Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling

Tomohiro Oda[1,2]  
Keijiro Araki[2]  
Peter G. Larsen[3]

[2] Kyushu University  
[3] Aarhus University
Agenda

1. Introduction: Exploratory Modeling
2. Code Generator for Exploratory Modeling
3. ViennaTalk and its Code Generators (Demo)
4. Performance and Discussion
5. Conclusion
exploratory specification using animation

Exploratory Spec 1

animation

Simulated Program 1

valid solution

solution space 3

abstraction
exploratory specification using animation

valid solution

Exploratory Spec 1

Exploratory Spec 2

Simulated Program 1

solution space 4

animation

abstraction
exploratory specification using animation

abstraction

Exploratory Spec 1

Exploratory Spec 2

Exploratory Spec 3

animation

Simulated Program 3

valid solution

solution space 5
rigorous specification

abstraction

valid solution

solution space

Exploratory Spec 1

Exploratory Spec 3

Rigorous Spec

Exploratory Spec 2
design and implementation

valid solution
Exploratory Modeling

At the beginning of a development, specification is **incomplete**

domain knowledge is **fragmented**

**limited shared understanding** among stakeholders

We learn through exploration over the problem domain by writing/analyzing/animating incomplete specifications
When do we use code generators?
Conventional use of Code Generator: Automated Implementation

Exploratory Spec 1
Exploratory Spec 2
Exploratory Spec 3
Rigorous Spec
Refined Spec
Final Program Code
valid solution
code generation
abstraction
solution space

10
Exploratory use of Code Generator: Animation by transpiler instead of interpreter
Transpiler vs Interpreter

Pros

• Performance
• Can modify generated code
• Linking and testing with existing implementation

Cons

• Debug (step execution, source level debugging)
Requirements to code generators for exploration (1/2)

**Automatic compilation and execution** of the generated source code because you'll run it often.

**Compilation and execution** of the generated source code must be controlled **in the same IDE** as the exploratory modeling because it should be "integrated" in the environment.

**Few limitations** on the specification language for automated code generation.

Don't bend the specification!
Requirements to code generators for exploration (2/2)

**Debug capability** must be enabled because the specification is incomplete.

**Understandable by stakeholders** with no formal methods background
Use GUI builders or visualization tools

**Permissive checking** by choice

**Continuous analysis**
because the specification is incomplete.
ViennaTalk's code generators

• VDM-SL to Smalltalk

• generate, compile, and run smalltalk code by just one click,

• able to turn on/off type checking and assertion checking,

• use classes and objects in Smalltalk's standard library so that it looks like a hand coded Smalltalk program,

• generate human readable/modifiable code, and

• generate a script, a class library or objects from VDM-SL spec
Demo

Tic-Tac-Toe
Performance benchmark: Prime numbers

**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 := []);`
operations

next : () ==> [nat1]
next() ==
cases space:
   [x] ^ - - ->
      (primes := primes ^ [x];
      sieve(x);
      return x),
   others -&gt; return nil
end;

sieve : nat1 ==> ()
sieve(x) ==
space := [space(i) | i in set inds space &amp; space(i) mod x &lt;&gt; 0];
operations

prime10000 : () ==> seq of nat1

prime10000() ===
    (setup(10000);
      while next() <> nil do skip;
      return primes);

19
## Performance of VienntaTalk's code generator

<table>
<thead>
<tr>
<th>Tool</th>
<th>Interp/CG</th>
<th>Impl Lang</th>
<th>Time (Overture CG=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDMTools</td>
<td>Intp</td>
<td>C++</td>
<td>79.6</td>
</tr>
<tr>
<td>VDMJ</td>
<td>Intp</td>
<td>Java</td>
<td>19.1</td>
</tr>
<tr>
<td>ViennaTalk</td>
<td>CG</td>
<td>Smalltalk(Script)</td>
<td>3.45</td>
</tr>
<tr>
<td>VDMTools</td>
<td>CG</td>
<td>C++</td>
<td>1.22</td>
</tr>
<tr>
<td>Overture tool</td>
<td>CG</td>
<td>Java</td>
<td>1.00</td>
</tr>
<tr>
<td>ViennaTalk</td>
<td>CG</td>
<td>Smalltalk(Class)</td>
<td>0.729</td>
</tr>
<tr>
<td>ViennaTalk</td>
<td>CG</td>
<td>Smalltalk(Object)</td>
<td>0.700</td>
</tr>
</tbody>
</table>
Performance of VienntaTalk (with Pharo5)'s CG

<table>
<thead>
<tr>
<th>Tool</th>
<th>Interp/CG</th>
<th>Impl Lang</th>
<th>Time (Overture CG=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDMTools</td>
<td>Intp</td>
<td>C++</td>
<td>79.6</td>
</tr>
<tr>
<td>VDMJ</td>
<td>Intp</td>
<td>Java</td>
<td>19.1</td>
</tr>
<tr>
<td>ViennaTalk</td>
<td>CG</td>
<td>Smalltalk (Script)</td>
<td>2.26 (was 3.45 w/ Pharo4)</td>
</tr>
<tr>
<td>VDMTools</td>
<td>CG</td>
<td>C++</td>
<td>1.22</td>
</tr>
<tr>
<td>Overture tool</td>
<td>CG</td>
<td>Java</td>
<td>1.00</td>
</tr>
<tr>
<td>ViennaTalk</td>
<td>CG</td>
<td>Smalltalk (Class)</td>
<td>0.639 (was 0.729 w/ Pharo4)</td>
</tr>
<tr>
<td>ViennaTalk</td>
<td>CG</td>
<td>Smalltalk (Object)</td>
<td>0.608 (was 0.700 w/ Pharo4)</td>
</tr>
</tbody>
</table>
Why Smalltalk often run faster than C++/Java?

In general, Smalltalk is 2~10 time slower than Java if they run the same algorithm.

...but
Example: Prime number

functions

\[
\text{isPrime : nat1} \rightarrow \text{bool}
\]

\[
\text{isPrime}(x) = \text{x} \not\equiv 1 \text{ and } (\text{forall } y \text{ in set } \{2, \ldots, x - 1\} \& \text{x} \mod y \not\equiv 0);
\]

what will happen if you evaluate \text{isPrime(20161107)}?
Example: Prime number

functions

\[
\text{isPrime} : \text{nat1} \rightarrow \text{bool}
\]

\[
\text{isPrime}(x) = \quad x \not< 1 \text{ and } (\forall y \in \{2, \ldots, x - 1\} \text{ and } x \mod y \not= 0);
\]

what will happen if you evaluate \text{isPrime}(20161107)?

This set eats really big memory!
Smalltalk code from

\[
\text{forall } y \text{ in set } \{2, \ldots, x - 1\} \& x \mod y \neq 0
\]

\[
[:_\text{forall} \mid _\text{forall} \text{ allSatisfy: } [ :y \mid x \not\equiv y \equiv 0 ] ]
\]

value: (2 to: x - 1)

This \textit{interval} object holds only two integers.
Why Smalltalk often run faster than C++/Java?

VDM specs defines the problem, not the algorithm.

A language with high abstraction and rich library provides

• a good selection of algorithms and

• their fine implementations.

Using objects and classes from the standard library naturally brings the above benefits to the generated code.
Summary

Code generators for exploratory modeling

- Shall be **handy to use** … all-in-one reflective environment
- Shall have **less limitation** to the spec … as less as interpreters
- Shall do **flexible checking** … as flexible as interpreters
- Shall run **fast** … with **rich and abstract** language/libraries

Future work

- Shall **debug** the VDM spec … step executions with VDM-SL source
- Shall support **testing**