MCC: A Model Transformation Environment

Anneke Kleppe

University Twente, Netherlands
a.kleppe@utwente.nl

Abstract. In the model driven software development process, software is built by constructing one or more models and transforming these into other models. In turn these output models may be transformed into another set of models until finally the output consists of program code that can be executed. Ultimately, software is developed by triggering an intricate network of transformation executions.

An open issue in this process is how to combine different transformation tools in a flexible and reliable manner in order to produce the required output. This paper presents a model transformation environment in which new transformation tools can be plugged in and used together with other available transformation tools. We describe how transformations can be composed. Furthermore, in the cause of answering the question where and how transformations can be successfully applied, we created a language-based taxonomy of model transformation applications.

Keywords: MDA, QVT, model driven development, model transformation, transformation taxonomy.

1 Introduction

Model Driven Architecture (MDA) [1, 2, 3] and Model Driven Engineering (MDE) [4] propose a software development process in which the key notions are models and model transformations. In this process, software is build by constructing one or more models, and transforming these into other models. The common view on this process is that the input models are platform independent and the output models are platform specific, and that the platform specific models can be easily transformed into a format that is executable. In other words, the model driven process is commonly viewed as a code generation process.

There is also a more generic view on model driven development [1, 5, 6], in which the difference between platform independent and platform specific is not dominant. The key to this more generic view is that the software development process is implemented by an intricate network of transformation executions, combined in various ways. This makes model driven development much more open and flexible.

For example, in figure 1 at the start there are two models, one that describes the functionality of the system (M_1) and one that describes the security aspects of the system (M_2). Because we require code that has a certain package structure, consisting of interfaces for each class in the main package and a subpackage containing the implementations, the first transformation we apply is one that changes M_1 into the

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 173 – 187, 2006.

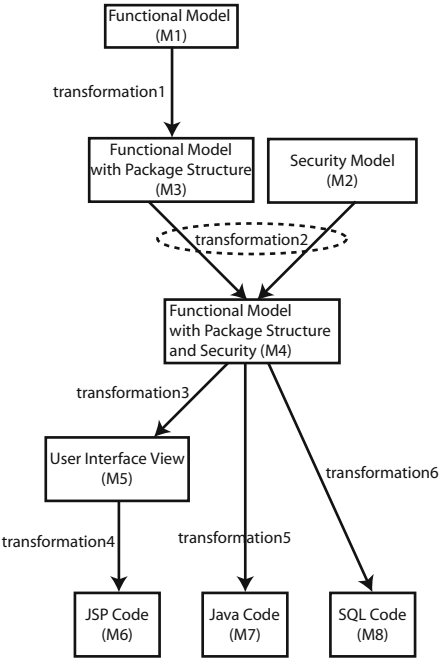

Fig. 1. Example of a combination of transformations

required structure. The resulting model is M_3. The second transformation merges M_3 with M_2, thus creating a model M_4 that has the right package structure and the required security aspects. Next we generate a model M_5 that contains all classes from M_4 that are directly visible to a certain actor named in the use cases in M_1. From this model we generate a platform specific model of the user interface (M_6). Meanwhile, we take M_4 as input to a transformation that generates a database model for the system (M_7), and we use M_4 again as input to a transformation to generates the middle tier of our system (M_8). Even in this fairly simply example, we can recognise six separate transformations.

In this paper we describe an open environment for model transformations in which users may combine the available tools that implement transformations, at will and apply them to models in various languages. Transformation tools may be added or removed, and are thereby available (or not) for composition. Language definitions may be added or removed thus enabling/disabling transformation of certain categories of models. The inclusion of separate language definitions also enables us to formalise and check the types of transformation tools, and the compositions of transformations that are allowed in the environment. We are working towards a model of MDA that makes the input/output relationship of transformations more explicit, and doing so makes transformation scripting look like expressions in functional languages.

We will often use a comparison with compiler technology to explain our ideas, because this comparison helps to illuminate the similarities and differences between traditional compilers and transformers, which are sometimes called *model compilers*.

Please note however, that not all knowledge of compiler construction can be transposed onto the field of model transformation directly. This is due to the fact that the languages in which the models are written are often visual and therefore multidimensional. (See [7]). Another difference is that transformers must be able to handle models in multiple languages. Whereas compilers work with multiple representations of the same program, from parse tree through several stages of abstract syntax tree, transformers work with multiple representations of multiple programs or models.

The paper is structured as follows. Section 2 explains why we have set out to implement an open model driven development environment. Section 3 gives a linguistically based taxonomy of transformations that is used in Section 4, which describes the formalisation of the units that are recognised in our environment. In section 5 the implementation of the environment is outlined. Section 6 contains references to related work and Section 7 concludes the paper with a short summary.

2 Rationale

This section explains the reasons for our approach to transformation composition. Key is the difference between internal and external composition. The environment that we describe in this paper is focused on external composition.

2.1 Internal Versus External Composition of Transformations

There are various approaches for model transformation that offer forms of compositionality, either based on sceduling, reuse, or logical composition of transformation rules. (See [6] for an overview.) For instance, the upcoming QVT standard [8] specifies a language in which one is able to express transformation definitions that consist of a number of mapping rules. The mapping rules may be combined by *calling*, or by using the *refines or extends* mechanisms.

We call this the *internal* composition of transformations, whereas the combination of transformation tools is called the *external* composition of transformations. The latter must tackle tool interoperability as well as the logical composition of transformation rules. In the following, the set of transformation rules that is implemented by a single execution of a single transformation tool will be called a *transformation definition*.

A special concern with interoperability of transformation tools is that not all transformation tools are ready to execute any transformation definition. Some are what we call *specialized transformation* tools, in which the transformation definition is hard-coded, in contrast to the *general transformation* tools, which are able to execute any transformation definition written in a given transformation language.

We focus on external composition of transformations, because it offers the user more flexibility, such as enabling the user to combine transformation tools from different sources. For example, open source transformation tools could be combined with vendor specific tools, and specialized transformation tools could be combined with general transformation tools.

2.2 Towards an Open Model Driven Development Environment

In our view there is ample reason for creating tool support for external composition of transformations. The user of transformation tools could very much benefit from a tool chain of transformation engines, each executing small parts of the transformation execution network. This gives the user full control over the process, thus enabling her or him to be most productive.

An open environment for model driven development should offer multiple transformation tools, multiple transformation definitions in various transformation languages, and multiple tools for other model-related services, such as model creation, or even model checking tools like SPIN [9]. The environment should take care of the concrete interoperability between the tools, and it must provide a means to specify the network of transformation executions that is necessary to produce the required outcome. To prove the feasibility of this approach we have build an open tool environment for transformation execution, which will be described in section 5.

3 Taxonomy of Model Transformation Applications

In this section we present a taxonomy of model transformations based on a linguistic approach. This taxonomy is needed for the formalisation of transformation composition in section 4. The transformations are categorised according to the part of its source and target language definition it addresses. In order to clearly define this taxonomy we first need to formalise our notion of language.

3.1 Language Definitions

Because a model transformation always relates the language of its source model with the language of its target model, one has to be aware of the structure of the definition of these languages in order to understand the different applications of model transformations. The formalisation of language given by Chen e.a. in [10] is a simple and elegant one. They define a language to be a 5-tuple $L = < A, C, S, M_S, M_C >$ consisting of abstract syntax (A), concrete syntax (C), syntax mapping (M_C), semantic domain (S), and semantic mapping (M_S).

However, for our purposes this formalism is too simple. We need to take into account languages that have multiple concrete syntaxes. For instance, one could argue that the visual diagrams of an UML model and the textual XMI format of that model are representations of the same abstract syntax graph[1] in two different concrete syntaxes. (The latter is called the serialization syntax in [11].) Another example is OCL, for which we have defined a second concrete syntax that resembles SQL [12]. Therefore, we extend the given formalism into the following.

1 Instead of using the more common term abstract syntax tree, we use the term abstract syntax graph to stress the fact that such a representation can be made for also languages that are context-free or type 0 in the Chomsky hierarchy.

Definition 1: *(Language) A language is a 5-tuple* $L = < A, SC, S, M_S, SM_C >$ *consisting of an abstract syntax (A), a set of concrete syntaxes (SC), a set of syntax mappings* (SM_C)*, a semantic domain (S), and a semantic mapping* (M_S)*. For each element* C_n *in SC, there is an element* M_{Cn} *in* SM_C*, which is a mapping between* C_n *and the abstract syntax A.*

Most of the times the mapping between a concrete syntax and the abstract syntax will be bi-directional, but this is not necessarily so. Sometimes a concrete syntax is used only to visualize a model, not to edit it or create it. The syntax mapping may be defined in either way; from abstract to concrete syntax, or from concrete to abstract syntax, or bi-directional. In section 4.1 we will look into this in more detail.

Another observation that needs to be made is that the semantics of some languages are not defined by giving a direct mapping of the abstract syntax to the semantic domain, instead they are defined by mapping the abstract syntax to the abstract syntax of another language of which the semantics are known. This type of semantics is known as translational semantics. We formalise this as follows.

Definition 2: *(Translational semantic mapping) A translational semantic mapping for language* L_i *with the use of language* L_j *is a semantic mapping* $TransM_{Si} = M_{Aij} \circ M_{sj}$*, where* M_{Aij} *is a mapping of the abstract syntax* A_i *of* L_i *to the abstract syntax* A_j *of* L_j*.*

3.2 Types of Transformations

In this section we present our taxonomy of transformations based on the formalisation of language in the previous section. An overview of the various types of transformations can be found in table 1, an overview of the relation between the transformation types and elements of the source and target language definition can be found in figure 2. Note that although the arrows in the figure indicate a bi-directionality, not all transformations need to be defined bi-directionally. The arrows indicate that transformations in both directions are possible.

Table 1. A taxonomy of transformations

Name	Category	Maps .. to ..
Syntax transformation	*Intra-language*	A_i -> C_i *and/or* C_i -> A_i
Semantic definition	*Intra-language*	A_i -> S_i
Refactoring	*Intra-model*	A_i -> A_i
View transformation	*Intra-model*	A_i -> A_i
Structure transformation	*Inter-model*	A_{i*} -> A_{j*}
Stream-based transformation	*Inter-model*	C_i -> C_j
Hybrid syntax transformation	*Inter-model*	A_i -> C_j

Intra-language transformations. The first category of transformations is formed by the *intra-language transformations*. Transformations in this category are used to

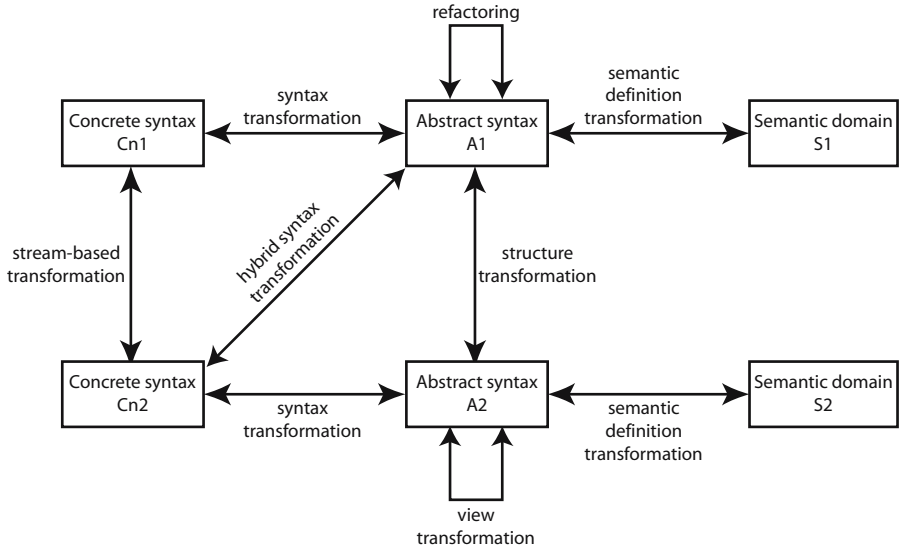

Fig. 2. The relation between transformations and language definition

define a language. They either specify one of the syntax mappings (M_{Cn}), or the semantic mapping (M_S). A transformation that implements a syntax mapping is called *syntax transformation*. An example of a syntax transformation is the MOF to text standard [13]. A transformation that implements a semantic mapping is called *semantic definition transformation*[2]. An example can be found in [14]. Note that in the case of a translational semantic mapping, either or both of the constituting mappings may be defined by an automated transformation.

Intra-model transformations. The second category of transformations is formed by the *intra-model transformations*. In this case the transformation is applied to a single model. Logically, the source and target model of the transformation are one and the same, and therefore the source and target language are the same as well. Again, we can recognise two subtypes in this category. The first subtype consists of the transformations that change the source model, which are also called *refactorings*, or *in-place transformations*. A refactoring is a mapping from the abstract syntax A_i of language L_i to the same abstract syntax A_i.

The second type of intra-model transformations are transformations that generate views. *View transformations*, like refactorings are mappings from abstract syntax A_i of language L_i to the same abstract syntax A_i, but they serve a different purpose. View transformations will never make changes in the source model, which is the purpose of a refactoring. Views present the same system from a different viewpoint, using different criteria. Views are always dependent upon their source model. If the source model changes the view should change. If the source model is removed, the view

[2] The word 'definition' is added here in order to avoid confusion with the term *semantic transformation*, which is often being used to indicate a transformation that is semantics preserving.

should be removed. We therefore consider the view to be part of the original source model, hence its category. The close link between source model and view also makes traceability a key issue to view transformations.

Inter-model transformations. The last category contains the transformations that are commonly considered to be the essence of model driven development: the *inter-model transformations.* Here, one model is transformed into another model, where the output model is often written in another language. Again, we can recognise a number of subtypes of this category. The first is very well-known in practise, namely the change of a textual representation of a model (or program) into another textual format. This is a mapping of a concrete syntax to another concrete syntax. We call this type of transformations *stream-based transformations.* The name indicates that these transformations are focused on textual, i.e. one-dimensional languages, which are handled sequentially, i.e. one token after another. Examples of this type of transformation are templates written in languages like AWK.

The second type of inter-model transformations maps an abstract syntax graph into a different abstract syntax graph. We call them *structure transformations*. Note that there is a difference between refactorings and structure transformations, even when the language of the source and target models are the same. A refactoring makes in-place changes in a model, therefore the input and output model is the same. A structure transformation produces a new model; the source and target model are two separate models. This might seem a minor difference from a theoretical viewpoint, but from the point of tool interoperability it is important.

What is making matters more complex is that structure transformations may take multiple input models and produce multiple output models. We describe in more detail how we handle this in section 4.3. In essence, the latest version of the QVT standard [15] focuses on structure transformations, although - as its name suggests - it should also provide a solution for defining views.

A third, very special case of inter-model transformations are the transformations that take an abstract syntax graph in one language as source and produce text in another language as output. Examples are transformations implemented in Velocity [16] templates. In this case the structure of the source model is available in the form of an abstract syntax graph, but the output is a character stream. We call this type of transformations *hybrid syntax transformations*. These transformations map the abstract syntax of one language upon the concrete syntax of another.

4 Elements in a Transformation Environment Architecture

This section describes the elements that constitute an open model driven development transformation environment.

4.1 Executable Units: Creators, Transformers, and Finishers

Because we focus on automation, the basic building blocks in our MDA environment are the tools that are able to execute transformations. The environment defines

three tool types: the creator, the transformer, and the finisher, which are defined as follows.

Definition 3: *(Creator) A creator is a tool that implements for some language L a mapping* M_{Cn} *in the direction from the concrete syntax to the abstract syntax.*

A creator is able to produce the abstract syntax graph of a model based on some concrete syntax, in other words, it implements a unidirectional syntax transformation. In traditional compiler terminology the creator would encompass the lexical analysis (scanning), syntax analysis (parsing), and semantic analysis (type checking, amongst other things). However, the creator concept is broader than the traditional parser concept. Because it is well-known that the complexity of parsing visual languages is in general NP-complete (see [7] for an overview of approaches to parsing visual languages), the creation of an abstract syntax graph is often automated using a syntax-directed editor. Such an editor is also considered to be a creator. Multiple creators may be defined for one language.

Definition 4: *(Finisher) A finisher is a tool that implements for some language L a mapping* M_{Cn} *in the direction from the abstract syntax to the concrete syntax.*

A finisher is able to take an abstract syntax graph of a model and to create some concrete syntax representation of this model. It could, for instance, write the model to file. Finishers, like creators, implement syntax transformations, but they may also implement hybrid syntax transformations. In traditional compiler terminology the finisher would be called a deparser. Again, the concept finisher is broader than the concept deparser. For instance, a syntax directed editor could provide a diagram generating option that implements the finisher functionality. Multiple finishers may be defined for the same language. Although in general the mapping M_{Cn} will be bi-directional, there is no need in the MDA environment to have a corresponding finisher for each creator, or vice versa.

Definition 5: *(Transformer) A transformer is a tool that implements the mapping* $A_{i*} \rightarrow A_{j*}$.

A transformer is able to take one or more abstract syntax graphs and to transform them into different abstract syntax graphs. It implements either a refactoring, a view transformation, or a structure transformation, in other words, it implements a model-to-model transformation.

4.2 Non-executable Units: ModelTypes or Languages

The fourth building block in our MDA environment specifies the type of the models to be transformed. This is an essential unit though it is not executable. It is defined as follows.

Definition 6: *(ModelType) The type* A_m *of a model m is the abstract syntax of the lan-guage in which M is written.*

We use the term *ModelType* instead of language to distinguish between the specification of a language and a certain implementation of this language. The relation

between a model and its *ModelType* is called the *instanceOf relationship* in [17], whereas the relation between a model and its language is called the *modelOf* relationship.

Because transformations may take multiple input models and produce multiple output models, we define the following.

Definition 7: *(Input types). Each executable unit (creator, transformer, or finisher) T defines an m-tuple of its input types* $T_{\text{inTypes}} = \{A_i .. A_m\}$.

Definition 8: *(Output types). Each executable unit T defines an m-tuple of its output types* $T_{\text{outTypes}} = \{A_i .. A_m\}$.

Note that in the above definitions we focus on the abstract syntax, therefore C_{inTypes} of creator C will be the empty sequence, and likewise, for finisher F, F_{outTypes} will be the empty sequence.

Not present in the model driven development environment are semantic definition transformations or stream-based transformations. The reason to exclude the latter is that model driven development focuses on structure instead of streams. Semantic definition transformations are excluded because their nature does not permit their use in a chain of transformation executions.

4.3 Combinations of Executable Units

Using the definitions from the previous section, it is easy to see that the functionality provided by a traditional compiler would be represented by a simple creator-transformer-finisher combination. The challenge of model driven development, how-ever, is not to rebuild compilers in a different fashion, but to use a network of trans-formers and (intermediate) models to produce the desired output. Therefore, in this section we present a means to define this network.

We propose to use the following three commonly known combinatorial operators which have proven to be successful in the history of computing.

- **Sequence:** a combination of two executable units; one is executed before the other, and the output of the first is the input of the second.
- **Parallel:** a combination of many executable units; the input to all of them is the same (set of) input model(s), the output is the combination of all the outputs of all of them.
- **Choice:** a combination of an ordered list of executable units; the first unit is executed if the conditions posed by this unit are met by its input, else the next unit is tried, until finally one of them is executed, or it is clear that the input does not meet the conditions of any of the units.

In order to formalise these operators, we need to introduce the following definitions.

Definition 9: *(Transformer type) The type of transformer T is the type of the function* FUN_T: $T_{\text{inTypes}} \text{ -> } T_{\text{outTypes}}$.

Definition 10: *(Creator type) The type of a creator C is the type of the function* FUN_C: $S_{\text{empty}} \text{ -> } C_{\text{outTypes}}$, *where* S_{empty} *represents the empty sequence.*

Definition 11: *(Finisher type) The type of a finisher F is the type of the function* FUN_F*:* $F_{inTypes}$ -> S_{empty}.

Definition 12: *(Transformer condition) The condition* $COND_T$ *of transformer T is the type of the function* $COND_T$*:* $T_{inTypes}$ -> *Boolean.*

Using the types of the basic elements and the combinatorial operators, we define a language of transformation expressions, according to the following rules.

1. The application of transformer T, denoted by T(m_1, .., m_n), is allowed when every m_i represents a model, and the n-tuple of types of these models {A_{m1}, .., A_{mn}} is equal to $T_{inTypes}$.
2. The conditional application of a transformer T, denoted by $T_{cond}(m_1, .., m_n)$, is allowed when the application of T is allowed and $COND_T(m_1, .., m_n)$ = true.
3. The sequence of unit T_1 followed by unit T_2, denoted by [T_1 ; T_2] , is allowed when $T_{1.outTypes}$ is equal to $T_{2.inTypes}$. Each unit can be either a creator, transformer, or finisher. The type of the combination is FUN_{T1} ° FUN_{T2}: $T_{1.inTypes}$ -> $T_{2.outTypes}$.
4. Unit T_1 and unit T_2 may always be combined in parallel, denoted by [T_1 || T_2]. The type of the combination is $T_{1.inTypes}$ Δ $T_{2.inTypes}$ -> $T_{2.outTypes}$ + $T_{2.out\text{-}Types}$, where + denotes the concatenation of both tuples, and Δ denotes a right tuple overwrite. A right tuple overwrite creates a tuple with the union of all the elements of the two tuples. Whenever both tuples have a given element, the value of the leftmost argument tuple is taken. Note that the output models of both participating units are separate; T_1 and T_2 do not generate parts of the same model, both produce their own output models.
5. A 'choice' combination of transformer T_1 and transformer T_2, denoted by [T_1 or T_2], is always allowed. The type of the combination is $T_{1.inTypes}$ Δ $T_{2.inTypes}$ -> $T_{2.outTypes}$ + $T_{2.outTypes}$, where + denotes the concatenation of both tuples, and Δ denotes a right tuple overwrite. The difference with the parallel operator is that both participating transformations will be applied conditional, as define in rule 2.

Compositions of transformers can be regarded as transformers themselves, thus allowing compositions to be used as part of another combination. The combinatorial operators defined above are higher order functions known amongst others from functional programming languages like Haskell [18].

5 The MDA Control Center Implementation

In section 2.1 we explained why a model driven development environment should be open to the addition of new transformation tools and why it should provide a means to combine the execution of transformation tools in a tool execution chain. In sections 3 and 4 we described the types of elements that can be part of such a development environment. In this section we describe how we have implemented such an environment.

5.1 The MCC Eclipse Plug-In

To implement our MDA environment we have created an Eclipse plug-in called *MDA Control Center (MCC)*. This environment uses the Eclipse extension point mechanism [19] to recognise the available units. It defines four extension points, each of which specifies a certain type of (Eclipse) plug-in.

5.2 Extension Points for the Executable Units

Three MCC extension points specify the three types of executable units defined in section 4.1*: Creator, Transformer*, and *Finisher.* Note that from the point of view of the MCC a transformer, creator, or finisher does not represent the actual tool, instead it represents the service offered by the tool. This also means that it is possible that a single plug-in implements multiple extension points. For example, the same plug-in may function both as a creator and as a transformer. In fact, one could build a plug-in that implements all extension points. In the following we will use the term *MCC service* to indicate either a creator, transformer, or a finisher.

An example of the declaration of a plug-in that implements both the transformer and the creator extensions points can be found in figure 3. In this example the creator reads resources of type "file" that have ".alan" as file extension, and produces models of type "IAlanModel", whereas the transformer takes as input an "IAlanModel" and produces as output an "OJPackage". (Alan is one of the languages for which we have defined a number of MCC services, see for more information [20], and "OJPackage" is part of our implementation of the Java metamodel, which is part of the Octopus tool [21].)

Note that each transformer may define multiple inputs and multiple outputs. In that case the order in which the outputs appear in the declaration determines the order of the elements of the tuples $T_{inTypes}$ and $T_{outTypes}$, as defined in section 4.3.

5.3 Extension Point for the Non-executable Unit

The fourth extension point specifies the type of the models to be transformed as defined in section 4.2: the *ModelType*. The fact that the MCC deals with in-memory representations of models, i.e the abstract syntax graphs, means that resources like files are not considered to be models. Another consequence of the focus on abstract syntax graphs, is that it is necessary to handle implementations of languages. *A ModelType* plug-in defines an implementation of the metamodel of a language.

For instance, it is not enough to claim that a certain model is a UML model as specified by the UML 2.0 superstructure [17], instead we need to know from which set of classes that implement the UML 2.0 superstructure, this model is an instantiation. There can be large differences between a model that is an instantiation of one UML implementation and another. For example, the Eclipse UML2 project [22] defines an implementation in Java based on EMF [23], but many other implementations —in other languages— are possible.

```
<plugin
  id="com.klasse.alan.alan2java"
  ....
  <extension
        point="com.klasse.mdacontrolcenter.creator">
     <creator
          resource="file"
          filter=".alan"
          output="com.klasse.alan.abstractsyntax.IAlanModel"
          label="Alan Model Creator"
          class="com.klasse.alan.MCCCreator">
     </creator>
  </extension>
  <extension
        point="com.klasse.mdacontrolcenter.transformer">
     <transformer
          output="com.klasse.javametamodel.OJPackage"
          input="com.klasse.alan.abstractsyntax.IAlanModel"
          label="Alan to Java Transformer"
          class="com.klasse.alan.javagen.MCCTransformer">
     </transformer>
 </extension>
</plugin>
```

Fig. 3. Example extension point implementations

5.4 Executing Transformations

The MCC offers its users the possibility to run simple Creator-Transformer-Finisher combinations without using any complex composition facilities. Each resource is associated with certain extra properties. Using these properties the user can indicate for each resource separately which creator should be used, and which transformer and/or finisher should be used to work on the thus created in-memory representation. Next to the properties view the MCC offers a button and a resource menu item called *Run Transformer*. By clicking this button or selecting the menu, the user can initiate a run of the service combination given by the properties of the given resource.

In order to make the combinations of executable units as defined in section 4.3 available to MCC users, we have created a small scripting language for transformation combinations. Each script itself defines a new transformer, which is available for use in another script or in a creator-transformer-finisher combination as defined by the resource properties. An example of an MCC script can be found in figure 4. It implements the example given in figure 1 in section 1.

```
transformer kleppe.myFirstScript (in m1: FuncModel,
                                  m2: Security)
{
    m4 := T2( T1( m1 ), m2 )->first();
    T4 ( T3(m4) ) || T5(m4) || T6(m4)
}
```

Fig. 4. An example transformer script

5.5 Type Checking

The interoperability between the executable units in MCC is taken care of by the Eclipse environment. However, MCC performs extra type checks on (the composition of) the executable units. A first type check that is performed by the MCC, is a check on the plug-in declarations. For each plug-in that declares input and output types, the types are matched against the types known in the MCC.

Additional type checking in MCC is implemented by dedicated operations that compare the type of each element in the list of inputs of the transformation with the required inputs ($T_{inTypes}$). If the types do not match, the user is issued an error message.

6 Related Work

We have found that the work of Xavier Blanc e.a. [24, 25] is closely related to the work described in this paper. Their Model Bus tackles the same problems in the manner of OMG's CORBA. There are however, a few differences, the most important one being that the MCC offers a scripting language to define new services.

Other work that has resemblance to our work is [26]. The differences between their ToolBus approach and MCC are that the ToolBus uses a common data representation whereas MCC offers more generality and flexibility because it uses several data representations, which are determined by the ModelTypes that are available in the environment. Furthermore, the ToolBus enables communication between processes other than data exchange, using messages or notes. The MCC does not offer this possibility.

The UMLAUT transformation toolkit [27] is build with the same intension as MCC: to provide the model designer with a freedom of choice with regard to combinations of transformations to be executed. The differences are that UMLAUT is limited to transforming UML models whereas MCC is able to handle models written in various languages. Furthermore, although UMLAUT provides a transformation library and a pluggable architecture, the composition of transformations in UMLAUT is internal rather than external.

7 Summary and Future Work

In this paper we have defined the elements that should be present in an open model driven development environment. In the process we have established the difference between internal and external composition of transformations, and we have developed a linguistically based taxonomy of transformations. Furthermore, we have described an implementation of the open model driven development environment, which includes a scripting language that enables the user to define his own transformations based on the transformation tools that are available.

To support the interoperability of transformation tools, every unit must be defined as Eclipse plug-in, but no further restrictions apply. This makes the MCC one of the most generic MDA environments. What is new in our approach is the application of knowlegde from the fields of compiler construction and functional programming to the area of model transformations. Our research has shown that the well-known concepts from the area of compiler construction have a limited application in the area of model transformation. In this paper we have extended these concepts to fit them to the new challenges of model driven development. In the future, our work will focus on taking into account performance or optimization (i.e., not model) parameters to transformations.

References

[1] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled, Principles of Model_Driven Architecture.* Addison-Wesley, 2004.

[3] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

[4] Stuart Kent. Model driven engineering. In *Proceedings of IFM2002*, volume 2335 of LNCS. Springer-Verlag, 2002.

[5] Colin Atkinson and Thomas Kühne. A generalized notion of platforms for model-driven development. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering,* pages 139–178, 2005.

[6] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In Jorn Bettin, Ghica van Emde Boas, Aditya Agrawal, Ed Willink, and Jean Bezivin, editors, *Proceedings of the 2nd OOPSLA Workshop on Generative Technique in the Context of the Model Driven Architecture,* Anaheim, October 2003. ACM Press.

[7] R. Bardohl, M. Minas, A. Schurr, and G. Taentzer. Application of graph transformation to visual languages, 1999.

[8] Revised submission for MOF 2.0 Query/Views/Transformations RFP. Technical Report ad/2005-03-02, OMG, March 2005.

[9] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering,* 23(5):279–295, 1997.

[10] Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson. Semantic anchoring with model transformations. In Alan Hartman and David Kreische, editors, *Model Driven Architecture – Foundations and Applications*, volume 3748 of LNCS. Springer-Verlag, November 2005.

[11] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories, Assembling Applications with Patterns, Models, Frameworks, and Tools.* John Wiley & Sons, 2004.

[12] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] MOF Model to Text Transformation Language RFP. Technical Report ad/04-0407, OMG, 2004.

[14] H. Kastenberg, A. Kleppe, and A. Rensink. Engineering objectoriented semantics using graph transformations. Technical Report, University of Twente, December 2005. Pre-final version available at http://www.cs.utwente.nl/rensink/papers/taaldraft.pdf.
[15] MOF QVT final adopted specification. Technical Report ptc/05-11-01, OMG, 2005.
[16] Velocity. http://jakarta.apache.org/velocity/.
[17] Ivan Kurtev Ivanov. *Adaptability of Model Transformations.* PhD thesis, University Twente, Enschede, The Netherlands, May 2005.
[18] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language. Technical report, February 1999.
[19] Erich Gamma and Kent Beck. *Contributing to Eclipse, Principles, Patterns, and Plug-Ins*. Addison-Wesley, 2004.
[20] Anneke Kleppe. Towards general purpose high level software languages. In Alan Hartman and David Kreische, editors, *Model Driven Architecture – Foundations and Applications,* volume 3748 of *LNCS*. Springer-Verlag, November 2005.
[21] Octopus: Ocl tool for precise UML specifications. http://www.klasse.nl/octopus.
[22] Eclipse uml 2 project. http://www.eclipse.org/uml2.
[23] The eclipse modeling framework. http://www.eclipse.org/emf.
[24] Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich. Model bus: Towards the interoperability of modelling tools. In *Proceeding of the Workshop on Model Driven Architecture - Foundations and Applications 2004,* Linkping, Sweden, June 2004. Linkping University.
[25] Xavier Blanc, Marie-Pierre Gervais, Maher Lamari, and Prawee Sriplakich. Towards an integrated transformation environment (ITE) for model driven development (MDD). In *Proceedings of the Invited Session "Model Driven Development", 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'2004),* Orlando, USA, July 2004.
[26] J.A. Bergstra and P. Klint. The discrete time toolbus – a software coordination architecture. *Science of Computer Programming,* 31:205–229, 1998.
[27] Jean-Marc Jézéquel, Wai-Ming Ho, Alain Le Guennec, and Franccois Pennaneac'h. UMLAUT: an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors*, Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.