Debugging Auto-Generated Code with Source Specification in Exploratory Modeling

Tomohiro Oda[1], Keijiro Araki[2], and Peter Gorm Larsen[3]

[1] Software Research Associates, Inc. (tomohiro@sra.co.jp)
[2] Kyushu University (araki@ait.kyushu-u.ac.jp)
[3] Aarhus University, Department of Engineering, (pgl@eng.au.dk)

Abstract. Automated code generation is one of the most powerful tools supporting VDM. Code generators have been used for efficient and systematic realisation during the implementation phase. In the exploratory phase of the formal modeling process, automated code generation can be a powerful tool for users to validate integration of generated code with existing system modules and graphical user interfaces. This paper discusses debugging possibilities in the exploratory specification phase, and introduces a debugger for auto-generated code.

1 Introduction

Animation of formal specification languages is a powerful technique in both modeling and testing of formal specifications [13] as a lightweight use of formal methods. Animation plays an important role in archieving a common understanding between developers and clients in the exploratory phase of formal specification [17]. Dialects of the VDM-family have executable subsets supported by interpreters [10,11]. We have been expanding the possible use of VDM interpreters [11,14,15].

Automated code generation [4,7,8,16,19] is another technique to animate formal specifications. Code generators have been used in the implementation phase to produce the program source code in a systematic and efficient way. In general, interpreters have the advantage of flexibility, debugging support and handiness, and thus they are used for animating VDM specifications as prototypes.

Code generators can also be used as an animation mechanism for the exploratory modeling. Code generators, in general, provide better performance and flexibility with a Graphical User Interface (GUI) and external components, which are beneficial in the exploratory specification phase as well as in the implementation phase. Use of code generators in the exploratory specification phase typically brings the following challenges:

- The user has to manage consistency between two sources, namely the VDM source specification and the generated source code.
- The user has to be fluent in both VDM and the target programming language.
- When debugging the generated program code, the user also has to locate the erroneous counterpart in the source specification and fix it.

These challenges need to be addressed in order to make code generators more practical in the exploratory phase.

The rest of this paper is organised as follows: section 2 introduces the exploratory specification phase, and describes debugging tasks in the exploratory phase. Afterwards, section 3 explains the ViennaVDMDebugger, a debugger for auto-generated code integrated with a conventional debugger, and section 4 gives technical issues and possibility of the debugger. Finally, section 5 describes related work and section 6 provides concluding remarks.

2 Debugging Auto-Generated Code with Exploratory Specification

The formal specification phase targets the production of a rigorous specification based on a deep understanding of the target system and its application domains. However, specification engineers sometimes have only limited knowledge of the application domains at the beginning of the specification phase. The earlier stage of the specification phase involves two parallel activities, to produce the specification and to learn the application domains from domain experts. We call this stage of the formal specification phase as exploratory specification [17].

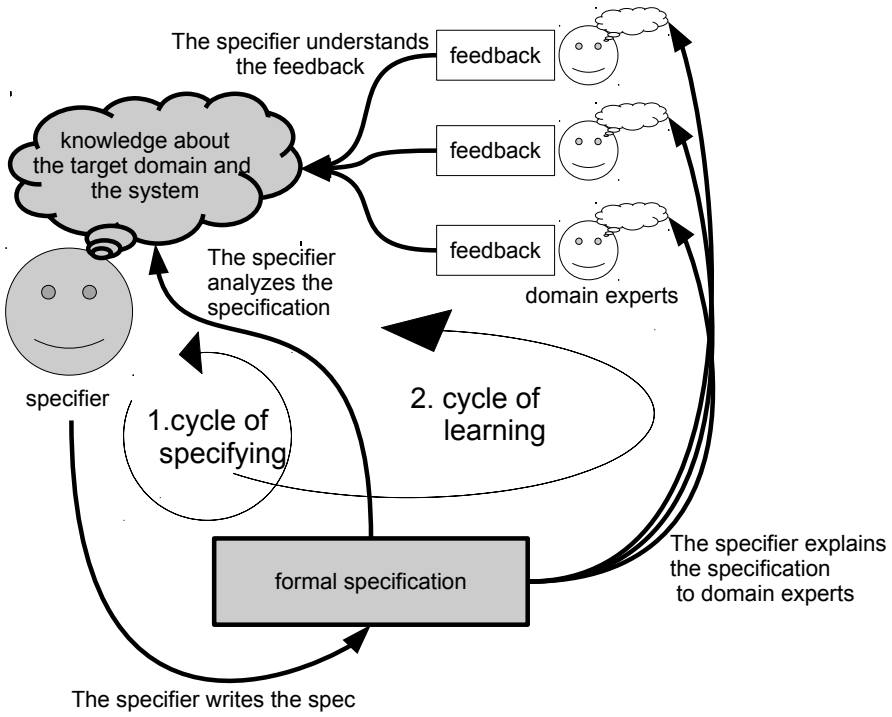

Fig. 1. The process of exploratory modeling

Table 1. Available ways of debugging animation by interpreters and code generators

Steps	Interpreters	Code generators
aware	watch variables, assertions	assertions
locating	step execution, call stacks, evaluating expressions	locating the bug in the target language and then tracing to the corresponding part in the spec
modifying	editor	editor
testing	reloading the spec and evaluating expressions	regenerating, recompiling, and restarting the code

Figure 1 illustrates the process of the exploratory specification phase. There are two cycles in the process. The first cycle starts with the specifier writing a specification based on the specifier's own knowledge of the target system and application domains. The specifier then analyses it with tools to obtain a deeper understanding. For example, a specification may suffer from an unexpected invariant violation error when animating the specification. The unexpected error is evidence that the specifier had an insufficient understanding of the target system. By debugging the specification, the specifier corrects both the specification and the understanding.

Another cycle is to learn from the domain experts. The specifier writes the specification and then explains it to the domain experts, including both application domains as well as technical domains. Animation is a strong tool to demonstrate the behaviour of the specification to domain experts who are not fluent in formal specification languages. Code generators enable the specification to be equipped with a native GUI and/or to be linked with external modules, which make the animation understandable and realistic to the application/technical domain experts.

Since the specifier needs to visualise the specifications to domain experts, the specifications must live in the larger context of GUI and networking libraries. We thus expect the specifier to understand both VDM and a programming language (Smalltalk, in our case). The goal of the exploratory specification is that the specification is validated by stakeholders, covering all functionalities and feasible to implement. The specification should be agreed by the client, application domain experts and also engineers.

In either cycle, animation often exhibits unexpected behaviour and raises errors because the specification in the exploratory phase is incomplete and error prone. The unexpected behaviour and errors are not only failures but also opportunities to understand the system and domains better. A debugger is a vehicle that provides the specifier with a new and deeper knowledge of the system. Debugging an animated specification typically involves the following steps:

1. The specifier or a domain expert becomes aware of an unexpected behaviour or an error.
2. The specifier locates the cause.
3. The specifier modifies the specification.
4. The specifier and the domain expert test the modified specification.

Available tool support varies by the method of animation. Table 1 summarises what the specifier does in each step of debugging an interpreted and code-generated animation.

Interpreters have powerful functionalities to help users to debug in each step. However, debugging a specification with a code generator is not straightforward. The specifier often has to locate the bug in the target programming language, and then trace to its counterpart in the specification. If the generated code was modified by hand, the locating task becomes more complicated; the specifier has to determine whether the unexpected behaviour or the bug was caused by the hand modification or by the source specification.

While automated code generators have advantages in performance and connectivity to external components, debugging support relies on the target language, and it is often the specifier's burden to trace the bug in the generated code to the source specification. A debugger together with a code generator that integrates two levels of debugging, namely the generated code and the source VDM specification, can reduce the burden of the specifier. The integration requires traceability from the bytes in the executable binary emitted by the target language's compiler to the syntactic fragments of the source VDM specification.

3 The ViennaVDMDebugger

In this section, we describe a debugger of the ViennaTalk environment [17,18]. ViennaTalk is a live VDM-SL development environment built on Pharo Smalltalk[4] [1]. Realisation of tool support for debugging auto-generated Smalltalk code at the source VDM-SL specification level involved two tasks: getting traceability from bytecodes to VDM-SL AST nodes and adding a UI for the source VDM-SL specification to the Smalltalk debugger. Each task will be explained in the following subsections.

3.1 Traceability

This subsection explains how traceability from bytecodes to VDM source fragments is realised. Figure 2 shows the architecture of ViennaTalk's code generators [16].

`ViennaVDMParser` parses a VDM source specification into a VDM AST (Abstract Syntax Tree), and `ViennaVDMFormatter` generates a VDM source specification from a VDM AST. `ViennaVDM2SmalltalkClass` generates Smalltalk classes and methods using `ClassBuilder` and `Compiler` that are provided in the standard library of Pharo Smalltalk. Note that ViennaTalk does not write a source file of the generated code, but creates a class instance in the Smalltalk environment.

Pharo Smalltalk provides traceability from bytecodes in a compiled method to a Smalltalk AST node. `ViennaVDM2SmalltalkClass` creates a mapping from Smalltalk AST nodes to their corresponding VDM AST nodes and stores it in the generated class. `ViennaVDM2SmalltalkClass` also stores a mapping between VDM AST nodes and VDM source ranges. Along with the Pharo Smalltalk's mapping from bytecodes to Smalltalk AST nodes, Smalltalk bytecodes can be traced to VDM AST nodes. To avoid modification to the Smalltalk `Compiler` class, the mapping from Smalltalk AST nodes to VDM AST nodes is composed of two mappings;

[4] `http://pharo.org/`

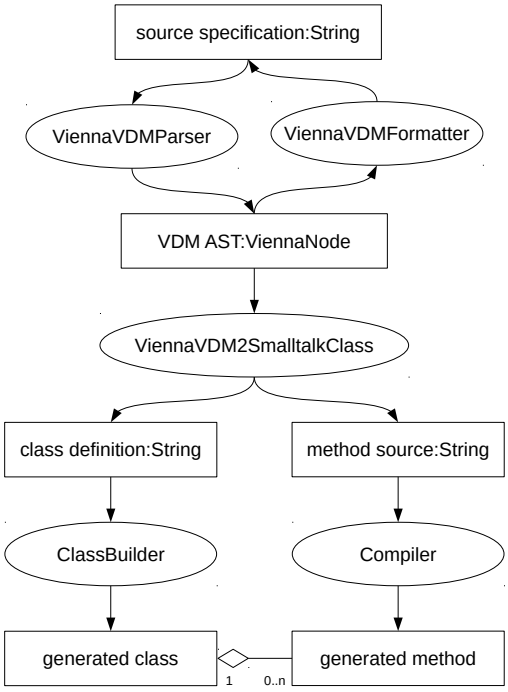

Fig. 2. Overview of ViennaTalk's code generators

one is between Smalltalk AST and Smalltalk source ranges created by the Smalltalk parser, and the other is between VDM AST and Smalltalk source ranges created by `ViennaVDM2SmalltalkClass`.

ViennaTalk defines `Vienna TracingString` as a subclass of the `String` class to implement the traceability between VDM/Smalltalk source strings and VDM AST nodes. `ViennaTracingString` holds links from its subranges to a source object.

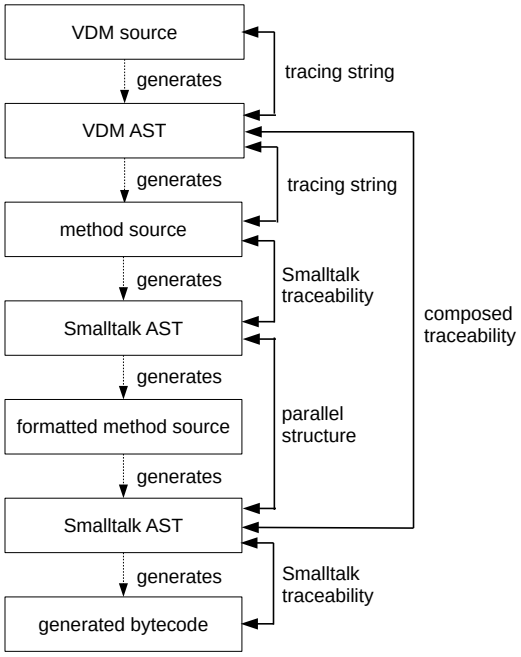

Fig. 3. Traceability composition

Figure 3 shows the composition of mappings along the chain of intermediate products from a VDM source to Smalltalk bytecodes. Traceability realised by `Vienna TracingString` is labelled as 'tracing string' in Figure 3. `ViennaVDMFormatter` generates a VDM source string as an instance of `ViennaTracingString` that has links from its subranges to the source VDM AST nodes. `ViennaVDM2Smalltalk-Class` generates a method source as an instance of `ViennaTracing String` that has links from its subranges to the source VDM AST nodes. ViennaTalk generates pretty-printed Smalltalk source code using Pharo Smalltalk's functionality via the Smalltalk AST nodes. The two Smalltalk source code items can be mapped by tracing the parallel structure of the two Smalltalk ASTs because the original source and the pretty-printed source are parsed to ASTs in the same structure. Along with Pharo Smalltalk's traceability from the Smalltalk AST nodes to corresponding Smalltalk

source subranges, ViennaTalk composes a mapping from the Smalltalk AST nodes to the VDM AST nodes. At total, an arbitrary bytecode in the generated methods can be traced up to its corresponding source VDM fragment.

`ViennaTranspiledObject` is the base class of all auto-generated Smalltalk classes. Each subclass of `ViennaTranspiledObject` manages its own specification dictionary that maps method names to VDM AST nodes in a similar way; each Smalltalk class has a method dictionary that maps method names to compiled methods. Each subclass of `ViennaTranspiledObject` has a mapping from Smalltalk AST nodes to their source VDM AST nodes. In Smalltalk, a program context is an object that can answer the object (the message receiver) and the method name (the message selector) that the program context is running in. Thus, the VDM AST node can be traced from a program context if and only if an instance of a subclass of `ViennaTranspiledObject` is in the call stack of the program context.

3.2 The UI of the Debugger

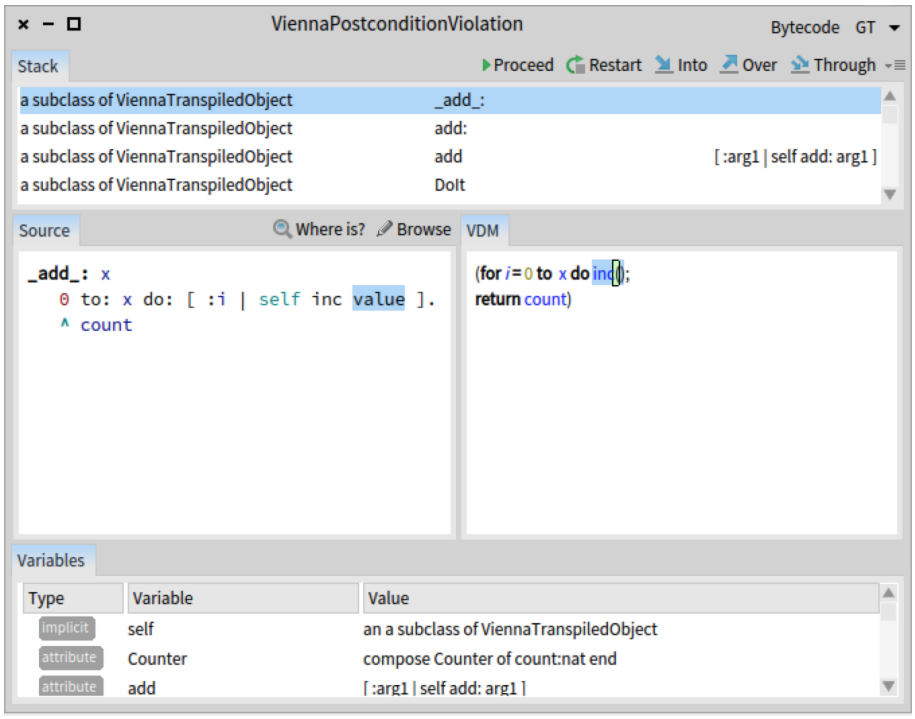

Fig. 4. A screenshot of VDMDebugger on ViennaTalk

Having traceability from program context to a VDM source fragment, ViennaTalk provides a debugger UI that displays the source VDM specification and the current executing fragment from the current program context. Figure 4 shows a screenshot of the VDMDebugger on top of ViennaTalk.

At the top, the debugger window has window operation buttons on the left, a window label `ViennaPostconditionViolation` in the middle and debugger selector buttons on the right. The window label indicates that the program execution is suspended because a post-condition is violated. The debugger selector buttons will be explained later. The second row is a series of buttons labeled `Proceed`, `Restart`, `Into`, `Over` and `Through` for step execution operations of Smalltalk program. The third row shows a call stack of the current program context where the user can select the method of concern. In the fourth row, Smalltalk source code and VDM source specification are displayed side by side, and the fragment of source that is currently to be executed is highlighted in each source. At the bottom, a list of variables and their values are shown.

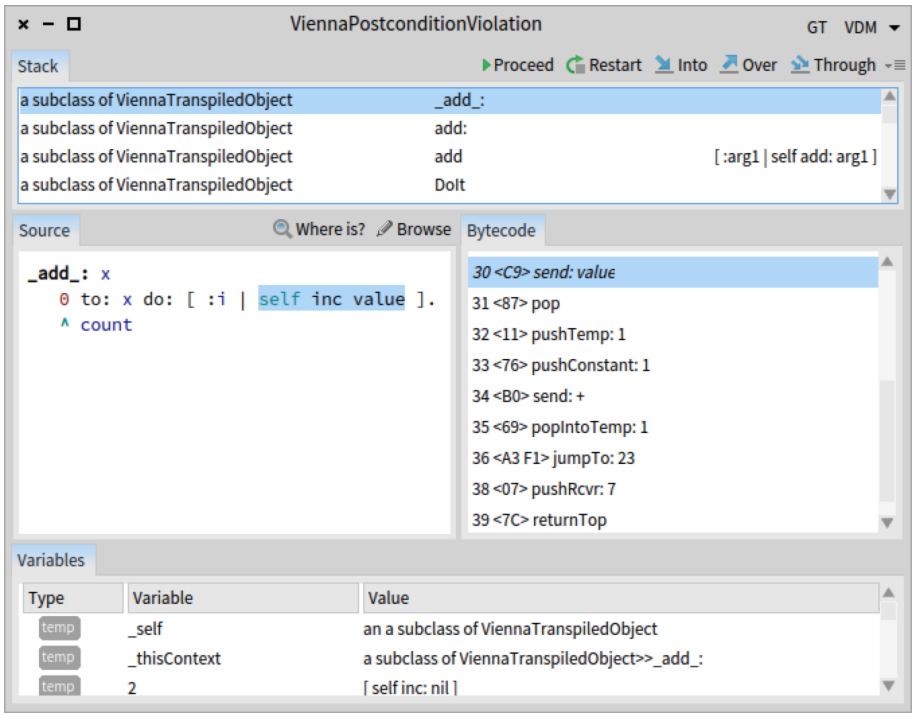

Fig. 5. A screenshot of bytecode presentation of the debugger

The debugger is integrated with the standard debugger in the Pharo Smalltalk system using a framework called a *moldable debugger* [2]. With the moldable debugger,

domain specific or purpose specific presentation can be added to the standard debugger in a pluggable manner.

The standard Pharo Smalltalk provides three presentations of the moldable debugger. The first is the generic presentation that displays the Smalltalk source code of the current program context. The second is the bytecode presentation that displays the Smalltalk source code and the bytecodes of the method being executed. The third is the unit test presentation that can visualise the difference between the expected and actual results of unit testing. ViennaTalk adds a presentation: the display of the source VDM specification of auto-generated classes as well as the Smalltalk source code.

The user can select a presentation to use via the debugger selector button on the right-top corner of the debugger window. Figure 5 shows a screenshot of bytecode presentation of the moldable debugger. By clicking on the `VDM` button at the right-top corner of the window, the debugger changes its presentation to VDMDebugger as shown in Figure 4. With these presentations, the user can see which particular part of the VDM source generated what bytecode in the generated class.

Given a program context, the VDMDebugger updates its view by the following steps:

1. The debugger scans the call stack of the program context for a context whose receiver is an instance of a subclass of `ViennaTranspiledObject`. The context is called *specification context*.
2. The debugger looks up a VDM AST node in the specification dictionary of the receiver class of the specification context. The VDM AST node is called *method spec node*.
3. The debugger generates a VDM source form the method spec node. The generated VDM source is a tracing string whose subranges are linked to their source AST nodes. The VDM source is called *method spec source*.
4. The debugger gets Smalltalk AST node from the current Program Counter (PC) value of the specification context, and looks up the VDM AST node in the AST-AST mapping of the receiver class. The VDM AST node is called *active spec node*.
5. The debugger finds the subrange of the method spec source linked to the active spec node. The subrange is called *active spec subrange*.
6. The debugger displays the active spec source and highlights the active spec subrange.

Each time when a program context proceeds, VDMDebugger shows the VDM source with the currently executed fragment highlighted along with the auto-generated Smalltalk source.

4 Discussions

In this section, we walk through a typical scenario of the debugging task on an auto-generated code to illustrate how VDMDebugger support the debugging task, and then discuss VDMDebugger in the cycle of specifying and the cycle of learning in the exploratory modeling process.

4.1 Example: debugging an erroneous counter specification

Figure 6 shows an erroneous VDM-SL specification of Counter. The bug is at the **for** statement in the definition of the `add` operation which repeats the `inc()` operation call `x + 1` times. When an expression `add(2)` is evaluated on a workspace of VDM-Browser in the transpiler mode, a post-condition violation exception is reported. VDMDebugger opens when the `Debug` button is pressed (See Figure 7). The user can also set a breakpoint at a generated method to bring the debugger.

```
state Counter of
    count : nat
init s == s = mk_Counter(0)
end

operations
    inc : () ==> nat
    inc() ==
        (count := count + 1;
        return count)
    post  count - count~ = 1;

    add : nat ==> nat
    add(x) ==
        (for i = 0 to  x do inc();
        return count)
    post  count - count~ = x;
```

Fig. 6. An erroneous VDM-SL specification of counter

ViennaTalk's transpiler generates three methods, namely `_add:`, `add:` and `add` from the `add` operation. The `_add:` method implements the body of the operation. The `add:` method checks the dynamic type of the actual arguments, checks the pre-condition, invokes the `_add:` method, checks the post-condition, and then returns the resulting value. The `add` method wraps `add:` as a closure object. The VDMDebugger opens with the source code of the `add:` method that signals the post-condition violation exception and the definition of the `add` operation with the post-condition expression highlighted to notify the post-condition expression has failed as shown in Figure 8. The execution of the `add:` method can be restarted by clicking the `Restart` button. The user clicks the `Over` button several times to step the execution forward until the invocation of the `_add:` method. The user then clicks the `Into` button to drill into the `_add:` method and repeat step execution to see how the execution proceeds and how the state changes. The user may understand the `inc()` is invoked 3 times by step execution of the `_add:` method. Figure 4 is a screenshot of VDMDebugger stepping in the `_add:` method.

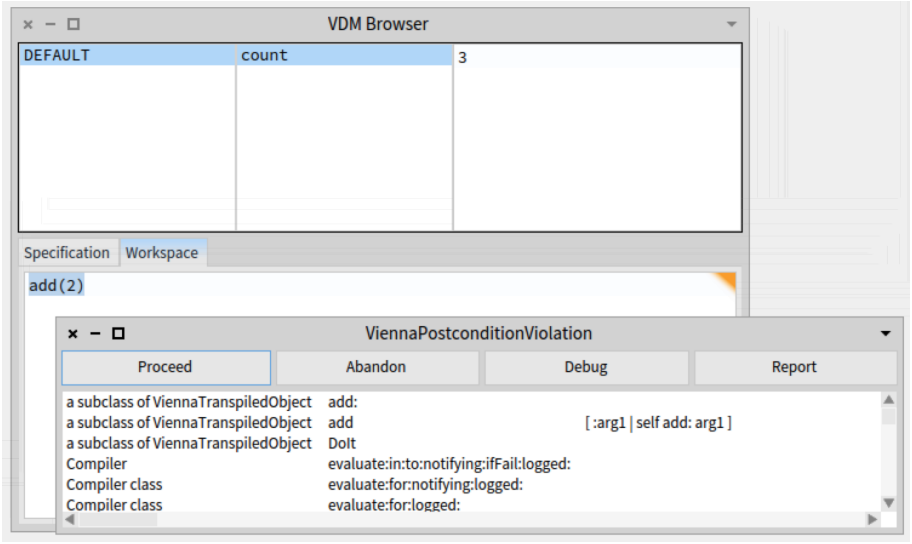

Fig. 7. A screenshot of an exception notifier raised by a post-condition violation

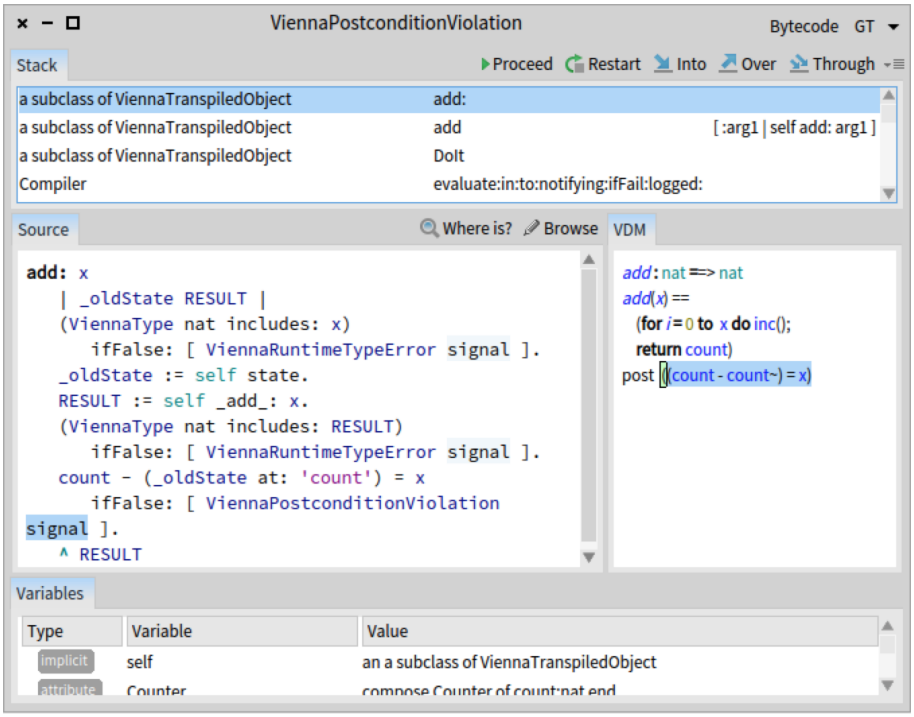

Fig. 8. A screenshot of VDMDebugger at post-condition violation

The debug execution commands such as Restart, Into and Over are based on Smalltalk's execution. Runtime type checking, pre-condition checking and post-condition checking can be debugged by step execution in the `add` method. It gives finer granularity of step executions compared to those at the VDM specification level, but it also entails extra workload for the user. For example, to reach the execution state shown in Figure 4 from the execution state shown in Figure 8, the user has to click the `Restart` button once, the `Over` button 10 times, the `Into` button once, and then the `Over` button four times. Step execution in the granularity of the VDM-SL specification is desired, and this will be the focus of future work.

4.2 VDMDebugger in the exploratory modeling

VDMDebugger is designed to support the specifiers in the cycle of specifying and the cycle of the learning presented in Figure 1. Each cycle imposes its own goal and constraints, and therefore has different requirements on the debugger.

In the cycle of specifying, the specifier is basically working alone and knows both VDM-SL and the target language, e.g. Smalltalk. The most frequent task is to understand the behaviour of the specified system rather than connectivity to a GUI or external systems. The code generator is used as a high performance animation engine. The specifier's concern is the behaviour of the specification rather than the behaviour of the Smalltalk program. The step execution operations are based on Smalltalk execution, which is a shortcoming of the current implementation of `VDMDebugger`. The specification and its state variables shown in the debugger is the minimum requirement, which the `VDMDebugger` satisfies.

In the cycle of learning, the specifier works with other stakeholders, such as application domain experts and technical leaders, who typically do not have background knowledge of VDM-SL and/or Smalltalk. The advantage of the code generator is not performance, but connectivity with GUI and external subsystems. When an unexpected behaviour such as a run-time error is observed, the specifier tries to locate the cause. The first call to make is whether the cause is in the source VDM specification or not. If the cause is in the source VDM specification, the task is to locate the bug in the VDM specification and fix it. Otherwise, the task is to locate the bug in Smalltalk code, such as GUI or connection to external systems. The specifier needs to check the VDM specification and Smalltalk code back and forth in a program context. For example, when a VDM operation is invoked by an external system via a HTTP connection and an assertion on the VDM state is violated, the cause may be either an invalid HTTP request, a bug in the code that invokes the VDM operation or a bug in the VDM specification. `ViennaVDMDebugger` provides a good support for checking both VDM specification and Smalltalk code in the same execution context.

ViennaTalk's traceability is also useful in the maintenance phase. The source VDM specification can be delivered embedded within the executable. Traceability is of key importance to formal methods. When an executable causes problematic behaviour, it is important the problematic instruction in the executable can be tracked to a particular part of the specification.

5 Related Work

VDMTools [9] has a feature for combining VDM-SL specifications with code written in C++ [5]. However, this support is only available from the interpreter and not from the generated code. On the other hand, the code generators of VDMTools are able to generate the code such that if a run-time error is encountered it is able to refer to the exact location from which the construct was generated [6].

For the Overture tool numerous features have been developed enabling a combination of VDM specifications and code written in different programming languages. The focus here has been at the interpretation level where a capability similar to that in VDMTools was presented in [14], which also enabled control to be distributed to an external application. Numerous code generators have also been developed for Overture. Of particular interest is the ability to bridge such external code between an interpretation level and a code generation level [3]. However, none of the efforts in Overture have enabled the kind of dual debugging presented in this paper.

Eiffel [12] is an Object-Oriented programming language with assertion features for Design by Contract. One can specify invariants among instance variables and pre-condition and post-condition of a method; VDM-SL also has invariants on types and states and pre-conditions and post-conditions of operations. As a general purpose programming language, Eiffel has rich and reliable library for GUI construction and network programming. The Eiffel compiler generates a native executable binary from source code. Debuggers are provided, with either GUI or command line, where the user can watch variables, set breakpoints, and execute the code by steps.

One interesting feature of Eiffel is that pieces of specifications such as invariant, pre-condition and post-condition program code are embedded within program code and they are organised by a class hierarchy. ViennaTalk also embeds pieces of specifications into auto-generated classes. The difference is that Eiffel compiler generates native executable binaries while ViennaTalk generates Smalltalk classes that resides upon Smalltalk Virtual Machine. As a result, Eiffel has advantages for producing deliverables that run efficiently on native OSs while Smalltalk has liveness that is beneficial in the exploratory modeling.

6 Concluding Remarks

Users validation and feasibility study in the exploratory specification phase is important to efficient application of lightweight formal methods. Code generators and interpreters are both mechanisms to animate formal specifications, and with the traceability of ViennaTalk, generated code can be efficiently debugged.

The foundation of ViennaTalk's traceability described in this paper relies upon the image based design of the Smalltalk environment where everything including program contexts and compiled methods are first class objects that programmers can extend in a manner. ViennaTalk pursues liveness of VDM-SL specifications. The VDMDebugger adds a further level of liveness to ViennaTalk by bringing VDM-SL source specifications into Smalltalk's live debugger.

We plan to enhance the integration between VDM-SL and the Smalltalk environment to fill the gap between formal specifications and program code. We believe such

integration will benefit the usability of both VDM-SL and Smalltalk: Smalltalk's flexibility will be made available to development of VDM-SL specifications, and rigorous construction will be made available to Smalltalk programming.

Acknowledgments

The authors would like to thank Paul Chisholm and Christoph Reichenbach for valuable feedback on drafts of this paper, and Tudor Gîrba for technical advice on the moldable tools. The authors also thank anonymous reviewers for insightful comments.

References

1. Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates, Kehrsatz, Switzerland (2009), http://pharobyexample.org
2. Chiş, A., Gîrba, T., Nierstrasz, O.: The moldable debugger: A framework for developing domain-specific debuggers. In: International Conference on Software Language Engineering. pp. 102–121. Springer (2014)
3. Couto, L.D., Lausdahl, K., Plat, N., Larsen, P.G., Pierce, K.: Decoupling validation UIs using Publish-Subscribe binding of instance variables in Overture. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 123–136. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
4. Diswal, S.P., Tran-Jørgensen, P.W., Larsen, P.G.: Automated Generation of C# and .NET Code Contracts from VDM-SL Models. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 32–47. Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
5. Fröhlich, B., Larsen, P.G.: Combining VDM-SL Specifications with C++ Code. In: Gaudel, M.C., Woodcock, J. (eds.) FME'96: Industrial Benefit and Advances in Formal Methods. pp. 179–194. Springer-Verlag (March 1996)
6. Group, T.V.T.: The VDM-SL to C++ Code Generator. Tech. rep., Kyushu University (January 2017), http://www.fmvdm.org/doc/
7. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)
8. Kanakis, G., Larsen, P.G., Tran-Jørgensen, P.W.: Code Generation of VDM++ Concurrency. In: Proceedings of the 13th Overture Workshop. pp. 60–74. Center for Global Research in Advanced Software Science and Engineering, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan (June 2015), http://grace-center.jp/wp-content/uploads/2012/05/13thOverture-Proceedings.pdf, gRACE-TR-2015-06
9. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science 7(8), 692–709 (2001)
10. Larsen, P.G., Lassen, P.B.: An Executable Subset of Meta-IV with Loose Specification. In: VDM '91: Formal Software Development Methods. VDM Europe, Springer-Verlag (March 1991)

11. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (October 2011), http://dl.acm.org/citation.cfm?id=2075089.2075107, ISBN 978-3-642-24558-9
12. Meyer, B.: Object-oriented Software Construction. Prentice-Hall International (1988)
13. Nielsen, C.B.: Dynamic Reconfiguration of Distributed Systems in VDM-RT. Master's thesis, Aarhus University (December 2010)
14. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-30885-7_19, ISBN 978-3-642-30884-0
15. Oda, T., Araki, K., Larsen, P.G.: VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In: Plat, N., Gnesi, S. (eds.) FormaliSE 2015. pp. 33–39. In connection with ICSE 2015, Florence (May 2015)
16. Oda, T., Araki, K., Larsen, P.G.: Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 48–62. Aarhus University, Department of Engineering, Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
17. Oda, T., Araki, K., Larsen, P.G.: A formal modeling tool for exploratory modeling in software development. IEICE Transactions on Information and Systems 100(6), 1210–1217 (June 2017)
18. Oda, T., Araki, K., Larsen, P.G.: ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 4:1–4:7. Prague, Czech Republic (Aug 2016)
19. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML-annotated Java. International Journal on Software Tools for Technology Transfer pp. 1–25 (2017), http://dx.doi.org/10.1007/s10009-017-0448-3