdges that is equivalent to the statement provided as input. Statements with or without results that are part of the statement provided as input are also translated to equivalent groups of ActivityNodes and ActivityEdges. These groups are connected to the first group using ControlFlows, for statements, or ObjectFlows, for statements with results.

The function T_{SWR} creates a group of ActivityNodes and ActivityEdges that is equivalent to the statement with result provided as input. Statements with results that are part of this statement are also translated to equivalent groups of ActivityNodes and ActivityEdges. These groups are connected to the first group using ObjectFlows.

4 Grammarware

In this section, we describe the implementation of our surface language that uses a tool for text-to-text transformations. Tools for text-to-text transformations are often referred to as *grammarware*. We start by describing our approach in Section 4.1. Section 4.2 describes the tools we used for the implementation and some important aspects of the implementation.

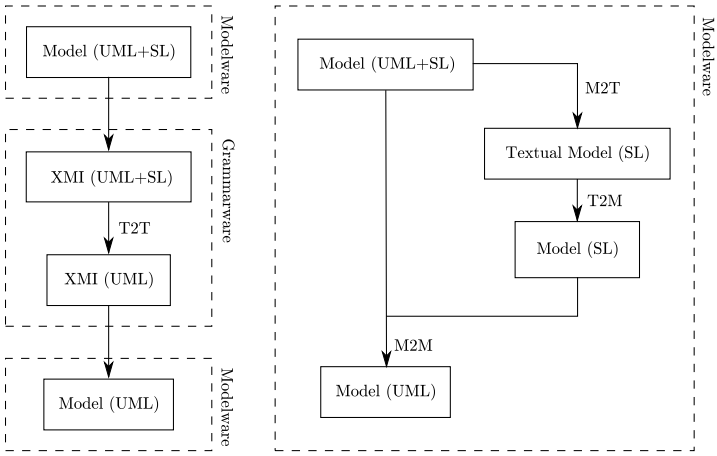

Figure 4. Two ways of incorporating textual languages in the UML

4.1 Approach

The leftmost part of Figure 4 gives a schematic overview of the transformation process when performed using a text-to-text (T2T) transformation.

The goal of this process is to transform a UML model containing behaviour specified using a surface language to a plain UML model. In the approach using grammarware, we transform models containing fragments of surface language to plain UML models by transforming the XMI [11] representations of those models. This transformation from one textual representation to the other consists of two steps:

(i) A mapping from names occurring in the model to XMI identifiers is made, by traversing the parse tree of the XMI representation of the original model, and storing each name and the corresponding identifier in a table.

(ii) The transformation described in Section 3.3 is performed, by translating fragments of surface language to XMI representations of equivalent Activities.

The first step of the transformation makes it possible to retrieve the identifier of an element in the second step. Each element in the XMI representation of a UML model has a unique identifier. Actions that refer to other elements, such as AddVariableValueActions and CreateObjectActions, refer to these other elements using their identifiers. An AddVariableValueAction refers to a Variable using the identifier of that Variable; a CreateObjectAction refers to a Classifier using the identifier of that Classifier.

Grammarware has been a subject of research for quite some time. An advantage of using grammarware to perform this transformation is the ease of use provided by the maturity of the tools and their documentation.

A disadvantage of transforming models using a text-to-text transformation is that the models have to be exported to a textual format. Having to deal with the textual

representation of a model lowers the level of abstraction of the transformation. In our case, for instance, the transformation deals with concepts of the XMI language, at a low level of abstraction, instead of concepts of the UML, at a higher level of abstraction.

4.2 Implementation

We implemented the transformation described in Section 3.3 following the approach described in Section 4.1 in the language ASF+SDF [21], using an IDE for that language, called the Meta-Environment. We give a short description of this language and discuss some of the details of our implementation below.

4.2.1 ASF+SDF and the Meta-Environment

The language ASF+SDF is a combination of the two formalisms ASF and SDF. SDF stands for *Syntax Definition Formalism*. It is a formalism for the definition of the syntax of context-free languages. ASF stands for *Algebraic Specification Formalism*. It is a formalism for the definition of conditional rewrite rules. Given a syntax definition in SDF of the source and target language, ASF can be used to define a transformation from the source language to the target language.

Context-free languages are closed under union and, as a result of this, the SDF definitions of two languages can be combined to form the definition of a new context-free language, without altering the existing definitions.

Using ASF in combination with SDF to implement transformations guarantees syntax safety. A transformation is syntax safe if it only accepts input that adheres to the syntax definition of the input language and it always produces output that adheres to the syntax definition of the output language.

Both SDF and ASF specifications can be exported by the Meta-Environment. The exported SDF specification can then be used by command line tools to produce parse trees and transform parse trees to text. The exported ASF specification can be compiled to a fast command line tool suited for the transformation of such parse trees.

4.2.2 Implementation Details

An advantage of using SDF to define the syntax of our surface language is that it enabled us to combine this definition with an existing syntax definition of the XMI format, without any alterations to the definitions. This is due to the previously mentioned fact that context-free languages are closed under union.

Because transformations implemented in ASF are syntax safe, the transformation from UML models containing fragments of surface language to plain UML models only produces results that adhere to the definition of XMI.

A disadvantage of the current implementation is that it can only parse one variant of XMI. Most tools that import or export files in the XMI format use their own interpretation of the format. These vendor specific interpretations are often incompatible with other interpretations. Because of this, our implementation is limited

```
$VariableName := $ReadVariableAction,
variableExists($Context, $VariableName) == true,
$Val := getVariableId($Context, $VariableName),
<$Id1, $Context1> := newId($Context),
<$Id2, $Context2> := newId($Context1),
$ObjectContent* :=
<node xmi:type="uml:ReadVariableAction" xmi:id=$Id1 variable=$Val>
  <result xmi:id=$Id2 />
</node>
====>
statementWithResult2Action($ReadVariableAction, $Context) =
  <$ObjectContent*, $Id2, $Id1, $Context2>
```

Figure 5. An ASF equation that creates a ReadVariableAction

```
sorts
  ReadVariableAction
context-free syntax
  VariableName -> ReadVariableAction
variables
  "$ReadVariableAction"[0-9]* -> ReadVariableAction
```

Figure 6. An SDF definition that defines the statement representing a ReadVariableAction

to XMI files produced by the UML2 plug-in of Eclipse, since it can only read and produce models that adhere to the interpretation of XMI of that plug-in.

A solution for this problem would be to introduce an intermediate language that serves as the starting point of a number of transformations to variants of XMI. We could then transform a model containing fragments of surface language to this intermediate language and subsequently from this intermediate language to a number of variants of XMI.

The limited portability is another disadvantage of using the Meta-Environment for the implementation of our approach, since it is currently only available for the Unix family of operating systems.

Figure 5 shows a part of the implementation in ASF of the transformation from behaviour modelled using our surface language to Activities. All variable names in this figure start with a dollar sign. The figure shows that a table mapping names to identifiers, denoted by the variable `$Context`, is used both to retrieve the identifier that corresponds to a given name as well as create fresh identifiers. The figure shows that every ReadVariableAction encountered in a fragment of surface language is replaced by the XMI in lines 7 to 9. Figure 6 shows the part of the SDF definition that defines the syntax of the surface language statement representing a ReadVariableAction and declares the corresponding variables. Line 4 of this definition defines that a ReadVariableAction is denoted by the name of a variable, as is specified in Section 3.1.

5 Modelware

This section describes the implementation of our surface language using tools for model-to-text, text-to-model and model-to-model transformations. Tools that can perform transformations related to models are often referred to as *modelware*. Section 5.1 describes our approach. Section 5.2 describes the tools we used for the implementation and some important aspects of the implementation.

5.1 Approach

The rightmost part of Figure 4 gives a schematic overview of the approach using model-to-text (M2T), text-to-model (T2M) and model-to-model (M2M) transformations within a UML modelling tool.

The process of using modelware to transform a UML model containing fragments of surface language to a plain UML model can be divided into the following steps:

(i) The fragments of surface language are extracted from the original model.

(ii) The extracted fragments are parsed and converted to a format usable by tools for model-to-model transformations.

(iii) The extracted and converted fragments of surface language are translated to equivalent Activities, as described in Section 3.3.

(iv) The fragments of surface language in the original model are replaced by the Activities created in the previous step.

An advantage of this approach is that all transformations can be performed from within one and the same modelling environment. In contrast to the approach described in Section 4.1, no models have to be imported or exported during the transformation process.

5.2 Implementation

We used three tools for model transformation from the openArchitectureWare platform to implement the transformation described in Section 3.3 following the approach described in Section 5.1. We describe these tools and the implementation below.

5.2.1 Xpand, Xtend, Xtext and openArchitectureWare

The openArchitectureWare platform offers a number of tools related to model transformation: Xpand is used for model-to-text transformations, Xtext [3] is used for text-to-model transformations and Xtend is used for model-to-model transformations. Xpand and Xtend are based on the same type system and expression language. The type system offers simple types, such as *string*, *Boolean* and *integer*, collection types, such as *list* and *set*, and the possibility to import metamodels. The expression language offers a number of basic constructs that can be used to create expressions, such as *literals*, *operators*, *quantifiers* and *switch expressions*.

Xpand is a template-based language that generates text files given a model. An

```
behavior b C {
  return self
}
```

Figure 7. An extracted fragment of surface language

Xpand template takes a metaclass and a list of parameters as input and produces output by executing a list of statements. There are a number of different type of statements, including one that saves the output generated by its statements to a file and one that triggers the execution of another template.

Xtext is a tool that parses text and converts it to an equivalent model, given a grammar describing the syntax of the input. Xtext uses ANTLR [15] to generate a parser that parses the textual representations of models. An Xtext specification consists of rules that define both a metamodel and a mapping from concrete syntax to this metamodel. Given a specification of a textual representation, Xtext also generates an editor that provides features such as syntax highlighting and code completion.

Xtend is a functional language for model transformation. It adds *extensions* to the basic expression language, which take a number of parameters as input and return the result of an expression.

5.2.2 Implementation Details

We use Xpand to extract fragments of surface language from models by traversing these models. For each instance of OpaqueBehavior in a model, the string describing its behaviour is stored in a text file, including the name of the OpaqueBehavior and the name of the Class it is contained in. Figure 7 shows the fragment of surface language extracted from the OpaqueBehavior of Figure 2.

We use Xtext to parse and convert the extracted fragments of surface language to a format that is readable by Xtend. Because Xtext uses ANTLR, the class of textual representations that can be parsed is restricted to those that can be described by an $LL(k)$ grammar. A disadvantage of using Xtext is that we had to modify our grammar for this reason.

One of the advantages of using the tools offered by the openArchitectureWare platform is their portability. The platform is a collection of plug-ins for Eclipse, and both Eclipse and these plug-ins are available on a number of different operating systems.

Figure 8 shows a part of the transformation implemented in Xtend from behaviour modelled using our surface language to Activities. The figure shows that a new ReadVariableAction, an OutputPin and an ObjectFlow are created, by defining local variables using *let expressions*. These expressions are followed by a *chain expression*, which denotes the sequential evaluation of the expressions connected by the "`->`" symbols. The last two of these expressions use the ObjectFlow to connect the OutputPin of the ReadVariableAction to the InputPin of another Action.

```
Void addReadVariableAction(
   uml::Activity a, uml::Package p, surfacelanguage::Variable v,
   uml::InputPin ip
) :
   let act = new uml::ReadVariableAction :
   let op = new uml::OutputPin :
   let of = new uml::ObjectFlow :
   a.node.add(act)
-> a.edge.add(of)
-> act.setResult(op)
-> act.setVariable(v.createVariable(p))
-> of.setSource(op)
-> of.setTarget(ip)
;
```

Figure 8. An Xtend extension that adds a ReadVariableAction to an Activity

6 Other Applications of our Approach

Our approach is not only suitable for the embedding of our textual surface language in Activities. The concept of OpaqueBehavior described in Section 3.2 can, for instance, also be used to embed textual languages describing behaviour in other parts of the UML. Similar concepts, like OpaqueExpression and OpaqueAction, can be used to embed textual languages for other purposes than describing behaviour. It is possible, for instance, to use a subset of Java as an expression language for UML StateMachines.

Thus far, we described how UML models combined with our surface language can be transformed to equivalent UML models. The result of the transformation described in Section 3.3, however, is only defined if the names used in the fragments of surface language of an input model correspond with elements that exist in the rest of the model. To check whether models meet this condition, we have implemented another version of our transformation, which performs a simple form of checking. This transformation takes a UML model containing fragments of surface language as input and transforms this into a list of error messages. The transformation traverses the model and the fragments of surface language, and checks whether the names used in the statements of the surface language correspond to elements that exist in other parts of the model. If the behaviour shown in the note in Figure 1 would refer to an attribute $self.b$, for instance, the transformation would produce a message stating that class A does not have an attribute named b.

7 Related Work

We chose to design and implement a new surface language, instead of implementing a design proposed by others. Section 7.1 describes two existing proposals for surface languages and indicates why we decided not to implement either of them.

There are many alternatives for the languages we used to implement our surface

language. Section 7.2 lists a number of alternatives for the grammarware we used and Section 7.3 lists a number of alternatives for the modelware we used.

Section 7.4 describes another approach for integrating textual and graphical modelling languages.

7.1 Surface Languages

Dinh-Trong, Ghosh and France propose an Action Language based on the syntax of Java [2]. We decided not to implement their Action Language because their definition of the language contains a number of primitive types and Java constructs whose relation to the UML is not specified. Other important features of their language are that parameters that serve as input or output of an Activity and attributes with multiplicity greater than one are not taken into account.

Haustein and Pleumann propose a surface language that is an extension of the OCL [4,10]. They embed OCL expressions in their language by adding an Action to the UML that evaluates an OCL expression and returns the resulting value. We took a different approach, because we wanted to design and implement a simple alternative for activity diagrams that did not rely on or incorporate other languages. Incorporating an expression language like the OCL in our language would introduce a large number of language constructs that have no relation to our primary interest, which is the specification of behaviour.

7.2 Grammarware

SDF is based on SGLR, a scannerless generalized LR parser [22]. As an alternative to using SDF, SGLR can be used to parse textual representations of models. Since SGLR can parse arbitrary languages with a context-free syntax and context-free languages are closed under union, multiple syntax definitions can be combined into one without any modifications to the original syntax definitions, as is the case for SDF.

Other common tools used for parsing, such as ANTLR, JavaCC [19] and YACC [5], can also be used to parse textual representations of models. They pose more restrictions on the grammars used for the description of the textual representations, however, since the grammars need to be of the *LALR* or the *LL* class.

After parsing the textual representations of models, the resulting parse trees have to be transformed. Besides using special purpose transformation tools, generic programming languages can be used to manipulate the parse trees. The source transformation language TXL [1] is an example of a special purpose language. Paige and Radjenovic [14], and Liang and Dingel [9] have experimented with TXL in the context of model transformation. Although their research also deals with using grammarware for transformations related to models, it differs from ours because it does not focus on the integration of text-based and metamodel-based languages.

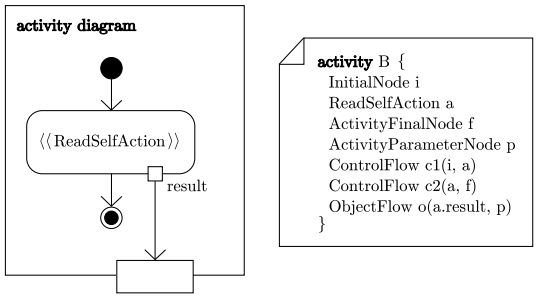

Figure 9. An activity diagram and a straightforward textual equivalent

7.3 Modelware

TCS [7] is an alternative for Xtext. It is suited for both text-to-model and model-to-text transformations, and uses one specification to define the transformations in both directions. In the case of TCS, the main constructs are called templates. These templates are similar to the rules of Xtext; each template specifies the textual representation of an instance of an element of the metamodel.

Figure 9 illustrates how the resulting languages differ from our surface language in case a straightforward mapping, like those offered by Xtext and TCS, is used without additional transformations. The behaviour shown in Figure 9 is equivalent to the behaviour shown in Figure 7. The description of behaviour shown in Figure 9 is much more wordy than that of Figure 7, even for such a trivial example.

There are many languages for model transformation, including QVT [13], ATL [6] and Epsilon [8]. Since our approach does not rely on any specific properties of Xtend, each of these transformation languages can replace Xtend in our implementation.

7.4 Embedding Textual Modelling into Graphical Modelling

Scheidgen's approach for integrating textual and graphical modelling languages [18] is based on the fact that Eclipse uses the *Model View Controller* pattern [17]. A mapping from textual notation to metamodel elements is used to generate a model from a textual representation of that model and vice versa.

This custom textual notation and the graphical notations provided by Eclipse provide independent *Views* for the same *Model*. The *Controller* is used to modify the underlying model, without interfering directly with the other views.

The embedded text editor contained in the implementation of this approach offers syntax highlighting and code completion. Similar to Xtext and TCS, the language describing mappings from textual notation to metamodel elements offers only straightforward mappings.

8 Conclusions and Future Work

We investigated two approaches for the integration of textual and graphical modelling languages, by implementing a textual surface language as an alternative for activity diagrams. We described this surface language, the two approaches, the implementations that follow these approaches, and a number of related applications.

The approach using grammarware transforms models containing fragments of surface language to plain models by rewriting the XMI representations of these models. A downside of this approach is that dealing with the XMI representation of models lowers the level of abstraction of these transformations. The current implementation can only parse one variant of XMI, but a future extension that introduces an intermediate language poses a solution for this shortcoming.

The approach using modelware extracts fragments of surface language from a model, converts these fragments to a representation based on metamodels, transforms them to equivalent Activities, and replaces the original fragments with the equivalent Activities. An advantage of this approach is that all of these operations can be performed from within one modelling environment. A disadvantage of the current implementation of this approach is that the available tools pose more restrictions on the grammar of the language we embed, in comparison to the approach using grammarware. Investigating the use of more advanced parsing technology as a basis for these tools is another promising direction for future research.

The approaches we presented are not limited to the transformation of models to equivalent models. We also implemented a transformation that transforms models containing fragments of surface language into a list of error messages, thus providing a simple form of checking.

Our approaches provide advantages over the approaches described in Section 7, because they both offer a more complex mapping from textual representations to metamodel elements, which can be used to obtain simpler textual representations. The fact that the implementation using grammarware poses less restrictions on the syntax of the textual language is also an advantage over these approaches.

References

[1] Cordy, J. R., *The TXL source transformation language*, Science of Computer Programming **61** (2006), pp. 190–210.

[2] Dinh-Trong, T., S. Ghosh and R. France, *JAL: Java like Action Language, version 1.1* (2006), Department of Computer Science, Colorado State University.

[3] Efftinge, S. and M. Völter, *oAW xText: A framework for textual DSLs*, in: *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[4] Haustein, S. and J. Pleumann, *OCL as Expression Language in an Action Semantics Surface Language*, in: O. Patrascoiu, editor, *OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal* (2004), pp. 99–113.

[5] Johnson, S. C., *Yacc: Yet another compiler compiler*, in: *UNIX Programmer's Manual*, Holt, Rinehart, and Winston, 1979 pp. 353–387.

[6] Jouault, F., F. Allilaire, J. Bézivin, I. Kurtev and P. Valduriez, *ATL: a QVT-like transformation language*, in: *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (2006), pp. 719–720.

[7] Jouault, F., J. Bézivin and I. Kurtev, *TCS: a DSL for the specification of textual concrete syntaxes in model engineering*, in: *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering* (2006), pp. 249–254.

[8] Kolovos, D. S., R. F. Paige and F. Polack, *The epsilon transformation language*, in: A. Vallecillo, J. Gray and A. Pierantonio, editors, *ICMT*, Lecture Notes in Computer Science **5063** (2008), pp. 46–60.

[9] Liang, H. and J. Dingel, *A Practical Evaluation of Using TXL for Model Transformation*, in: *SLE '08: Proceedings of the 1st International Conference on Software Language Engineering*, 2008.

[10] Object Management Group, *Object Constraint Language, OMG Available Specification, Version 2.0* (2006).

[11] Object Management Group, *MOF 2.0/XMI Mapping, Version 2.1.1* (2007).

[12] Object Management Group, *Unified Modeling Language: Superstructure 2.1.2* (2007).

[13] Object Management Group, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0* (2008).

[14] Paige, R. and A. Radjenovic, *Towards Model Transformation with TXL*, in: *Proceedings of the workshop on Metamodelling for MDA*, 2003.

[15] Parr, T. J. and R. W. Quong, *ANTLR: A predicated-LL(k) parser generator*, Software — Practice and Experience **25** (1995), pp. 789–810.

[16] Petre, M., *Why looking isn't always seeing: readership skills and graphical programming*, Communications of the ACM **38** (1995), pp. 33–44.

[17] Reenskaug, T., *Models - views - controllers*, Technical report, Xerox Parc (1979).

[18] Scheidgen, M., *Textual Modelling Embedded into Graphical Modelling*, in: I. Schieferdecker and A. Hartman, editors, *Proceedings of the 4th European Conference on Model Driven Architecture* (2008), pp. 153–168.

[19] Sriram Sankar and Sreenivasa Viswanadha and Rob Duncan, *JavaCC: The Java Compiler Compiler* (2008), https://javacc.dev.java.net/.

[20] van den Brand, M., J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser and J. Visser, *The ASF+SDF Meta-Environment: a Component-Based Language Development Environment*, in: *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS (2001), pp. 365–370.

[21] van Deursen, A., *An overview of ASF+SDF*, in: A. van Deursen, J. Heering and P. Klint, editors, *Language Prototyping: An Algebraic Specification Approach*, World Scientific Publishing Co., 1996 pp. 1–30.

[22] Visser, E., "Syntax Definition for Language Prototyping," Ph.D. thesis, University of Amsterdam (1997).

[23] Völter, M., *openArchitectureWare 4.2 - the flexible open source tool platform for model-driven software development* (2007).