SiTra: Simple Transformations in Java

D.H.Akehurst[1], B.Bordbar[2], M.J.Evans[2], W.G.J.Howells[1] K.D.McDonald-Maier[3]

[1] University of Kent
{D.H.Akehurst, W.G.J.Howells}@kent.ac.uk
[2] University of Birmingham
B.Bordbar@cs.bham.ac.uk, mje33@cantab.net
[3] University of Essex
kdm@essex.ac.uk

Abstract. A number of different Model Transformation Frameworks (MTF) are being developed, each of them requiring a user to learn a different language and each possessing its own specific language peculiarities, even if they are based on the QVT standard. To write even a simple transformation, these MTFs require a large amount of learning time. We describe in this paper a minimal, Java based, library that can be used to support the implementation of many practical transformations. Use of this library enables simple transformations to be implemented simply, whilst still providing some support for more complex transformations.

1 Introduction

Model Driven Engineering (MDE) or Model Driven Development (MDD) [7] is an approach to software development in which the focus is on *Models* as the primary artefacts in the development process. Central to MDD are *Model Transformations*, which map information from one model to another. In general, we can view MDD as a general principle for software engineering that can be realised in a number of different ways (using different standards) and supported by a variety of tools. One of the most common realisations of MDD is via the set of OMG standards known as Model Driven Architecture (MDA) [25]. MDA, it is claimed, improves the software development process by enhancing productivity, portability, interoperability and ease of maintenance [20]. There are currently a variety of MDD tools that can be used to implement transformations [29].

Specification and definition of a model transformation is a complex task. This involves significant domain knowledge and understanding of both the source and target model domains. Even when you understand both models, defining the mapping between corresponding model elements is no easy task. Recently a variety of model transformation specification languages have been developed e.g. [17, 22, 32]. These languages are very rich and are used in various domains [9, 33, 37]. However, elegant execution of the specifications is still a research issue in many cases and may require significant manual intervention in order to provide an implementation. Implementation of a model transformation requires a different set of skills.

In a large project, it is possible to divide the specification and implementation between two different groups of people who have relevant skills. In the case of smaller groups of developers and newcomers to MDD, the combined effort involved in becoming an expert in the two sets of skills described above is overwhelming. In

particular, the steep learning curve associated with current MDD tools is an inhibitive factor in the adoption of MDD by even very experienced programmers.

To address this issue, the current paper describes a simple Java library for supporting a programming approach to writing transformations, based on the following requirements:

- **Use of Java for writing transformations:** This relinquishes the programmer from learning a new language for the specification of transformations
- **Minimal framework:** To avoid the overhead of learning a new Java library, the presented method has a very small and simple API

The presented method is not intended as a replacement for a full Model Transformation Framework or as a model transformation specification language, rather it is intended as a "way in" for experienced programmers to start using the concepts of transformations rules, without the need to learn a new language, or get to grips with a new framework of tools and development environments.

Our library enables transformations rules to be written using Java in a modular fashion and includes the implementation of an algorithm to execute a transformation based on those rules.

The next section of this paper provides some background on MDD. Section 3 introduces an example transformation task which is used in section 4 to aid the description of the use of our simple transformation library. Sections 5 and 6 discuss the Limitations of SiTra and compare it to other model transformations approaches. The paper concludes in section 7.

2 Background

2.1 MDD

A model transformation is a program that takes as input a graph of objects and provides as output another graph of objects. If we consider the development of a program that provides a solution to this problem there are a number of alternative ways to structure it.

A very basic (unstructured) approach would be to write a single function (or method) containing a mix of loops and *if* statements that explore the input model, and create objects for the output model where appropriate. Such an approach would be widely regarded as a bad solution and it would be very difficult to maintain.

A better solution, from a programming perspective, would be to make use of a programming pattern, such as the visitor pattern [14]. This provides a controlled way to traverse a source model, and a sensible means to structure the code for generating an output model. However, this pattern does have a few drawbacks. Firstly, the input model must be implemented in such a way that is supports the visitor pattern (i.e. the objects must implement a certain interface); an input model may well not support the required interfaces. Secondly, the visitor pattern is designed to navigate tree structures rather than graphs.

A Model Transformation approach to structuring a solution would make use of the following two concepts:

1. Transformer – the primary transformation object; it contains a collection of rules, and manages the process of transforming source model objects into target model objects.
2. Rule – a rule deals with specific detail of how to map an object from a source model into an object of the target model. One or more rules may or may be applicable for the same type of object and it is necessary to have a means to determine the applicability of a rule for a specific object, not just its type.

2.2 Model Transformations

Within the context of MDD, model transformation is the primary operation on models that is talked about. However, it is not the only one; operations such as model comparison, model merging etc are also considered, although these could be seen as particular types of model transformation.

The concept of model transformations existed before QVT and even before MDA. The following topics each address some aspect involving the notion of transforming data from one form to another.

- Compiling Techniques [1]
- Graph Grammar/Transformations [11]
- Triple Graph Grammars [30]
- Incremental Parsers [16]
- Viewpoint framework tools [13]
- Databases, Update queries
- Refinement [10]
- XML, XSLT, XQuery [34-36]

To be literal about it, even simple straight forward programming is frequently used as a mechanism for transforming data. This becomes more stylised when programming patterns such as the Visitor pattern [14] are used as a way to visit data in one model and create data in another.

Some techniques [4, 5, 27] base the transformation language on the notion of relations. However, this too is a new application of the old ideas as originally applied (for example) in the fields of databases (e.g. Update Queries) and System Specification (or refinement) using the formal language Z [31] a language which is heavily dependent on the notion of relations and their use for the manipulation of data.

The interesting aspect of the MDD approach to transformation is the focus on:

- Executable specifications; unlike the Z approach.
- Transforming models; models being viewed as higher level concepts than database models and certainly higher level than XML trees.
- Deterministic output; the main problem with Graph Grammars is that they suffer from non-deterministic output, applying the rules in a different order may result in a different output.

Much work on model transformation is being driven by the OMG's call for proposals on Queries, Views and Transformations (commonly known as QVT) [26]. There are a number of submissions to the standard with varying approaches, a good

review of which is given by [15] along with some other (independent) approaches such as YATL [28], MOLA[18] etc. and earlier work such as [2].

There are a set of requirements for model transformation approaches given in [15], of which, the multiple approaches it reviews, each address a different subset. As yet there is no approach that addresses all of the requirements.

3 A Simple Transformation Library

Our simple transformation library consists of two interfaces and a class that implements a transformation algorithm. The aim of the library is to facilitate a style of programming that incorporates the concept of transformation rules. One of its purposes is to enable the introduction of the concept of transformation rules to programmers who are, as yet, unfamiliar with MDD; thus enabling the programmer to stay with familiar tools and languages and yet move towards an MDD approach to software engineering.

The two simple interfaces for supporting the implementation of transformation rules in Java are summarised in Table 1. The Rule interfaces should be implemented for each transformation rule written. The Transformer interface is implemented by the transformation algorithm class, and is made available to the rule classes.

```
interface Rule<S,T> {
  boolean check(S source);
  T build(S source, Transformer t);
  void setProperties(T target, S source, Transformer t);
}
interface Transformer {
  Object transform(Object source);
  List<Object> transformAll(List<Object> sourceObjects);
  <S,T> T transform(Class<Rule<S,T>> ruleType, S source);
  <S,T> List<T> transformAll(Class<Rule<S,T>> ruleType,
                             List<S> source);
}
```

Table 1

Rules

A transformation problem is split up into multiple rules; our SiTra library facilitates this using the *Rule* interface. A class that implements this interface should be written for each of the rules in the transformation. The methods of this interface are described as follows:

1. The implementation of the *check* method should return a value of *true* if the rule is applicable to the source object. This is particularly important if multiple rules are applicable for objects of the same type. This method is used to distinguish which of multiple rules should be applied by the transformer.
2. The *build* method should construct a target object that the source object is to be mapped to. A recursive chain of rules must not be invoked within this method.
3. The *setProperties* method is used for setting properties of the target object (attributes or links to other objects). Setting the properties is split from

constructing the target so that recursive calling of rules is possible when setting properties.

If it is impossible to distinguish between multiple rules using the *check* method, explicit rule invocation must be used to transform objects for which multiple rules apply. Objects that are derived from properties of the source object should be converted to objects for properties of the target object by calling the transform method on the transformer. It is the job of the transformer algorithm to keep track of already mapped objects, it is not necessary to be concerned about this when writing a rule.

Transformer

In order to use the rules, add the rule classes to an instance of the *Transformer* interface and call the *transform* method with the root object(s) of the source model.

An implementation of the *Transformer* interface is provided with a class *SimpleTransformerImpl*. It implements the simple transformation algorithm shown in Table 2. The full implementation of this algorithm includes additional error handling, not shown here for clarity of reading the algorithm rather than the error handling.

The four methods on the transformer interface are simply different convenience mechanisms for invoking the same algorithm. Two facilitate explicit invocation of a rule, and two facilitate the transformations of a list of source objects into a list of target objects.

```
T transform(Class<Rule<S,T>> ruleType, S sourceObj) {
  List<Rule> rules = getRules(ruleType)
  for(Rule r: rules) {
    if ( r.check(source) ) {
      T tgt = getExistingTargetFor(ruleType, source);
      if (tgt==null) {
        tgt = r.build(source, this);
        recordMapping(ruleType, source, tgt);
        r.setProperties(tgt,source,this);
      }
      return tgt;
    }
  }
}
```

Table 2

The transformation algorithm takes two parameters, the type of the rule to use and the source object to transform. This provides an explicit transformation (i.e. the rule to use is explicitly provided). Alternatively, implicit invocation can be used (an alternative method on the Transformer interface) which passes the Rule interface as the ruleType for this algorithm.

The *getRules* method retrieves a list of rule objects that conform to the type of the passed ruleType. These rules are each checked to see if they are applicable to the source object (using the *check* method of the *Rule* interface). If the rule is applicable, and the source object has not already been mapped using that rule (the *getExistingTargetFor* method), then the build method of the rule is invoked in order to construct the target object. This target is subsequently recorded by the transformer so that future transformations of the same source object by the same rule do not cause duplicate target objects. Finally the *setProperties* method on the rule is invoked;

having recorded the mapping between source and target previously, any transformation request within *setProperties* that invokes the same rule on the same source object will simply return the already built object, rather than trying to build a new one and causing a non terminating recursive loop.

4 Case Study

To illustrate the use of our simple transformation library, we define an example transformation problem based on the example addressed at the Model Transformations in Practice workshop of MoDELS 2005 [8]. The next subsection gives an overview of this example, followed by a subsection that discusses the use of our library to provide an implementation.

4.1 Example Problem

This example requires the definition of a transformation from a simple class diagram language into a relational database specification. The models for each of these languages form the source and target models for the transformation. They are illustrated below in Figure 1

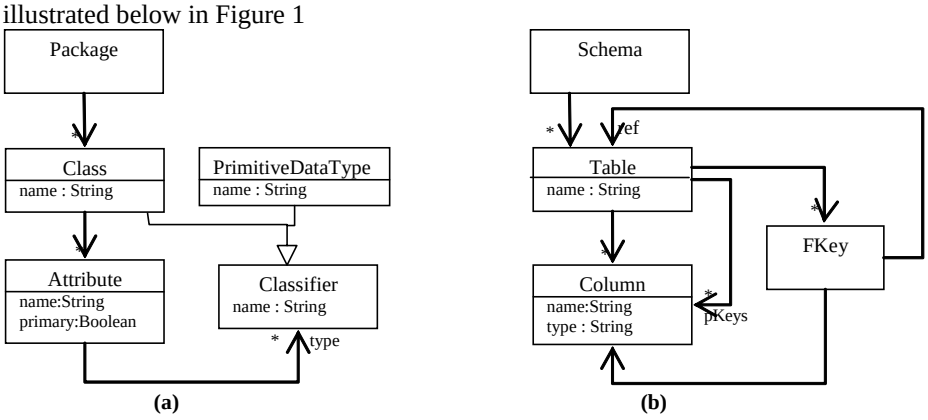

Figure 1

The detailed requirements of the transformation are summarised as follows, more details can be read in the call for papers of the workshop [8]:

- Classes are to be mapped to tables
- Attributes are to be mapped to Columns in the table
- An attribute marked as primary forms part of the primary key of the table
- The name of the attribute is the name of the column
- The type of the attribute, if it is a primitive data type, is to be the type of the column
- If the type of the attribute is a class, then the attribute should be mapped to a set of columns that are the primary key columns of the table corresponding to the class. In this case the name of the columns should be formed by combining the name of the attribute being mapped and the existing column name.

- The foreign keys of a table should be defined, in accordance with the columns correspond to an attribute with a class type.

4.2 Using SiTra

We describe here a selection of different rule implementations that help us to illustrate how to write rules of differing complexity. We use the above example as the problem for which the rules are written. The full implementation of the problem can be downloaded from [3].

Simple Rule

A very simple rule to implement is one that maps one object and its attributes directly onto another, i.e. a very simple Class to Table transformation rule. The SiTra implementation for such a rule could be written as indicated in Table 3.

```
class Class2Table implements Rule<Class,Table> {
  ...
  public Table build(Class cls, Transformer t) {
    Table tbl = new Table( cls.getName() );
    return tbl;
  }
  public void setProperties(Table tbl, Class cls, Transformer t) {
    for(Attribute att: cls.getAttribute()) {
      tbl.addColumn( new Column( att.getName(),
                                 att.getType().getName() )
                   );
    }
    return tbl;
  }
}
```

Table 3

This rule is very simple and does not fully adopt the concepts of transformation rules. The code correctly constructs a corresponding table object for the source class object. However, it explicitly carries out the construction of column objects for each attribute. Using the concept of model transformation rules, this mapping should be carried out by a separate rule.

As a separate rule it could be reused in a different context (e.g. for determining a set of primary keys); as it is currently we would have to repeat the code for mapping attributes to columns if we wished to reuse it.

Facilitating Rule Reuse

To illustrate the reuse of a rule, we extend the Class2Table rule to require it to set the property on the table objects that indicates which columns are primary keys. In the simple class diagram model, there is a property on the Attribute class for indicating which attributes should be considered primary; and in the RDB model there is a property on the Table class which indicates a set of columns that define the primary key.

We split the mapping code into two rules, one for classes to tables and one for attributes to columns. The new Class2Table rule only contains code that concerns

constructing a table object and setting its properties. All the code regarding constructing and setting properties of columns is moved to a new Attribute2Column rule. These rules are shown in Table 4.

```
class Class2Table implements Rule<Class, Table> {
  ...
  public Table build(Class cls, Transformer t) {
    Table tbl = new Table( cls.getName() );
    return tbl;
  }
  public void setProperties(Table tbl, Class cls, Transformer t) {
    tbl.setColumn((List<Column>)t.transformAll(cls.getAttribute()) );
    List<Attribute> primAtts;
    ... // select attributes from cls with 'getPrimary()==true'
    tbl.setPKeys( (List<Column>)t.transformAll(primAtts) );
    return tbl;
  }
}
class Attribute2Column implements Rule<Attribute,Column {
  public Column build(Attribute att, Transformer t) {
    Column col = new Column( att.getName(), att.getType().getName() );
    return col;
  }
  ...
}
```

Table 4

The transformation of attributes to columns, and thus the invocation of the Attribute2Column rule, is caused by calls to the transformer (shown in bold type), which request the transformation of a list of attributes. The Attribute2Column rule is reused in the Class2Table rule, rather than explicitly constructing columns as in the previous version. In fact, the transformation algorithm will determine whether or not to invoke the Attribute2Column rule, depending on whether or not it has already recorded a mapping for each source attribute object.

Hierarchy of rules

The ability to write rules and reuse them is sufficient for most simple transformations. However, to support slightly more complex transformation problems, we can introduce a notion of a hierarchy into the rules, thus enabling us to implement situations as follows.

The example requires us to have two mechanisms for mapping attributes onto columns:

1. If an attribute has a type that is a primitive data type, perform the mapping as before.
2. If the type of an attribute is a class, then we map the attribute to a collection of columns, created from the primary key attributes of the class type. The names of the columns must be constructed from the original attribute name, and the names of the primary key attributes of the class type. This mapping process may of course be recursive, and the primary keys may have a type that is a class.

This requirement requires that we alter the Attribute to Column mapping rule, rather than mapping an attribute to a single column, we map an attribute to a set of columns; and we define two separate mapping rules:

- *PrimitiveTypeAttribute2SetColumn*
- *ClassTypeAttribute2SetColumn*.

The Class2Table rule does not need to know which of these rules is being used for a particular attribute; it simply calls the transformer, requesting the transformation of attributes into sets of columns, much as before, but now we must flatten the returned list of sets of columns.

Although a primitive data type attribute always maps to a single column, the transformation is made simpler by treating the two rules the same. We can define a common 'super' rule (a common super type) for the two rules; the intention is to ensure that the target (and source) types of each sub rule are conformant. The common rule and sub rules are defined as shown in Table 5.

```
abstract class Attribute2SetColumn
         implements Rule<Attribute, Set<Column>> {
}
class PrimitiveTypeAttribute2SetColumn extends Attribute2SetColumn {
  public boolean check(Attribute att) {
    return att.getType() instanceof PrimativeDataType
  }
      ...
}
class PrimitiveTypeAttribute2SetColumn extends Attribute2SetColumn {
  public boolean check(Attribute att) {
    return att.getType() instanceof Class
  }
    ...
}
```

Table 5

As you can see in the code, the check method for the two sub rules is different, and this is used by the transformer to determine which rule to invoke for each attribute.

Explicit rule invocation

Another more complex feature, useful when defining model transformations, is the ability to explicitly invoke a specific rule (or super rule) for a particular source object. In fact, based on our experience, we prefer to always invoke rules explicitly wherever possible as this means that the transformation algorithm operates more efficiently, we have a clearer vision of where recursive rule invocation may be occurring, and the Java generics mechanism handles casting the result of the transform method.

In the example, a table may contain many foreign keys, and each foreign key should reference the table for which its set of columns forms a key. Our mapping from class to table must also set the collection of foreign keys for a table, in addition to setting the table's primary keys and columns. The foreign keys for a table can be created by mapping attributes onto FKey objects.

The difficulty here is that attributes are already mapped onto a set of columns (by the Attribute2SetColumn rule). We are now requiring a second rule (Attribute2FKey) that also maps class type attributes, but onto different target objects.

The transformer currently has no way of knowing which of these rules it should use when asked to transform a source object of type Attribute. Both rules are needed, but used at different times. There is no way to distinguish between them using properties of the source object and the check method.

The only solution is to explicitly inform the transformer of which rule we wish to use. (In this specific case there is another way to perform the transformation without specifying the rule, but it is messy.)

Table 6 illustrates a version of the Class2Table rule that make use of explicit rule invocation, by calling the transform method and passing the type of the rule we wish to invoke.

```
class Class2Table implements Rule<Class,Table> {
  ...
  public Table build(Class cls, Transformer t) {
    Table tbl = new Table( cls.getName() );
    return tbl;
  }
  public void setProperties(Table tbl, Class cls, Transformer t) {
    for(Set<Column> cols:
        t.transformAll(Attribute2SetColumn.class,cls.getAttribute()) {
      tbl.getColumn().addAll(cols);
    }
    List<Attribute> primAtts;
    ... // select attributes from cls with 'getPrimary()==true'
    for(Set<Column> cols:
        t.transformAll(Attribute2SetColumn.class, primAtts){
      tbl.getPKeys().addAll(cols);
    }
    tbl.setFKey( t.transformAll( Attribute2FKey.class,
                                 cls.getAttribute())   );
    return tbl;
  }
}
```

Table 6

The code highlighted in bold type shows the explicit invocation of transformation rules.

5 Limitations of SiTra

The primary purpose of SiTra is to be simple. Some of the limitations can be over come by extending the transformer interface, but we feel that this would violate our primary objective of a "simple" transformation approach. This of course has a cost, specifically that there are limitations in that we cannot tackle some of the more complex transformation problems.

One of the more general limitations regards a situation in which there is more than one rule that should map to the same target object. There is no way to determine, using SiTra, which of the rules should construct the target object. It is necessary for the designer of the transformation to decide which rule should construct the object; the others must retrieve it using that rule.

Another limitation is regarding the recursive invocation of rules. We facilitate this by splitting the construction and setting properties of a target object. However, there

is no means to enforce this, and there are potential design issues regarding situations in which some properties may need to be set in the build method and some not.

These limitations are associated to fairly complex transformation problems, and given the main aim of SiTra as a tool to support the implementation of simple transformations, they are not considered to be failings of SiTra, and they are simply acceptable limitations given the primary purpose of the library.

6 Comparison

The example transformation was addressed by a number of different submissions to the MTIP workshop [8]. Using these submissions we can provide a comparison with the approach described in this paper. As stated in the introduction, the library described in this paper is not intended as a replacement for a full Model Transformation Framework or as a model transformation specification language, rather it is intended as a "way in" for experienced programmers to start using the concepts of transformations rules, without the need to learn a new language, or get to grips with a new framework of tools and development environments.

Given this purpose it can be argued that a comparison between SiTra and the existing transformation languages and frameworks is not really appropriate. However, it is interesting to note what can and can't be achieved with SiTra in relation to these other approaches.

The graph transformation approaches [21, 32] have many merits with respect to formalism and a long history of us. However, they require a significant amount of new material to be learnt for novice users and also require significant libraries and development environments in terms of supporting framework. The source and target models are expressed using the notion of graphs, where as with SiTa, the source and target models are simple Java objects. The transformations specification use similar concepts of rules but require a new language to be learnt for writing them, rather than the SiTra approach of using a programming language directly.

The declarative rule based approaches [4, 17, 22] suffer many of the same problems. They all require a specific model transformation specification language to be leant. Tefkat [22] and ATL [17] are both supported by a transformation engine and environment similar in concept to our Transformer implementation class (as the engine) and a Java IDE (as the environment), although in a much more heavyweight manner than SiTra.

Our Java based environment does not of course provide any specific support for debugging transformations; debugging has to be done via Java debugging tools, which are sufficient, however do make debugging a little more complex as one has to debug the rules via the internal workings of the Transformer class.

The imperative approaches [19, 23, 24] are perhaps the most similar to SiTra in terms of the style of writing a transformation rule. However, they too, all expect the transformation writer to learn a new language, and require use of a bespoke environment in which to execute the transformations.

7 Conclusion

The primary conclusion of the paper is that simple transformations can be implemented simply. It is unnecessary to have a huge MTF framework in order to

solve a simple problem. Larger MTFs are useful for more complex situation when a full Model Driven Development environment is used; however, for simple transformations a simple framework is sufficient.

Model transformations are a new concept and introducing them to engineers unfamiliar with MDD can be problematic. By using a programming language as the basis for writing transformation rules, we eliminate one of the barriers to learning the concept of software engineering via transformations, namely that of learning a new language.

This paper has illustrated the use of a small code library as the basis to support development of a model transformation. This approach includes mechanisms for rule reuse, sub typing of rules, alternative transformation algorithms, and is not constrained by a specific model repository implementation.

SiTra is obviously not a declarative approach to model transformation; it is definitely an imperative approach, based on the underlying programming language of Java. It supports the explicit or implicit invocation of specific transformation rules; source and target objects can be single objects or collections of objects. It is design to support single direction transformations, with no support for iterative or active transformations. In essence it is designed to support the simplest kinds of transformation, primarily as a means to aid a programmer in learning the concept of writing transformation rules. However, we have found it to be a very useful and effective mechanism for implementing a number of transformations, and made serious use of it as a means to implement transformations as part of other projects.

In addition to the example illustrated in this paper the authors have been making effective use of this library to support other transformation applications such as: OWL-S to BPEL [12]; UML State diagrams to VHDL [6]; diagrams to abstract syntax of State Diagrams; and XML to XMI.

Acknowledgements

This research is supported at the University of Kent though the European Union ERDF Interreg IIIA initiative under the ModEasy grant.

References

1. Aho, A., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley. ISBN 0201100886 (1986)
2. Akehurst, D.H.: Model Translation: A UML-based specification technique and active implementation approach. Computing. University of Kent at Canterbury, Canterbury (December 2000)
3. Akehurst, D.H., Bordbar, B.: SiTra. 2006. http://www.cs.bham.ac.uk/~bxb/SiTra.html
4. Akehurst, D.H., Howells, W.G., McDonald-Maier, K.D.: Kent Model Transformation Language. Model Transformations in Practice Workshop, part of MoDELS 2005, Montego Bay,Jamaica (October 2005)
5. Akehurst, D.H., Kent, S., Patrascoiu, O.: A relational approach to defining and implementing transformations between metamodels. Journal on Software and Systems Modeling 2 (November 2003) 215
6. Akehurst, D.H., Uzenkov, O., Howells, W.G., McDonald-Maier, K.D.: Compiling UML State Diagrams into VHDL: An Experiment in Using Model Driven Development.

ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences), Genova, Italy (submitted)
7. Berre, A., Hahn, A., Akehurst, D.H., Bezivin, J., Tsalgatidou, A., Vermaut, F., Kutvonen, L., Linington, P.F.: State-of-the art for Interoperability architecture approaches. InterOP Network of Excellence - Contract no.: IST-508 011, Deliverable D9.1 (November 2004)
8. Bezivin, J., Rumpe, B., Schurr, A., Tratt, L.: Call for Papers. Model Transformations in Practice Workshop, part of MoDELS 2005, Montego Bay, Jamaica (August 2005)
9. Bordbar, B., Staikopoulos, A.: On Behavioural Model Transformation in Web Services. Conceptual Modelling for Advanced Application Domain. Springer Verlag, Shanghai, China (2004)
10. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Springer-Verlag, Berlin, Germany. ISBN 1-85233-245-X (2001)
11. Ehrig, H., Engels, G., Kerowski, H.-J., Rozenberg, G.: editors Handbook Of Graph Grammars And Computing By Graph Transformation Volume 2: Applications, Languages and Tools. World Scientific (1999)
12. Evans, M., Bordbar, B., Akehurst, D.H.: Model tranformation from OWLs to BPEL: a case study. The 9th IEEE International EDOC Conference (EDOC 2005), Hong Kong (submitted)
13. Finkelstein, A., Kramer, J., Nuseibah, B., Finkelstein, L., Goedicke, M.: Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. International Journal of Software Engineering and Knowledge Engineering 2 (March 1992) 31-58
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0201633612 (1995)
15. Gardner, T., Griffin, C., Koehler, J., Hauser, R.: A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. OMG, ad/03-08-02 (2002)
16. Ghezzi, C., Mandrioli, D.: Incremental Parsing. ACM Transactions on Programming Languages and Systems 1 (1979) 564-579
17. Jouault, F., Kurtev, I.: Transforming Models with ATL. Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (October 2005)
18. Kalnins, A., Barzdins, J., Celms, E.: Basics of Model Transformation Language MOLA. Workshop on Model Driven Development (WMDD 2004), Oslo, Norway (June 2004)
19. Kalnins, A., Celms, E., Sostaks, A.: Model Transformation Approach Based on MOLA. Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (October 2005)
20. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture--Practice and Promise. Addison-Wesley. ISBN 032119442X (2003)
21. Konigs, A.: Model Transformations with Tripple Graph Grammars. Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (October 2005)
22. Lawley, M., Steel, J.: Practical Declarative Model Transformation With Tefkat. Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (October 2005)
23. Muller, P.-A., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., Studer, P., Jezequel, J.: On Executable Meta-Languages applied to Model Transformations. Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (October 2005)
24. Murzek, M., Kappel, G., Kramler, G.: Model Transformation in Practice Using the BOC Model Transformer. Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (October 2005)
25. OMG: Model Driven Architecture (MDA). Object Management Group, ormsc/2001-07-01 (July 2001)

26. OMG: Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. Object Management Group, ad/2002-04-10 (April 2002)
27. OMG: Revised submission for MOF 2.0 Query / Views / Transformations RFP (ad/2002-04-10), QVT-Merge Group, Version 1.0. Object Management Group (April 2004)
28. Patrascoiu, O.: YATL:Yet Another Transformation Language. 1st European MDA Workshop, MDA-IA, University of Twente, the Nederlands (January 2004) 83-90
29. PlanetMDE. http://planetmde.org/
30. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Tinhofer, G. (ed.): WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Vol. 903. LNCS, Springer Verlag, Herrsching, Germany (June 1994) 151-163
31. Spivey, J.M.: The Z Notation: a reference manual. Prentice Hall (out of print, available at http://spivey.oriel.ox.ac.uk/~mike/zrm/). ISBN 0139785299 (2001)
32. Taentzer, G., Ehrig, K., Guerra, E., Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varro, D., Varro-Gyapay, S.: Model Transformations by Graph Transformations: A Comparative Study. Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (October 2005)
33. Vela, B., Acuna, C.J., Marcos, E.: A Model Driven Approach for XML Database Development. ER 2004: 23rd International Conference on Conceptual Modeling. Springer, Shanghai, China (November 2004)
34. W3C: XSL Transformations (XSLT) Version 1.0. Clark, J. (ed.). W3C Remomendation, REC-xslt-19991116 (November 1999)
35. W3C: XML 1.1. Yergeau, F., Cowan, J., Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E. (eds.). W3C Remomendation, REC-xml11-20040204 (April 2004)
36. W3C: XQuery 1.0 and XPath 2.0 Data Model (XDM). Fernandez, M., Malhotra, A., Marsh, J., Nagy, M., Walsh, N. (eds.). W3C Candidate Remomendation, CR-xpath-datamodel-20051103 (Novermber 2005)
37. White, J., Schmidt, D.C., Gokhale, A.: Simplifying Autonomic Enterprise Java Bean Applications Via Model-Driven Development: A Case Study.: MoDELS, Montego Bag, Jamaica (October 2005)