Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling

Tomohiro Oda[1,2], 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. Different dialects in the VDM-family have executable subsets. Simulated execution of a specification, either by an interpreter or a code generator, is an effective technique not only to produce production source code but also to validate the specification. This paper describes requirements on code generators for earlier stages of formal specification, a phase called exploratory modeling, and introduces an implementation in ViennaTalk. Performance, readability and liveness of the generated code are also evaluated.

1 Introduction

Formal methods are mostly used to precisely state the desired properties of a system to be engineered. In general, such formal specifications are not necessarily executable [6]. However, in order to communicate the insight of a formal specification it may make sense to express it in an executable subset in order to be able to animate its conceptual behaviour to stakeholders with little understanding of the formal notation that has been used [5, 1]. Interpretation of subsets of the different VDM dialects have been enabled for many years [12, 13]. Different tools exist for the different VDM dialects and here the most widely used ones are Overture [11] and VDMTools [10]. However, there is a balance between the time spent on making the specification executable and the insight gained so the time used to enable exploratory modelling needs to be properly balanced [3]. Different means of enabling executability exist so in this paper pros and cons for alternative solutions are considered. The primary focus in early exploration involves interpretation of executable subsets of VDM models but this also involves integration with legacy or graphical user interface code [4, 14]. Another alternative here is using code generation [7]. Target languages of code generators include those compiled into native binary code, those compiled into bytecode that runs on virtual machines and those executed by interpreters.

The rest of this paper is organised as follows: Section 2 provides a brief overview of exploratory modelling in general and ViennaTalk specifically. Afterwards, Section 3 provides a brief introduction to the code generation functionality from both Overture and VDMTools, followed by general requirements for code generation functionality seen from an explorative modelling perspective. Then Section 4 presents the core results in this paper indicating how the code generation features in the ViennaTalk solution can support the code generation requirements for exploratory modelling. Section 5 then

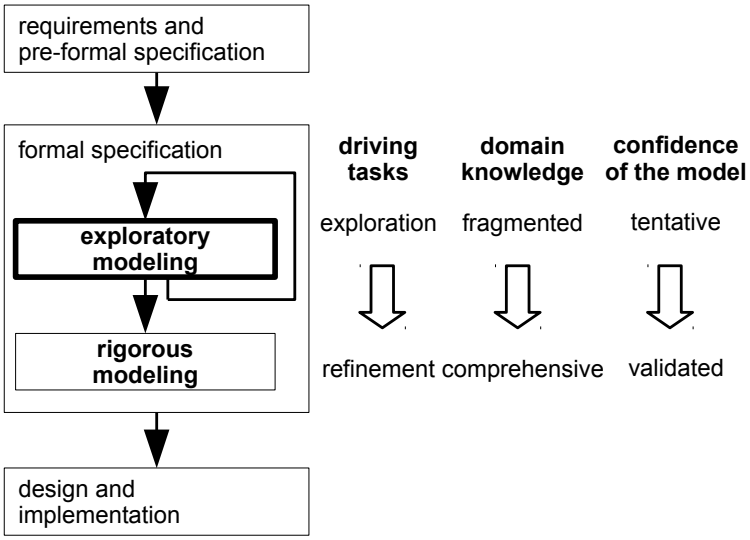

Fig. 1. Exploratory modeling in the different development phases

evaluates and compares the different code generators and their performance. Finally, Section 6 concludes the paper and points out future work possibilities.

2 Exploratory Modeling and ViennaTalk

There is no complete formal specification at the beginning of any development. Formal methods engineers often have limited and fragmented domain knowledge at the earlier stages of the specification phase. The formal methods engineers learn the nature of the target domain by writing tentative models of the system to be developed. The formal methods engineers' confidence in the model increases as the model is validated by stakeholders. The model gains in maturity and rigour when it is passed to the design and implementation phase. We call the earlier stages of the formal modeling "exploratory modeling"[15, 17, 16] (see Figure 1). The exploratory modeling involves learning efforts and validation by domain experts to find an appropriate abstraction of the problem domain. The exploratory modeling is followed by the rigorous modeling to make the model precise, concise and assured.

ViennaTalk [16] is a meta-IDE to develop tools for exploratory modeling in VDM-SL built on top of Pharo Smalltalk [2]. Figure 2 shows the configuration of ViennaTalk and external libraries of the Pharo system.

ViennaTalk is equipped with packages for interpreter wrappers called ViennaTalk-Engine, runtime support packages for VDM-SL such as ViennaTalk-Type and ViennaTalk-Value, parsers and accompanying source code analysis tools packaged as ViennaTalk-Parsers and the animation manager called VDMC. Three prototyping tools are built and bundled in ViennaTalk.

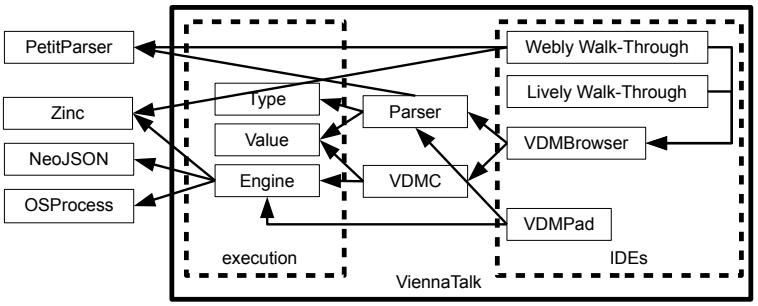

Fig. 2. Configuration of ViennaTalk

One of these is VDMBrowser [17], which has syntax highlighting, pretty printer and a live animation interface named Workspace. VDMBrowser can be used as a development tool, and also can be embedded into other development tools. The second is a UI prototyping environment named Lively Walk-Through [17], which enables VDM-SL engineers and UI designers to build a prototype with a VDM-SL executable specification and a prototypical user interface. VDM-SL engineers can confirm that the VDM-SL specification provides enough functionality to drive the user interface. UI designers can confirm that the UI is designed with appropriate assumptions on the system's functional model. The VDM-SL engineers and the UI designers can let domain experts, who may not have an engineering background, "test-drive" the UI prototype in order to validate the VDM-SL specification and the UI design. The third package is a Web API prototyping environment called Webly Walk-Through [17]. Operations and functions exported in a VDM-SL specification can be published as a Web API so that client programs can be accessed via HTTP. The fourth is VDMPad, a lightweight WebIDE for VDM-SL.

These tools are designed to support exploratory modeling in VDM-SL. A common feature among those tools is live animation that gives flexibility required by exploratory modeling. Liveness of a programming language is an ability to modify a running program without aborting the program and to continue the execution of the program with the modified code. Live animation on ViennaTalk enables the user to evaluate expressions as well as to modify the specification while the animation is running, so that the user can more deeply understand what's going on in the animation and try more alternative definitions. Flexibility is the most important aspect of support tools for exploratory modeling.

3 Requirements on Code Generators for Exploratory Modeling

VDMTools is equipped with C++ and Java source code generators. The code generators of VDMTools are reliable and widely applied in industry. The Overture tool also provides a Java generator [8] and also recently, a C generator.

Those generators are developed for the final stage of the formal specification phase: to generate production code. Java, C and C++ require development tools independent

of the VDM development environment. Use of those external tools may disturb the user because the user has to come in and out of the Integrated Development Environment (IDE) although IDEs are designed to host all activities in an integrated manner. Code generators for exploratory modeling should address those issues in order to support the user whose mental focus is not on the production of code, but on the modeling task.

In this section, requirements on code generators for exploratory modeling are listed. The requirements R1-a through R1-d are requirements specific to code generators. The requirements R1, R2 and R3 are general requirements on support tools for exploratory modeling.

R1 Direct interaction with smaller models.

Interaction between the user and the model at hand plays an important role in exploratory modeling. Direct interaction is required to support tools for exploratory modeling in general. In case of code generators, this general requirement can be broken down to specific requirements R1-a through R1-d.

R1-a Automatic compilation and execution of the generated source code.

Because the user's focus is on modeling, requiring the user to modify the generated source code should be avoided.

R1-b Compilation and execution of the generated source code must take place in the same IDE as the exploratory modeling.

IDEs are designed to support the user in an integrated manner. Switching between IDEs may diminish benefits of IDEs.

R1-c Few limitations on the specification language for automated code generation.

Exploratory modeling is carried out at earlier stages of the specification phase. The model is therefore expected to be abstract. The code generator for exploratory modeling should be able to generate executable code from as abstract a specification as possible.

R1-d Debug capability must be enabled.

Because the source model is still tentative and not sufficiently rigorous, it may be error prone. In order to debug the generated code, it is desirable that the generated code can be debugged in the way that the hand-written source code in the target language is usually debugged. The generated code should be human readable and can be easily modified.

R2 Understandable by stakeholders with no formal methods background.

In exploratory modeling, it is useful to ask domain experts and get feedback from the model. Code generators also should be suitable for communicating with stakeholders with no programming background. It is therefore desirable that GUI construction tools or visualisation techniques are available in the target language.

R3 Permissive checking by choice.

Exploratory modeling handles tentative, immature, often inaccurate models. Rejecting all suspicious models is not always productive. It is desirable that a code generator provides options to turn on or off static type checking, runtime type checking and runtime assertion testing.

R4 Continuous analysis.

Generated code can be unit tested efficiently. It is desirable that the generated code can be continuously tested by test frameworks. This requirement mainly affects choices of target languages.

4 Code Generators in ViennaTalk

ViennaTalk can automatically generate 3 kinds of Smalltalk programs, namely classes, objects of anonymous classes and scripts. The user of ViennaTalk can use 3 source code generators in three different ways: in the VDMBrowser, in the Live translator and by embedding VDM-SL expressions/statements inside Smalltalk code.

4.1 Design Rationale

We listed the design requirements based on the requirements on code generators for exploratory modeling **R1-a** through **R4**. Using the Smalltalk environment automatically satisfies requirements **R1-a** and **R1-b**. Requirement **R2** and **R4** can not be realised by code generators alone, but depend on other development tools. However, it should be noted that the generated code should be modifiable to combine with such development tools. The resulting design rationales are listed below.

- The code generator should cover as large a subset of VDM-SL as possible. This is requirement **R1-c**.
- The code generator should be able to turn on and off the runtime checking of types and assertions. This is requirement **R3**.
- The generated code should be as much like hand-written source code as possible. This is for requirements **R1-d**, **R2** and **R4**.
- The generated code should be traceable to the original VDM-SL source. This is for requirements **R1-d**, **R2** and **R4**.
- The generated code uses the standard Smalltalk library and extends its classes if necessary. New classes to implement VDM values should be minimised. This comes from requirement **R1-d**.

There are essential differences in the type systems between VDM-SL and Smalltalk and this makes naive mappings from VDM-SL types to Smalltalk classes infeasible. In VDM-SL, one value may belong to many types, i.e. 1 is a value of `nat`, `nat1`, `int`, `int inv` x == x < 10, `[nat]`, `nat | char` and so on. In Smalltalk, one object must have only one class. It is therefore not possible to define a one-to-one mapping between VDM-SL types and Smalltalk classes. A more flexible mapping between type objects and value objects is needed. Thus we made the following design decision.

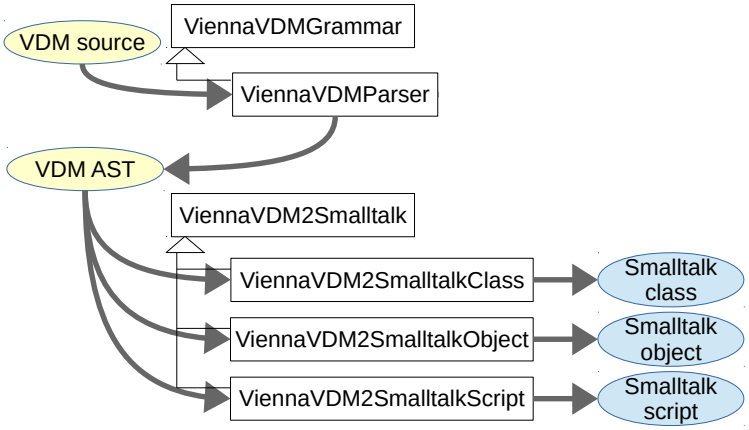

Fig. 3. Parsers and AST visitors to generate Smalltalk artefacts from VDM-SL source

– Types are not classes. A VDM type is an instance independent of classes of its values.

4.2 Implementation

Figure 3 illustrates the classes associated with parsing and code generation. ViennaTalk uses PetitParser[9] to implement a parser for VDM-SL. PetitParser is a combinatory parser library where a parser object can be synthesised from one or more parser objects. The framework provided by PetitParser allows to separate grammar definitions and corresponding outputs. ViennaVDMGrammar defines the grammar of VDM-SL which only accepts valid VDM-SL sources but does not produce any output. Subclasses of ViennaVDMGrammar defines outputs corresponding to each grammatical construct defined by the ViennaVDMGrammar class. ViennaVDMGrammar has a subclass named VDMParser which outputs an abstract syntax tree from the given VDM-SL source. The ViennaVDM2Smalltalk class is an abstract class that traverses over a given abstract syntax tree and produces Smalltalk code in the form that each of its three concrete classes defines.

The ViennaVDM2SmalltalkClass class generates one document class that represents the whole specification and module classes each of which implements the corresponding VDM-SL module. If the specification is flat, only one class for the specification will be generated. The users of ViennaTalk does not have to operate any external development tools. The generated classes can be browsed using Smalltalk's conventional class browsers, and are already compiled. The ViennaVDM2SmalltalkClass class is designed to be used in the transition from a VDM-SL specification to a Smalltalk implementation.

The ViennaVDM2SmalltalkObject class generates the same classes as those generated by the ViennaVDM2SmalltalkClasses class except that the anonymous classes do

not have global names. Because these anonymous classes do not appear in the global name space, the ViennaVDM2SmalltalkObject class leaves less footprints than the ViennaVDM2SmalltalkClass. The ViennaVDM2SmalltalkObject is designed to test specifications faster than using interpreters, either in automated test frameworks or in exploratory testing by domain experts.

The ViennaVDM2SmalltalkScript class generates a piece of Smalltalk script for a flat specification or one with a single module. Multi-moduled specifications are not supported. State invariants are not automatically checked in the generated program code. The ViennaVDM2SmalltalkScript is designed to test specifications by evaluating expressions on it or to embed a VDM expression in Smalltalk program code.

4.3 Runtime library

To generate Smalltalk code, runtime support for the generated code is required to fill the difference between VDM-SL and Smalltalk. One difference to fill is the type system. Smalltalk is a class-based and dynamically typed OO programming language and VDM-SL is a statically typed language without the OO features. One language construct in Smalltalk that possibly corresponds to the type in VDM-SL is the class. However, a naive mapping between types in VDM-SL and classes in Smalltalk is not feasible because every Smalltalk object belongs to a class while a value in VDM-SL may belong to multiple types.

ViennaTalk provides type *objects* that represent VDM-SL's types independent of classes of objects for VDM-SL values. As the types are objects in ViennaTalk, it is possible to send a message to those objects, to compose new type objects from existing type objects, and query membership of a value object. The summary of mapping of VDM-SL types to the type object in Smalltalk and the class of its value objects is shown in Table 1. The composite type and the token type have no corresponding classes in the Smalltalk standard library, and therefore ViennaComposite and ViennaToken are defined accordingly.

Some operations and language features of VDM-SL are not present in Smalltalk. The ViennaComposition class supports function compositions and the ViennaIteration implements iteration of the function composition. Pattern matching is not supported in the standard Smalltalk language. ViennaRuntimeUtils provides functionalities to simulate the pattern matching mechanisms.

4.4 Pattern matching

Pattern matching is a powerful language construct of VDM-SL which many programming languages do not support. Although pattern matching can be replaced with `if` expressions, it is not desirable to sacrifice abstraction in the exploratory phase of the specification. Supporting pattern matching is strongly required to satisfy the requirement **R1-c**. The Smalltalk language does not employ the pattern matching mechanism by itself, but ViennaTalk implements, as a library, pattern matching mechanisms for various types of patterns that appear in VDM-SL.

The ViennaRuntimeUtils class defines methods for all kinds of patterns in VDM-SL, each of which returns all possible bindings of pattern identifiers as a dictionary

Table 1. Mappings among types in VDM-SL, type objects and classes of value objects

VDM-SL type	**type object**	**the class of value objects**
nat	ViennaType nat	Integer
nat1	ViennaType nat1	Integer
int	ViennaType int	Integer
real	ViennaType real	Float
bool	ViennaType bool	Boolean
<quote>	ViennaType quote: #*quote*	Symbol
[*t*]	*t* optional	*t*'s class or UndefinedObject
t1 * *t2*	*t1* * *t2*	Array
t1 | *t2*	*t1* | *t2*	*t1*'s or *t2*'s class
set of *t*	*t* set	Set
set1 of *t*	*t* set1	Set
seq of *t*	*t* seq	OrderedCollection
seq1 of *t*	*t* seq1	OrderedCollection
map *t1* to *t2*	*t1* mapTo: *t2*	Dictionary
inmap *t1* to *t2*	*t1* inmapTo: *t2*	Dictionary
t1 -> *t2*	*t1* -> *t2*	BlockClosure
t1 +> *t2*	*t1* +> *t2*	BlockClosure
token	ViennaType token	ViennaToken
compose *t* of *f1* : *t1* *f2* :- *t2* *t3* end	ViennaType compose: '*t*' of: {*f1* . false . *t1*. {*f2* . true . *t2*}. {nil . false . *t3*}}	ViennaComposite
t inv *pattern* ==*expr*	*t* inv: [:*v* | *expr*]	*t*'s class

object. The code generator emits a piece of Smalltalk code that first gets the possible bindings by the ViennaRuntimeUtils class and then assigns the corresponding value to each pattern variable.

4.5 Preconditions and postconditions

In VDM-SL, preconditions and/or postconditions can be attached to explicit definitions of functions and operations. In exploratory modeling, runtime checking of preconditions and postconditions should be possible by the user's choice because the specification at hand may be error prone and needs runtime checking to confirm that the generated code is running as expected. Quoting preconditions and postconditions is necessary because evaluating a quoted precondition or postcondition with various kinds of parameters is an effective approach to understanding specification.

ViennaTalk's code generators support both runtime checking of preconditions and postconditions and quoting them. ViennaTalk automatically produces quoted preconditions with the `pre_` prefix and quoted postconditions with the `post_` prefix. ViennaTalk generates, from a function or an operation, a method to check the precondition and the postcondition that is separate from the method that implements the function's or the

operation's body if the runtime assertion checking option is turned on. When a precondition or postcondition is violated, an exception will be signalled accordingly. This helps the user programmer to remove the precondition and postcondition check easily. It is also possible for the user programmer to add extra precondition and/or postcondition as required by implementation details such as Smalltalk's native libraries.

4.6 Type invariants

Type invariants are also important assertions in VDM-SL. It does not only cause a runtime error when a value of the type does not satisfy the invariant, but also affects the return values of the `is_` family of expressions.

In ViennaTalk, type invariants are implemented as testing block closures held by type objects. A type object responds to the `includes:` message, which is common vocabulary among Smalltalk's collection objects for the membership query. For example, `ViennaType nat inv: [:x | x < 10]` produces the type object for `nat inv x === x < 10`, and `(ViennaType nat inv: [:x | x < 10]) includes: 100` evaluates to `false`. ViennaTalk's code generators emit a `includes:` message for the `is_` family of expressions and also for the runtime type checking.

4.7 State invariants

State invariants are another language construct of VDM-SL which is not supported in many programming languages. Along with preconditions and postconditions, state invariants must be checked at runtime when animating a specification to validate it.

ViennaTalk uses the Slot [18] mechanism of Pharo to implement state invariants. A Slot is a user-defined model of variables which defines (1) what bytecode should be generated for an assignment to a variable, (2) what bytecode should be generated for a read access to the variable, (3) what should be performed when the variable is assigned to, and (4) what should be performed when the variable is read. The `ViennaStateSlot` class is a slot for instance variables with invariants. When a variable declared as a ViennaStateSlot is assigned a value, the `inv` message is sent to the object that holds the variable. The default definition of the `inv` method is to do nothing, and ViennaTalk's code generator overrides the `inv` method to check the invariant. If the invariant is not satisfied, an exception is signalled. The state invariant is also quoted as a function.

4.8 Runtime type checking

Runtime type checking brings a subtle issue to Smalltalk code generators because Smalltalk is a dynamically typed language and has no static types on variables, arguments and return values. It is not common in Smalltalk programs to test the type of a value. Generating runtime checking code may be considered against the design rationale to generate code that is as *natural* as possible. On the other hand, runtime type checking is a strong technique in animating VDM-SL specifications for validation.

ViennaTalk addresses this issue by providing an option to enable or disable generation of runtime type checking. When the runtime type checking is turned on, the code

```
isPrime := [ :x |
        [ :_forall |
                _forall allSatisfy:
                        [ :y | (y <= 1 or: [ x = y ]) or: [ x \\ y ~= 0 ] ] ]
        value: ViennaType nat1 ]
```

Fig. 4. Smalltalk code automatically generated from a specification of prime numbers with a type bind in a set comprehension

```
isPrime := [ :x |
        [ :_forall |
                _forall allSatisfy:
                        [ :y | (y <= 1 or: [ x = y ]) or: [ x \\ y ~= 0 ] ] ]
        value: (ViennaType nat1 runtimeSet: (1 to: 256)) ]
```

Fig. 5. Smalltalk code modified to use {1,...,256} as approximation of the nat1 type

generator inserts a test using the `includes:` message to check if the value object belongs to the specified type object. If the runtime type checking fails, an exception will be signalled.

4.9 Type binds

Type binds can be used in set comprehensions, sequence comprehensions or map comprehensions, but they are not always executable. To make a comprehension expression with a type bind executable, one can rewrite the comprehension to use a set bind with a proper definition of the set. The resulting specification is executable but sacrifices abstraction. It is desirable to enable executability without modifying the abstract specification to satisfy the requirement **R1-c**.

ViennaTalk provides the `runtimeSet` mechanism that enables the generated Smalltalk code to approximate a type object with infinite members by a finite collection object. For example, the code shown in Figure 4 is generated from the VDM-SL expression
`isPrime := lambda x:nat & forall y:nat1 & y <= 1 or x = y or x mod y <> 0`, but is not executable due to the set comprehension with a type bind. Figure 5 is a modified version that the nat1 type is given a runtime set `(1 to: 256)` for approximation at the last line. The `isPrime` in Figure 5 is executable, i.e. `isPrime value: 23` evaluates to `true`. This approximation on the last line is apparently dangerous in rigorous analysis, but can be useful, with caution, to animate specifications with set binds in the exploratory modeling.

5 Evaluation

5.1 Benchmark: Prime numbers

The benchmark used to evaluate code generators is to generate a list of prime generators from natural numbers from 2 up to 10000 shown in Figure 6. The algorithm is similar to

```
state Eratosthenes of
    space : seq of nat1
    primes : seq of nat1
init s == s = mk_Eratosthenes([], [])
end
operations
    setup : nat1 ==> ()
    setup(x) ==
        (space := [k | k in set {2, ..., x}];
        primes := []);
    next : () ==> [nat1]
    next() ==
        if space = [] then
          return nil
        else let x = hd space in
          (primes := primes ^ [x];
           sieve(x);
           return x);
    sieve : nat1 ==> ()
    sieve(x) ==
        space := [space(i)
                  | i in set inds space
                  & space(i) mod x <> 0];
    prime10000 : () ==> seq of nat1
    prime10000() ==
        (setup(10000);
        while next() <> nil do skip;
        return primes);
```

Fig. 6. Benchmark specification: enumerate prime numbers less than 10000

the sieve of Eratosthenes but slightly different. The original Sieve of Eratosthenes writes to the same array of numbers. The algorithm used for benchmarking, on the other hand, generates a new list of the remaining numbers so that the benchmark will reflect the processing speed of comprehensions which often appears in practical specifications. As a result, the benchmark results mainly reflect efficiency of sequence comprehension, if expression, sequence operations and integer arithmetics.

The benchmark was run by interpreters and code generators of VDMTools, the Overture tool and ViennaTalk. They were run on Ubuntu 16.04 on a virtual machine with i5-3210M CPU @ 2.50GHz x 2 cores and 4GB main memory. Table 2 summarises the result.

As expected the code generators run faster than the interpreters. Amongst the code generators, ViennaTalk's code generators took the best and the worst positions. The script generated by ViennaTalk was slow because the script has to capture the program context (stack frame) to perform the return statements. In many languages, capturing

Table 2. Benchmarking result

Tool	Interpreter or Code generator	Language	Pattern matching	Time (ms)	Time (Overture CG=1)
VDMTools	Interpreter	C++	Yes	22,044	79.6
VDMJ	Interpreter	Java	Yes	5,281	19.1
ViennaTalk	Code generator	Smalltalk(Script)	Yes	957	3.45
VDMTools	Code generator	C++	Yes	337	1.22
Overture tool	Code generator	Java	No	277	1.00
ViennaTalk	Code generator	Smalltalk(Class)	Yes	202	0.729
ViennaTalk	Code generator	Smalltalk(Object)	Yes	193	0.700

gcc version 5.4.0, Java HotSpot(TM) 64-Bit Server VM(build 25.101-b13), Pharo 4 with 32-bit Cog VM.

```
public static void sieve(final Number x) {
    VDMSeq seqCompResult_2 = SeqUtil.seq();
    VDMSet set_2 = SeqUtil.inds(Eratosthenes.space);
    for (Iterator iterator_2 = set_2.iterator(); iterator_2.hasNext();) {
        Number i = ((Number) iterator_2.next());

        if (!(Utils.equals(Utils.mod(
                ((Number) Utils.get(Eratosthenes.space, i)).longValue(),
                x.longValue()), 0L))) {
            seqCompResult_2.add(((Number) Utils.get(Eratosthenes.space, i)));
        }
    }
    Eratosthenes.space = Utils.copy(seqCompResult_2);
}
```

Fig. 7. Java code for the sieve operation generated by the Overture tool

the current programming context is not allowed to user programmers. The Smalltalk environment has a mechanism to capture the program context but it is costly. The classes and objects generated by ViennaTalk do not need to capture the program context, but can simply return from the method context. The cost to capture the program context is the only difference between script and classes/objects. The difference between the generated classes and the generated objects is that the classes of the generated objects are anonymous. The generated methods and internal structures are the same.

In general, it is known that native binary code compiled from C++ source runs faster than Java code on Java VM, and Java code on Java VM runs faster than Smalltalk code on Smalltalk VM. However, in this benchmark, the Smalltalk objects and classes generated by ViennaTalk ran faster than Java and C++ code generated by the Overture tool and VDMTools in that order. This simple benchmark is not sufficient to investigate why Smalltalk code generated by ViennaTalk outperforms Java code and C++ code generated from the same VDM-SL source. One possible reason is the container objects for VDM-SL values.

Figure 7 shows the Java source code generated by the Overture tool, and Figure 8 is the Smalltalk code generated by ViennaTalk. The Java code includes getter calls such as `longValue()` inside the iterator for loop. The Smalltalk code uses only functionalities provided by the standard Smalltalk library, and does not use any container objects for VDM-SL values. Smalltalk is a flexible language in which the programmer can de-

```
_sieve_: x
    space := ((1 to: space size)
            select: [ :i | (space value: i) \\ x ~= 0 ]
            thenCollect: [ :i | space value: i ])
        asOrderedCollection
```

Fig. 8. Smalltalk code for the sieve operation generated by ViennaTalk

```
_prime10000
    self setup: 10000.
    [ self next value ~= nil ] whileTrue: [  ].
    ^ primes

prime10000: _op
    | _oldState RESULT |
    RESULT := self _prime10000.
    (ViennaType nat1 seq includes: RESULT)
        ifFalse: [ ViennaRuntimeTypeError singal ].
    ^ RESULT
```

Fig. 9. Generated Smalltalk code with runtime type checking

fine new methods on the kernel classes such as `Object`, `Collection` and `Integer`. For example, ViennaTalk extends the `Collection` class to define a `power` method to compute the power set of finite sets instead of providing `VDMSequence` class and defining a `power` method on it. On the other hand, the code generators of VDMTools and the Overture tool use container classes for VDM values such as the `Number` class because the target language, namely C++ and Java, do not allow user programmers to modify primitive data types and built-in classes. ViennaTalk takes advantage of the flexibility of the Smalltalk language to avoid overhead of container objects.

5.2 Code Readability

Figure 9 shows the generated Smalltalk code from the prime10000 operation. Two methods are defined in the code: `_prime10000` and `prime10000:`. The operation body and runtime type checking is separated into different methods.

The generated code is readable and also traceable. The `_prime10000` method looks natural as if it is hand-written, and pretty printed according to the Smalltalk standard coding convention. We can also see a clear correspondence between the `_prime10000` method and the `prime10000` operation in Figure 6. The `prime10000:` method implements runtime type checking on the return value. The return value from `_prime10000` is assigned to the temporary variable called RESULT, and it is tested if it is a value of the sequence of `nat1`. In Smalltalk, runtime type checking is not usually performed because Smalltalk employs a dynamic type system. ViennaTalk's code generator gives options to turn the runtime type checking and runtime assertion tests. The user can simply turn off runtime type checking generation of the code generator, or let the code generator emit runtime type checking code and confirm the generated code runs correctly. It is also easy to remove the runtime type checking because the type checking code is clearly separated from the code of the operation body.

5.3 Liveness

The Overture tool and VDMTools require external compilers and build tools while ViennaTalk does not. After the Overture tool or VDMTools generates a source code, the user need to build a native binary file or a Jar archive, and run it. The execution is performed as a separate process.

This difference does not only affect the burden of the user, but also makes a big difference in liveness. The generation and compilation of the prime number example in Figure 6 do not take a second on ViennaTalk. When objects of generated classes are running, the user can overwrite the specification, and re-generate Smalltalk classes. The running objects keep running as instance objects of the newly generated classes. With the liveness of the ViennaTalk's code generator, the user has more flexibility and lower barrier against trial and error in exploration.

6 Conclusion and Future Work

The automated code generators from VDM-SL specifications to Smalltalk are implemented based on requirements on exploratory modeling and design rationale. The performance and readability of the generated code are satisfactory. However, we can still improve usability of the code generator by developing debugging tools on the generated code. It is desirable that the specification can be edited and debugged during execution of the generated Smalltalk bytecode. We believe it is possible to implement such a specification-level bytecode debugger using the Smalltalk's flexible frameworks for live programming.

Acknowledgments This work is partially supported by Grant-in-Aid for Scientific Research (S) 24220001 and Grant-in-Aid for Scientific Research (C) 26330099.

References

1. Andersen, M., Elmstrøm, R., Lassen, P.B., Larsen, P.G.: Making Specifications Executable – Using IPTES Meta-IV. Microprocessing and Microprogramming 35(1-5), 521–528 (September 1992)
2. Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates, Kehrsatz, Switzerland (2009), `http://pharobyexample.org`
3. Fitzgerald, J.S., Larsen, P.G.: Balancing Insight and Effort: the Industrial Uptake of Formal Methods. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems, Essays in Honour of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays. pp. 237–254. Springer, Lecture Notes in Computer Science, Volume 4700 (September 2007), iSBN 978-3-540-75220-2
4. 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)
5. Fuchs, N.E.: Specifications are (preferably) executable. Software Engineering Journal pp. 323–334 (September 1992)

6. Hayes, I., Jones, C.: Specifications are not (Necessarily) Executable. Software Engineering Journal pp. 330–338 (November 1989), http://www.cs.man.ac.uk/csonly/cstechrep/Abstracts/UMCS-89-12-1.html
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. Jørgensen, P.W.V., Larsen, P.G.: Towards an Overture Code Generator. In: The Overture 2013 workshop (August 2013)
9. Kurs, J., Larcheveque, G., Renggli, L., Bergel, A., Cassou, D., Ducasse, S., Laval, J.: PetitParser: Building Modular Parsers, pp. 377–412. Square Bracket Associates (September 2013)
10. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science 7(8), 692–709 (2001)
11. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
12. 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)
13. 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
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.: 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)
17. Oda, T., Yamomoto, Y., Nakakoji, K., Araki, K., Larsen, P.G.: VDM Animation for a Wider Range of Stakeholders. In: Ishikawa, F., Larsen, P.G. (eds.) Proceedings of the 13th Overture Workshop. pp. 18–32. 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
18. Verwaest, T., Bruni, C., Lungu, M., Nierstrasz, O.: Flexible object layouts: Enabling lightweight language extensions by intercepting slot access. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 959–972. OOPSLA '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2048066.2048138