Toward Customizable and Bi-directionally Traceable Transformation between VDM++ and Java

Fuyuki Ishikawa[1]

GRACE Center, National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan,
f-ishikawa@nii.ac.jp

Abstract. VDM allows for formalization, verification and validation of software specifications, typically focusing on only abstract essences of the target system. Therefore, it is necessary to derive programs from VDM specifications in an efficient and reliable way, while incorporating details and reflecting implementation strategies. This paper discusses a method and tool to support this process through customizable and bi-directionally traceable transformation. Specifically, transformation rules from VDM++ to Java are specified for explicitly defining the implementation strategies and customizing the code generation process, e.g., introduction of implementation-specific variables. These rules can then be used for bidirectional transformation between VDM++ specifications and Java code. Changes at either of the two sides can be reflected to the other side even with the existence of implementation strategies that essentially lead to gaps between VDM++ and Java. This paper reports initial attempts that started with definitions of variables and method signatures, or structure definitions as in class diagrams.

Keywords: VDM, Bidirectional Model Transformation, Code Generation, Traceability

1 Introduction

VDM is a method for formal specification of software systems [5, 6]. Currently, VDM is said to be lightweight, because the languages and current supporting tools for VDM can be used in a similar way to those for programs. Specifically, modules or classes are defined with variables and methods (functions/operations), and typically tested by using the interpreter [1, 4].

Because structures of the languages for VDM (VDM-SL and VDM++) are similar to those of common programming languages such as Java, it is somewhat easy to syntactically map VDM specifications to implementation code, for many common notations often used. Actually it is necessary to do so in an efficient and reliable way, to reflect what are defined and examined in VDM into implementation code.

On the other hand, abstraction is the key in VDM (or specifications in general) [5,6]. Only abstract essences of the target system are modeled while implementation details are abstracted away. For example, the languages use abstract data types that do not mention how they are allocated with the memory space and manipulated. In addition, developers may choose to use declarative notations or to omit details unnecessary for the intended analysis of the specification. This way, there are gaps between VDM specification and implementation code. It is therefore necessary to derive programs from VDM specifications in an efficient and reliable way, while incorporating details and reflecting implementation strategies.

Existing VDM tools have not investigated this aspect. For example, code generators in VDM Toolbox only provide a few options and do not allow for incorporation of implementation strategies [4]. As a result, developers often need to modify the generated code to reflect the strategies. In addition, in that case it is necessary for developers to exclude the modified parts to avoid override by code generation and manually manage the changes. As another example, transformation tools between UML class diagrams and VDM specifications have been recently focused on [4, 11]. However, this leads to a situation where developers have two class diagrams: one written in VDM vocabularies with abstraction, and the other written in C++ or Java vocabularies with implementation details. Although the latter is popularly used in common development processes, it is not clear how to locate the former when introducing VDM. In order to effectively leverage VDM in the development process, the essential gaps in such a situation should be discussed and handled.

In response to the problem discussed above, this paper discusses a method and tool to manage the gaps between VDM specification and implementation code through customizable and bi-directionally traceable transformation. The approach is to use transformation rules from VDM++ to Java for explicitly defining the implementation strategies and customizing the code generation process. The rule language allows for local overriding so that default rules are defined to generate valid Java code while developers can add rules to customize the transformation process. In addition, a solid transformation theory and its implementation are used to provide the basis of bi-directionally traceable transformation [3,7]. This allows for reflecting changes at either of the VDM and Java sides to the other side, even with the existence of the gaps.

Realization of the method and tool requires much effort for coverage of various syntax elements as well as ideally sophisticated user interface. This paper reports initial attempts for proof-of-concept implementation, which deal with very basic parts of VDM++ and Java.

```
class TestClass                        public class TestClass{
  instance variables                     private double a;
    private a : real;                    private float b;
    private b : real;                    private int x;
    private s : seq of nat;              private LinkedList<Integer> s;
    private state : State;               private Logger log;
       -- only in model                     // only in impl
end TestClass                          }
```

Fig. 1. Transformation Example

2 Motivation

2.1 Abstraction Gaps between VDM++ and Java

Figure 1 shows a simple example of variable definitions in a VDM++ specification and its transformation into Java code. Besides the necessary syntax translation, this example includes the following transformation.

1. The variable *a* of the *real* type in VDM++ is implemented as the *double* type in Java.
2. The variable *b* of the *real* type in VDM++ is implemented as the *float* type in Java.
3. The variable *s* of the *seq of nat* type in VDM++ is implemented as the *LinkedList*<*Integer*> type in Java.
4. The variable *state* is unnecessary in the implementation code. This situation happens, for example, when a variable is necessary only to define invariants, or to define a mock to let the model run (replaced by libraries in the implementation code).
5. The variable *log* is necessary only in the implementation code. This situation happens, for example, when details unnecessary for the targeted analysis or value-added functionality such as logging are abstracted away in VDM++.

The transformations #1, #2 and #3 illustrate how abstract types are converted to concrete types that define how to allocate data with the memory space and manipulate. Among them, #1 and #2 illustrate customization (definition of different conversion for the same type). The transformations #4 and #5 illustrate model-specific or implementation-specific variables that exist at only one side of VDM++ and Java. Although the example only includes variable definitions, equivalent discussion stands also for method definitions.

This way, VDM, or early (formal) modeling methods in general, essentially leads to abstraction gaps. Specifically, to clarify the links between abstract model and the implementation, it is necessary to explicitly distinguish and manage different aspects included or excluded in the abstract model.

– What aspects in formal specification (VDM) are essential decisions, inherited to the implementation (Java code)

    - Inherited as they are (with syntax translation)
    - Inherited as with additional decisions (e.g., how to allocate and manipulate data on memory)
- What aspects in formal specification (VDM) are tentative and unnecessary in implementation (e.g., assertions, tentative mock to let it run without concrete implementation)
- What additional aspects are introduced only into the implementation (e.g., loggers)

This paper focuses on these gaps between VDM++ and Java. The gaps essentially come from the fact generally design decisions or implementation strategies are made and introduced when deriving implementation code from specifications. The approach in this paper thus does not consider VDM-to-Java "translation" but "transformation" (not generating code with equivalent structures and behaviors).

2.2 Expected Usage of VDM

VDM does not define one specific usage on how to incorporate it into the development process. This paper focuses on usages to rigorously model and validate design, used as input to implementation teams (rather than to model early requirements only for understanding the domains and problems there). VDM is suitable for the usages, compared with other methods for abstract and formal modeling, as its languages include more concrete and design-aware syntax such as object-orientation.

As illustrated in the example in Section 2.1, each class is modeled with some abstraction, but the basic class structures are discussed defined concretely. With this expectation, this paper does not consider integrating variables and methods from multiple VDM++ classes to one Java class, or decomposing variables and methods in one VDM++ class into multiple Java classes. The latter is especially useful to gradually introduce complexity, but is covered with other methods such as Event-B [2] or similar refinement methods for VDM [10].

2.3 Goal Setting

For the usages to be cost-effective and attractive, it is necessary to reflect what are modeled and validated in VDM into the implementation, in an efficient and reliable way. This paper proposes an approach to explicitly describe the gaps between the formal specification and the implementation as transformation rules. Below describes the goals this approach is intended to achieve, as well as detailed approaches given characteristics of VDM and the expected usages.

First, customized code generation is supported. In the approach, developers can customize the code generation to incorporate their own abstraction strategies or implementation strategies by specifying transformation rules. Assuming similar structures in the formal specification and in the implementation, the transformation rules denote strategies such as making data type concrete. The rules themselves just denote syntax transformation to accept wide range of customization, e.g., use of database connectivity and comment insertion. On the other hand, it is costly and often unnecessary to specify all the transformation rules. Therefore default rules are provided, and the proposed language for transformation rules allows for customization through overriding the defaults. This approach also facilitates to leverage hierarchical definition and reuse of rule sets, e.g., to reflect common implementation strategies in the domain, to generate comments in specific styles required in the team, or to generate annotations for further processing and analysis.

Second, verification through common test cases is supported. As VDM itself, if referred to as a lightweight method, does not define a specific formal way to obtain concrete code that satisfies what are validated in the specification. The approach in this paper does not consider to formally obtain code, either, as it allows for very wide range of syntax transformation. This point is different from fully formal methods that consider stepwise refinements where each refinement step is semantically (mathematically) describable and provable. Although stepwise refinements would be possible also in VDM, current tools do not support and there will be limitations when dealing with object-orientation in VDM++. Instead, test cases used for checking the VDM specification should be used for checking the implementation code. To support this test case inheritance, it is possible to automatically generate test code for the VDM specification and the implementation from one configuration, by understanding the abstraction gaps explicitly specified within the transformation rules. This aspect was discussed in the author's previous paper [8] and is omitted in this paper (though trivial changes are necessary).

Finally, traceability between formal specification and its implementation is supported. Transformation rules explicitly keep the relationships between the VDM specification and the implementation code, or how the latter is derived from the former. The relationships are essential to understand and make modifications in an existing set of specification and implementation. This paper constructs the transformation rules on the basis of a solid underpinning for bidirectional graph transformation. It allows for tracking what part in the VDM specification correspond to what part in the implementation code. In addition, it also allows for automatically reflecting changes in the implementation code to the VDM specification. Specifically, given the limitations of the current code generator, discussed in Section 2.1, this paper aims at supporting the following properties,

1. Suppose Java code J is generated from VDM++ specification V, and J is modified into J' only by introducing implementation-specific elements. Generation of VDM++ specification from J' then leads to V. The same holds for the case where $J' = J$.
2. Suppose Java code J is generated from VDM++ specification V, and V is modified into V' only by introducing model-specific elements. Generation of Java code from V' then leads to J.

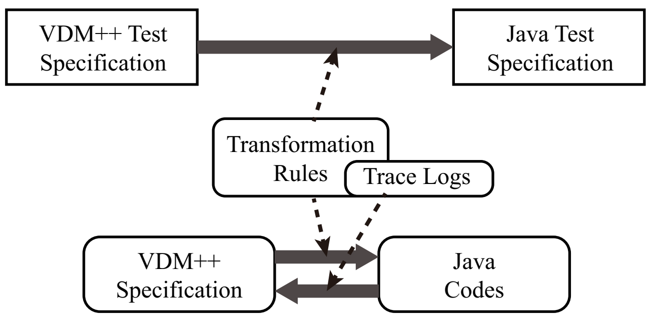


Fig. 2. Expected Usages of Proposed Framework

3. Suppose Java code J is generated from VDM++ specification V, J is modified into J', and VDM++ specification V' is generated from J'. Generation of Java code from V' then lead to J'.

The current scope of this paper is to investigate benefits and limitations of the proposed framework when the transformation rules are explicitly given by developers. Further challenges are discussed as future work, i.e., to deal with vagueness that appears when explicit knowledge is not given by developers.

2.4 Expected Features

Figure 2.4 illustrates expected features and usages of the proposed framework. As in common usages of VDM, formal specification models are built in the VDM++ language with some abstraction strategies. Class diagrams can be used to first clarify the structural aspects (variables and method signatures) of the design before the rigorous specification in VDM++. Behavioral aspects can also be modeled too, in an abstract but executable way. The VDM++ specification is analyzed through type check, review and other techniques as well as check with given test cases. These tasks can be supported by existing tools, i.e., VDM++ editor, interpreter, debugger and test frameworks as well as UML editor and UML-VDM translators.

Transformation rules are then specified, defining differences to be introduced between the VDM specification and the Java code, besides the syntax differences. Java code are then generated using the rules. The transformation procedure internally processes parse trees of the VDM specification and the Java code. Transformation itself can be done with Abstract Syntax Trees (ASTs), but locations of syntax elements in texts are also kept for implementing visual support for traceability. This paper uses the term "parse tree" instead of common "AST". In addition to transformation of the VDM specification (of the target system), the VDM test specification can be also transformed into Java test code, by using the transformation rules. Thus common test cases can be checked both for the VDM specification and the Java code.

When modifications are made in the VDM specification, Java code are generated again. Depending on contents of the modifications, transformation rules may be changed as well. It is possible to extract what parts of the VDM specification are affected by each rule, to help understand whether each rule is affected by the modifications or not. When modifications are made in the Java code, basically they can be reflected back to the VDM specification by using the logs kept in the previous transformation (VDM++ to Java). However, completeness of this functionality depends on the user interface to manage changes at the Java code, which is out of the scope of this paper.

Features involved in the above description may be used in different ways. For example, developers may choose to only rigorously model and validate the interface, and use generated Java code as skeleton. Developers may define rules incrementally and iteratively, to check the result of transformation, find points to customize by additional rules, and run transformation again.

This way, the framework provides support for connecting various deliverables in the VDM++ (or abstract modeling) world and ones in the Java (or implementation) world.

3 Overview of Approach

3.1 Transformation Rules

Transformation rules specify how to syntactically transform specific parts of a VDM++ specification into Java code. First of all, pure syntax translation, without introducing any implementation decisions, is at the base of transformation. Suppose VDM++ specification includes the following variable definition.

```
private x : real;
```

This fragment is translated into the following Java fragment.

```
private real x;
```

Syntax differences (the order and the delimiter in this case) are handled, without any transformation rules.

The above example is only for illustration, as *real* is not a valid Java type. There is no really equivalent type in Java, as *real* in VDM++ refers to a mathematical notion and does not define any specific format put on the memory [1]. Therefore a transformation rule is mandatory in this case to declare how the *real* type is implemented in Java, as *double*, as *float* or possibly as a user-defined class.

Suppose *double* is chosen to implement all the references to the *real* type. The following rule indicates this decision.

[1] VDM tools, especially interpreters may define specific formats to implement the real type

```
type-implement: real by double
```

This rule changes the transformation result as follows.

```
private double x;
```

The first part (*type-implement* in the rule denotes a pattern of implementation decisions. Default rules are defined so that valid Java code can be obtained even if developers define no rule. The current framework follows an existing code generator, and choose *double* as the default for *real*. It is actually unnecessary for developers to specify the above rule by themselves.

Suppose only for the variable x, exceptionally the *float* type is used. Another rule, reflecting an implementation decision, is then added by developers. This rule clarifies to which part the rule is applied, and locally overrides the above rule.

```
class: TestClass{
  type-implement: real by float in variable x
}
```

This way, the language for transformation rules allows developers to customize code generation behaviors when the default is not acceptable. The remainder of this section describes patterns embedded in the rule language, which are extracted from existing literatures on VDM, primarily books [5, 6].

Other rules include introduction or removal of a new variable, a new argument of a method, a new method, and a sentence inserted within the behavioral description of a method.

3.2 Foundations in Transformation

A theory for bidirectional graph transformation is applied to process the transformation rules [7]. It defines a set of graph transformation functions and their semantics so that changes in the result graph can be reflected to the source graph in a well-behaved way (in a certain sense).

Graph transformation can be defined by using the languages UnCAL or UnQL+. UnQL+ provides a high-level notation for four types of manipulation, select, replace, delete and extend. On the other hand, UnCAL is a foundational algebra with full expressivity, working as the background of UnQL+. In this paper, the high-level language UnQL+ is sufficient to illustrate the essences, though implementation of the proposed framework may also use UnCAL for detailed control of transformation.

UnQL+ allows for definition of transformation as a query, similar to a SQL query, extracting specific parts from the source graph and constructing a graph possibly adding new parts. For example, below reviews the rule in Section 3.1, that implements the *real* type by *double*.

```
type-implement: real by double
```

This rule is converted to UnQL+ queries including the following one, which replaces the term *real* in the type declaration in each variable definition.

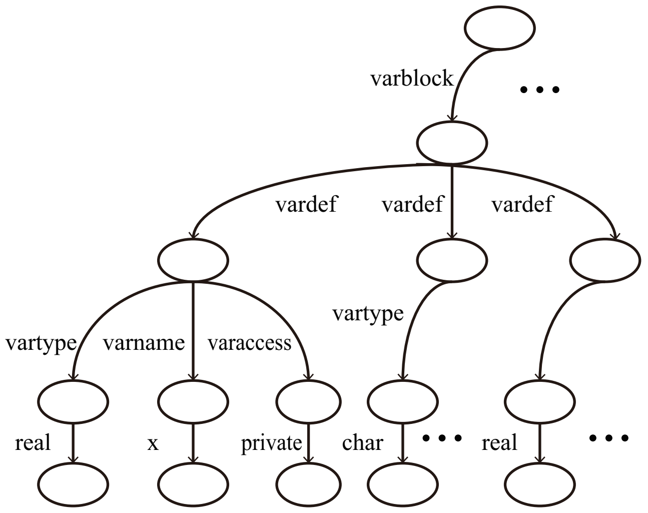


Fig. 3. Manipulation on Syntax Tree

```
replace
  varblock.vardef.vartype -> $a
  by {double:{}}
  in $db
  where {real:{}} in $a
```

Figure 3.2 illustrates an expected VDM++ syntax tree and how this query works. The first line denotes the type of query: replace some parts of the source graph with a given graph. The second and fifth lines define subtrees to be replaced. The second line refers to child subgraphs of a node reached by tracing the path *varblock.vardef.vartype* from the root. The fifth line defines conditions to declare each of the extracted subgraphs is replaced only if it contains a leaf node with the label *real*. The third line defines each of the subgraphs is replaced with a leaf node with the label *double*. The fourth line just refers to the source graph as the input to the processor (*$db*).

The above query is one of the queries generated from the transformation rule to implement *real* by *double*. For example, another rule is necessary to use *Double* inside the complex type (e.g, *HashSet*<*Double*>). This rule is converted to a similar query, but a regular expression are used for the path description to match occurrences of *real* nested in complex types.

A result graph of a query can be an input to another query. UnQL+ ensures composablity, i.e., it is possible to define a complex transformation by defining and applying small transformations one by one. In the proposed framework, each transformation rule is converted to one or a few queries in UnQL+, and then processed in order. When a query is composed from multiple rules, specific rules are evaluated before default rules. In the example described in Section 3.1, first the specific occurrence of the label *real* are replaced with *float*, and then remaining occurrences of *real* are replaced with *double*.

The other types of queries are also available, selecting or deleting designated subgraphs as well as extending (inserting) a graph to be a child of the designated path. With these types of UnQL+ queries, the transformation rules can be implemented on the basis of a solid graph transformation theory, though further examples are omitted.

3.3 Understanding and Tracing

The underlying theory and tool make logs about from which node in the VDM++ syntax tree each node in the Java syntax tree is derived from (see [7] for details). This allows for extracting correspondences between VDM++ fragments and Java fragments. When VDM++ fragments are deleted by transformation rules, there is no corresponding Java fragments. The same stands for the case when Java fragments are inserted.

As a UnQL+ query includes description of target subgraphs to be replaced, deleted or extended, it is possible to construct a select query to extract the subgraphs. Thus it is possible to identify VDM++ fragments that are replaced or deleted by each transformation rule. In addition, it is possible to identify Java fragments that are inserted by each transformation rule by identifying and logging nodes newly introduced by each rule application.

With these mechanisms, it is possible to identify correspondences among VDM++ fragments, Java fragments and transformation rules.

3.4 Reflecting Changes Backward

This paper discusses what support is feasible when the Java program is modified, and then the changes are reflected back to the VDM++ specification.

When a syntax element is replaced, added or deleted in the Java program, it can mean either an implementation-specific decision or an essential change that should be reflected to the VDM++ specification. Therefore it is necessary for supporting tools to ask developers to make some input to identify the intention.

When a modification in the Java program means an additional implementation-specific decision, a transformation rule should be defined accordingly (e.g., changing the way a concrete type is implemented, removing/introducing a variable/method). The rule is necessary to keep the modification even if a Java program is regenerated from the VDM++ specification (possibly with further modification). Thus this case can be dealt with the presented framework. Practically, automatically deriving a rule from the edited Java program would be attractive and feasible, rather than explicitly inputing the rule, but it is out of the scope of this paper and will be discussed as future work.

When a modification in the Java program means an essential change, it depends on the kind of the modification how it should be reflected to the VDM++ specification. Below discusses this point.

Suppose an element that equally exists in both VDM++ and Java is replaced. An example of this case is renaming of a variable or method, which requires the same renaming in the VDM++ specification (note that any transformation rule does not change a name of a variable or method). In this case, it is possible to reflect the change in the Java syntax tree back to the VDM++ syntax tree. The underlying transformation theory originally supports this kind of backward transformation, though the tracing mechanism presented in Section 3.3 can do as well. On the other hand, it is necessary for the user interface to understand the renaming change occurred. This will be realized, for example, by forcing developers to use a provided command for renaming (common in Eclipse-based editors), or by detecting text edit by the developer. The same discussion stands for removing an element that equally exists in both VDM++ and Java.

On the other hand, careful consideration is required for reflection of insertion in the Java program to the VDM++ specification. Generally in the underlying transformation theory, there can be multiple source graphs (VDM++ parse trees) that are transformed into the identical target graph (Java parse tree). In the case of insertion, logs kept in the previous forward transformation do not provide any information for identifying how a unique source graph is chosen among the possible ones, specifically for the inserted nodes at the target graph. This general discussion also stands for the transformation rules proposed in this paper. For example, a new *int* variable in Java may be reflected back as a new *int* variable in VDM++, but there is no reason to exclude the possibility to have a new *nat* variable in VDM++. Use of custom rules makes it difficult to deal with this problem. Because of this essential difficulty, currently insertion is recommended to be made in the VDM++ specification, though accumulation of practical use will lead to definition of default unique inverse transformation rules for insertion.

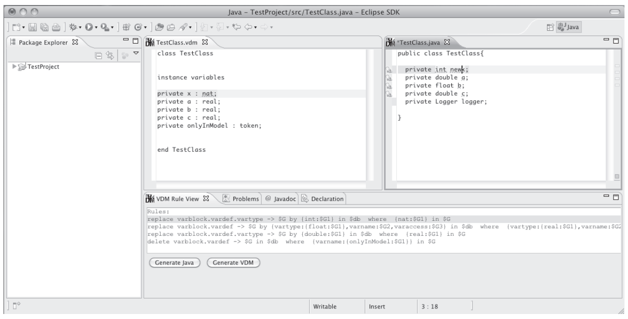
4 Prototype Implementation of GUI-based Tool

This section describes a prototype implementation of GUI-based tool for the framework.

The tool internally uses the implementation of the transformation theory with UnQL+/UnCAL, called GRoundTram [3]. GRoundTram provides the functionalities for transformation from a VDM++ syntax tree to a Java one, leaving traces for understanding from which VDM++ syntax elements each Java syntax element is derived. GRoundTram also provides the functionalities for backward transformation using the logs in the forward transformation, involving change detection in the Java syntax tree.

The tool implements the features described in Section 3, which are integrated with the features of GRoundTram. The features include conversion of transformation rules into UnQL+ queries as well as construction of select queries to identify the VDM++ fragments to which each rule affect.

Currently, the primary feature in terms of the user interface is a three-window interface to help understand and trace relationships between VDM++ specifi-



```
class TestClass


instance variables

private x : nat;
private a : real;
private b : real;
private c : real;
private onlyInModel : token;


end TestClass
```

```
public class TestClass{

  private int newx;
  private double a;
  private float b;
  private double c;
  private Logger logger;

}
```

```
Rules:
replace varblock.vardef.vartype -> $G by {int:$G1} in $db  where  {nat:$G1} in $G
replace varblock.vardef -> $G by {vartype:{float:$G1},varname:$G2,varaccess:$G3} in $db  where  {vartype:{real:$G1},varname:$G2
replace varblock.vardef.vartype -> $G by {double:$G1} in $db  where  {real:$G1} in $G
delete varblock.vardef -> $G in $db  where  {varname:{onlyInModel:$G1}} in $G
```

Fig. 4. Screenshot of GUI-based Tool

cation, transformation rules and Java code (Figure 4). When a text fragment is selected in one of the three texts (i.e., VDM++, Java, and transformation rules), highlight related text fragments in the other two texts. For example, when some text fragment is selected in the VDM++ specification, then transformation rules are highlighted that make changes on the fragment. At the same time, Java code that correspond to the VDM++ fragments are also highlighted. This user interface can be implemented with the mechanisms described in Section 3.3.

The current implementation is based on partial syntax definitions of VDM++ and Java. Future work for practical use includes full coverage of the syntax or integration with existing parsers.

5 Concluding Remarks

This paper has discusses a method and tool to support the process to derive implementation from VDM specification, in an efficient and reliable way, through customizable and bi-directionally traceable transformation.

The approach allows developers to choose any point between two extremes: fully automatic code generation without customizability and fully manual coding. Examples include cases where only some classes are implemented to accept concurrent access, and cases where a platform-specific library is used to replace a few mock classes in VDM++. On the other hand, the approach is also suitable for iterated and derivative development where both of the specification and the implementation need to be updated consistently. The transformation rules work as explicit documentation of relationships between the specification and the code, which is essential especially when responsible developers change.

Future work includes enhancement of practical implementation, such as coverage of default transformation rules, dedicated user interface and evaluation with large specifications and various implementation strategies (e.g., using databases). Future work also includes semantical support on the top of the current syntactical layer, in order to ensure transformation results are valid, at least with the default rules and preferably certain types of custom rules by developers. However the author believes the approach presented in this paper provides a solid foundation for traceability between formal specification in VDM and its implementation with programming languages.

Acknowledgments

The author would like to thank participants in Top SE [9], an education program for the industry, for questions and opinions on the problem discussed in this paper. The author would like to thank for the BiG project [3] team for quick support in the GRoundTram tool as well as Florian Wagner for his help on implementation of the prototype.

References

1. Overture - Open-source Tools for Formal Modelling. `http://www.overturetool.org/`
2. RODIN - rigorous open development environment for complex systems. `http://rodin.cs.ncl.ac.uk/`
3. The BiG Project. `http://www.biglab.org/`
4. VDM information web site. `http://www.vdmtools.jp/`
5. Fitzgerald, J., Larsen, P.G.: Modelling Systems: Practical Tools and Techniques in Software Development. Cambridge University Press (1998)
6. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs For Object-oriented Systems. Springer (2005)
7. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K., Matsuda, K.: Bidirectionalizing graph transformations. In: The 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010) (September 2010)
8. Ishikawa, F., Murakami, Y.: Challenges in inheriting test cases configurations from vdm to implementation. In: The 7th VDM-Overture Workshop (2009)
9. Ishikawa, F., Taguchi, K., Yoshioka, N., Honiden, S.: What Top-Level Software Engineers Tackles after Learning Formal Methods - Experiences from the Top SE Project. In: The 2nd International FME Conference on Teaching Formal Methods (TFM 2009). pp. 57–71 (November 2009)
10. Kawamata, Y., Sommer, C., Ishikawa, F., Honiden, S.: Specifying and checking refinement relationships in vdm++. In: The 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009) (2009)
11. Lausdahl, K., Lintrup, H.K., Larsen, P.G.: Connecting UML and VDM++ with open tool support. In: The 16th International Symposium on Formal Methods (FM 2009). pp. 563–578 (2009)