Newcastle University

COMPUTING SCIENCE

Proceedings of the 12th Overture Workshop,
Newcastle University, 21 June, 2014

N. Battle and J. S. Fitzgerald (Eds.)

TECHNICAL REPORT SERIES

No. CS-TR-1446 **January 2015**

TECHNICAL REPORT SERIES

No. CS-TR-1446 January, 2015

Proceedings of the 12th Overture Workshop, Newcastle University, 21 June, 2014

N. Battle and J. S. Fitzgerald (Eds.)

Abstract

This volume contains papers and abstracts of presentations given at the 12th Overture Workshop on the Vienna Development Method (VDM), which took place at Newcastle University in June 2014. VDM is one of the longest established formal methods, having its origins in compiler development work in IBM in the 1970s. The community-based Overture initiative is developing robust tools for VDM on an open platform that has been successfully adapted for collaborative modelling and simulation in embedded systems design, and latterly SoS modelling, verification and testing.

The 12th workshop reflected the breadth and depth of work in VDM. Contributions included reports of current work on fundamental approaches to reasoning about concurrency, design space exploration, the use of tools in teaching, and of course tools (interpreter design, code generation and the maturing architecture of Overture itself). In this report we include papers, extended abstracts for four of the talks given, and short abstracts of the remainder.

Bibliographical details

BATTLE, N., FITZGERALD, J.S.

Proceedings of the 12th Overture Workshop, Newcastle University, 21 June, 2014
[By] N. Battle and J. S. Fitzgerald (Eds.)

Newcastle upon Tyne: Newcastle University: Computing Science, 2015.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1446)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1446

Abstract

This volume contains papers and abstracts of presentations given at the 12th Overture Workshop on the Vienna Development Method (VDM), which took place at Newcastle University in June 2014. VDM is one of the longest established formal methods, having its origins in compiler development work in IBM in the 1970s. The community-based Overture initiative is developing robust tools for VDM on an open platform that has been successfully adapted for collaborative modelling and simulation in embedded systems design, and latterly SoS modelling, verification and testing.

The 12th workshop reflected the breadth and depth of work in VDM. Contributions included reports of current work on fundamental approaches to reasoning about concurrency, design space exploration, the use of tools in teaching, and of course tools (interpreter design, code generation and the maturing architecture of Overture itself). In this report we include papers, extended abstracts for four of the talks given, and short abstracts of the remainder

About the editors

Nick Battle is a Senior Designer at Fujitsu UK and has spent the last 30 years working in the computer industry. He wrote the VDMJ tool, which has been extended and integrated with other work to create the Overture Tool for working with all dialects of VDM. He continues to take an active interest in the adoption of formal development methods in industry.

John Fitzgerald is Professor of Formal Methods in Computing Science at Newcastle University, UK, where he jointly leads research on advanced model-based engineering. He recently led the COMPASS consortium developing methods and tools for engineering Systems of Systems, and from 2015 will lead Newcastle's work on four new research and innovation projects on collaborative design of Cyber-Physical Systems (CPS). He is currently establishing a new CPS laboratory, and leading the development of former industrial land in the heart of the city as a new centre for Computing Science, and for multidisciplinary research on digitally enabled urban sustainability.

Suggested keywords

FORMAL METHODS
VDM
SOFTWARE ENGINEERING
SYSTEMS ENGINEERING

Proceedings of the 12th Overture Workshop
Newcastle University
21 June, 2014

The 12th in the "Overture" series of workshops on the Vienna Development Method (VDM), its associated tools and applications, was held in association with the COMPASS[1] project at Newcastle in June 2014. The workshop aimed to identify and encourage new collaborative research, and to foster current strands of work towards new projects and publications.

VDM is one of the longest established formal methods, having its origins in compiler development work in IBM in the 1970s. In the 1990s, the basic VDM modelling language was standardized by ISO and the first commercial tools emerged. Since 2000, the method has been extended to support object-orientation, concurrency, real-time and distribution. Advances in these areas led to the development of new technology for the design of embedded systems based on collaborative modelling and co-simulation[2]. A notable recent development has been the very successful combination of VDM with Circus as a basis for the COMPASS Modelling Language (CML) – the first formal modelling language developed specifically for systems of systems (SoSs).

Research in VDM and Overture is driven by the need to develop industry practice as well as fundamental theory. Consequently, the provision of tool support has been a priority for many years. The community-based Overture[3] initiative is developing industry-strength tools on an open platform that has been successfully adapted to form platforms for co-modelling and co-simulation in embedded systems design (Crescendo[4]), and latterly SoS modelling, verification and testing (Symphony[5]).

The 12th workshop reflected the breadth and depth of work in VDM. Contributions covered topics as diverse as fundamental approaches to reasoning about concurrency, design space exploration, the use of Crescendo in teaching, and of course tools (interpreter design, code generation and the maturing architecture of Overture itself). In this report we include papers extended abstracts for four of the talks given, and include the abstracts of the remaining. Contributed presentations are to be found in the Overture Wiki[6].

We are grateful to the School of Computing Science at Newcastle University for its kind hospitality in hosting the workshop for the fourth time in the fifteen year history of the series.

Nick Battle

John Fitzgerald

[1] Comprehensive Modelling for Advanced Systems of Systems (www.compass-research.eu)
[2] Fitzgerald JS, Larsen PG, Verhoef MHG, eds. *Collaborative design for embedded systems: co-modelling and co-simulation*. Berlin: Springer, 2014.
[3] Overture: www.overturetool.org.
[4] Crescendo: www.crescendotool.org.
[5] Symphony: www.symphonytool.org.
[6] 12th Overture Workshop Wiki: http://wiki.overturetool.org/index.php/12th_Overture_Workshop.

Contents

Interpreting Implicit VDM Specifications using ProB

Kenneth Lausdahl[1], Hiroshi Ishikawa[2]
and Peter Gorm Larsen[1]

[1] Department of Engineering, Aarhus University,
Finlandsgade 24, DK-8200 Aarhus N, Denmark
[2] Department of Information and Culture,
Niigata University of International and Information Studies
3-1-1 Mizukino, Nishi-ku, Niigata, 920-2292, Japan

Abstract. Modelling of software can with advantage start by the creation of an initial software design at a high-level of abstraction expressed as implicit specifications. The Vienna Development Method is a formal method that support different levels of abstraction including such implicit specification mechanisms. However, most of the existing tool-support for VDM does not support this style of modelling adequately from an analysis perspective. In this paper we demonstrate how such implicit specifications can be made interpretable through the use of the ProB constraint solver allowing them to be validated like explicit specifications. We shown how an internal translation to ProB is made and how this integrates with the existing VDM interpreter from Overture.

1 Introduction

A traditional approach to model software is to start with creating an initial software specification at a high level of abstraction expressed in an implicit style. The Vienna Development Method (VDM) [3,8] can model at various levels of abstraction by using implicit and/or explicit descriptions to define functionalities.

Operations and functions written in explicit style include descriptions about how a calculation from input to output can be carried out. Therefore such specifications can be executed and validated by using Overture tool (and using VDMTools). Because of the pragmatic tool support (strong on simulation of executable models) this means that the majority of VDM models in the last decade have used the executable subset of the language [11,14]. Excitability has been an academic discussion for a number of years [7,6]. However, operations and functions written in implicit descriptions include descriptions does not show *how* they work but what constraints they shall satisfy. The work presented in this paper targets a change in this trend towards more implicit-style models.

The structure of this paper is as follows: Section 2 describes the overview of VDM and its tools. Section 3 describes the ProB constraint solver that is cooperated with VDM to execute implicit style specifications. Then Section 4 provides the translation rules used for the interpreter integration. Section 5, demonstrates how to translate expressions using a few concrete examples. Based on this work, a small case study, soccer referee's book, is shown in Section 6. The specification is described both implicit and explicit style. We then deal with the description in implicit style. This is ongoing work so there are implicit style descriptions where our proposed framework does not yet work as presented in Section 7. In Section 8, some related works are provided. Finally, in Section 9, we will conclude this work and show future work.

2 The Vienna Development Method

The Vienna Development Method (VDM) is one of the longest established model-oriented formal methods, and was originally developed at the IBM laboratories in Vienna in the 1970's. The VDM specification Language (VDM-SL) is a higher-order language which is standardized by the International Organization for Standardization (ISO), and has a formally defined syntax, and both static and dynamic semantics [12,19].

Models in VDM are based on data type definitions built from simple abstract data types using booleans, natural numbers, characters, tokens and type constructors for record, product, union, map, set and sequences. Type membership may be restricted by predicate invariants. Persistent state is defined by means of typed variables, again restricted by invariants (and potentially initialized). Operations that may modify the state can be defined not only explicitly by using imperative statements but also implicitly by using standard pre- and post-condition predicates.

Other dialects of VDM has also been produced and here VDM++ [4] and VDM Real-Time (VDM-RT) [20]. The work presented here is reusable for these other dialects as well, but in the case study we present here we limit ourselves to the VDM-SL notation. All of these dialects are supported by the open source Overture tool [10] and VDM-SL and VDM++ are supported by VDMTools [9].

3 The ProB as a Constraint Solver

ProB [16,17] is an animator and model checker for the B-Method [1]. The constraint-solving capabilities of ProB supports: model finding, deadlock checking and test-case generation. The core of ProB is implemented in Prolog, and supports multiple platforms including a Java API used in this work. The ProB Logic Calculator is able to evaluate arbitrary expressions and predicates using the B syntax. This includes support for predicate logic, set theory, and simple arithmetic constraints.

ProB can deal with basic data types: Booleans, Sets, Numbers, Relations, Sequences, Records and Strings. Booleans are defined in terms of TRUE and FALSE but these can only be used in predicated if specified as:

$$x = \text{TRUE} \wedge y = \text{TRUE}$$

a predicate such as $z > 0$ can be converted into a boolean using $bool(z > 0)$. Sets are described using { } and includes operators such as set-comprehensions and $POW(S)$ etc. Numbers includes integers ($\mathbb{Z}$) and naturals ($\mathbb{N}$). Relations includes the standard operators for dealing with relations and specified as $\{a \rightarrow b\}$. Sequences are denoted by [ ] and includes basic operators to get the first, last, tail etc. Records are structs with named and typed fields and are specified as $struct(ID \in S, \cdots ID \in S)$ and constructed using $rec(ID \in E, \cdots ID \in E)$. Finally, strings are specified as STRING containing all possible strings.

The animator operates on a *machine* which can define sets, definitions, constants, properties, variables, invariants, initialization and operations. However, in the context of this works only the sets matter. These define enumerated elements that may be used to represent quotes etc. An example to define a set will be shown in Section 5.1.

4 Extending the VDM Interpreter

Validation of VDM specifications can be done using the VDM interpreter [14] if the specification is specified in an explicit style. However, if the more abstract style is used there is only limited support for any analysis. Therefore, it is beneficial to extend the interpreter such that it is able to interpret an implicitly defined specification.

The person registry specification shown below is an implicit specification with a type definition of a person and a state holding the registry. The *Add* operation adds a new person based on his name, and checks that the new person gets a new social security number. It can be observed that the *Add* operation only has a post-condition and no explicit body and therefore it cannot be interpreted by the interpreter.

$String = Char^*$

$$\begin{aligned} Person :: \quad ssn &: \mathbb{Z} \\ name &: String \end{aligned}$$

$Registry :: persons : Person\text{-}\mathbf{set}$

$\mathbf{init}\ persons = \{mk\text{-}Person(1, \texttt{"Jane Doe"})\}$

$Add\ (name\colon String)$

$\mathbf{ext\ wr}\ persons : Person\text{-}\mathbf{set}$

$$\mathbf{post}\ mk\text{-}Person(ssn, name) \in persons \land \forall p\cdot \in \overleftarrow{persons} \land p.ssn \neq ssn \land \\ persons = \overleftarrow{persons} \cup \{mk\text{-}Person(ssn, name)\}$$

The interpreter evaluates a call to $Add(\texttt{"John Doe"})$ as shown on the left in Figure 1. Since the *Add* operation in this case does not have a body then the evaluation of the body cause the interpretation to be aborted. The approach in this paper hooks in an additional step just before the body is evaluated as illustrated on the right in Figure 1.

Here the VDM specification is examined and the pre-state and post-state (denoted with _) are typed in ProB as shown in List. 1 at line 1 and 2. Then the current state of the model is constrained (line 3), and the translation of the post-condition itself which describes the behaviour of the operation (line 4-6). Finally, the arguments to the operation must constrain the argument (line 7).

```
1  per:POW(struct(ssn:INT, name:STRING)) &
2  per_:POW(struct(ssn:INT, name:STRING)) &
3  per={rec(ssn:1, name:"Jane Doe")} &
4  rec(ssn:k, name:g):per_ &
```

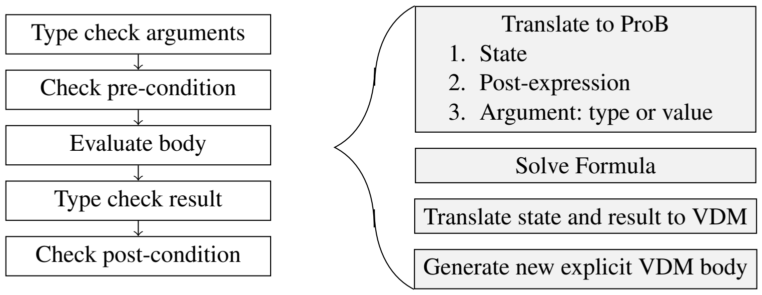

Fig. 1: Interpreter execution of implicit functions and operations.

```
!(p).(p:per => p.ssn /=k) &
per_ = per \/ {rec(ssn:k, name:g)} &
g="John Doe"
```

List. 1: ProB input formula for the *Add* operation in the person register specification.

This forms a formula that ProB may be able to solve if a solution exists within the limitations from the solver. If a solution is found like shown in List. 2, then this must be converted back to VDM. This is then done with a type lookahead based on the state type definition or the declared return type from the operation itself.

```
Solution:
       ( per = {} &
         per_ = {rec(ssn:1,name:"Jane Doe"),
                  rec(ssn:-128, name:"John Doe")} &
         per = {rec(ssn:1,name:"Jane Doe")} &
         k = -128 &
         g = "John Doe" &
         per_ = {rec(ssn:-128,name:"John Doe")} &
         [p,ssn] = 0 )
```

List. 2: ProB solution for the *Add* operation with the argument `"John Doe"`.

Note that in this case the *Add* operation allowed read/write access to the *persons* field of the state but it did not state that any existing persons should be preserved.

Finally these new VDM values are combined into an explicitly defined expression that the interpreter can evaluate. The interpreter then continues its normal execution with this custom body. The body is then evaluated and the resulting changes are type checked and finally assigned to the state and return value from the initial call. If an error occurs in this extension then the interpreter will catch it, because it checks typing, invariants and the post-condition before the new values are accepted into the normal execution of the interpreter.

5 Translation between VDM and ProB

This section describes how types, definitions and expressions are translated between VDM and ProB.

5.1 Translating Types

This section will describe how a subset of the types from the VDM language can be expressed in ProB. The translation does not cover the product, optional, object and function types which are left for future work.

Basic Types The basic types in VDM consists of: boolean ($\mathbb{B}$), numeric ($\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$, and $\mathbb{N}1$), character, quotes, and tokens. Some of these have a one-to-one translation: `BOOL`, `INT`, `NAT`, `NAT1`, but others like $\mathbb{R}$ are not possible to present, and characters are not directly representable, both of these are not covered here.

The token type in VDM is a type that wraps around any type hiding the inner value and only allows comparison. This means that a token value is represented by its inner value-type which is the type of the argument to the token constructor $mk\text{-}token(1)$, in this case $typeof(1) \in \mathbb{N}1$. However, if an expression like $a \in token$ is given with no token constructor then it is impossible to calculate the type of the value the token may have, and therefore the smallest type it can have is a union type of all possible types in VDM. However, a translation that scans the specification for $mk\text{-}token$ expressions and constructs a union type of only these types will be sufficient to represent the token type. If a specification does not include any $mk\text{-}token$ expressions then the token type can be considered as undefined and there for any type could be chosen e.g. `INT`:

$$\begin{aligned}
&\exists mkt \in specification \cdot mkt \in mk\text{-}token \;\Rightarrow \\
&\quad typeof(token) \equiv union\text{-}type(\{mkt \mid mkt \in specification \cdot mkt \in mk\text{-}token\}) \\[1em]
&\neg\, \exists mkt \in specification \cdot mkt \in mk\text{-}token \;\Rightarrow \\
&\quad typeof(token) \equiv \mathtt{INT}
\end{aligned} \tag{1}$$

The proposed translation with union types may only be used if all arguments to $mk\text{-}token$ has a type different from tuple and map since these will be represented in the same way in ProB making the translation of the result to VDM impossible. Consider the following two expressions shown the representation in VDM and ProB:

$$\begin{array}{ll}
\text{VDM:} & mk\text{-}token(\{1 \mapsto 2\}) \neq mk\text{-}token((1,2)) \\
\text{ProB:} & \qquad\qquad \mathtt{1|\text{-}>2\ =\ (1,2)}
\end{array} \tag{2}$$

The quote type can be represented in ProB as an enumerated set of all quotes in the specification. We can make use of a basic machine that defines a single enumerated set S has the form:

MACHINE <Machine_Name> **SETS** $S = \{A, B\}$ **END**

Consider the following example:

$$q \in \text{DENMARK} \mid \text{GERMANY} \tag{3}$$

This can then be converted into ProB with a enumerated QUOTES set added to the machine defined as

$$\textbf{MACHINE}\ \texttt{CountryName}\ \textbf{SETS}\ \mathit{QUOTES} = \{\mathit{Denmark}, \mathit{Germany}\}\ \textbf{END}$$

and with the expression:

$$q \in \{\mathit{Denmark}, \mathit{Germany}\} \tag{4}$$

and a single fixed value of q can be specified as $q = \mathit{Denmark}$. Note that quote types in VDM almost always are used as union types similar to enumerations in other languages.

Named Invariant Types and Records A named invariant type and records are named types that can have invariants. Since types cannot be declared by name in ProB the types must be represented by their definition alone. This, for named invariant types means that the type they rename will be used directly, and annotated with the invariant. This means that a type $B = \mathbb{N}$ with the invariant $\mathit{mk}\text{-}B(x) \triangleq x > 100$ can be defined in ProB as $x{:}\, \mathbb{N} \wedge x > 100$.

The record type is a composite type that also requires the addition of the predicate from the invariant whenever it is constructed. A record type is also used to represent module state as explained in Section 5.2. A translation to ProB is possible since it also contains the notion of records named **`struct`** for typing and **`rec`** for value creation. Below is shown a VDM record on the left an the corresponding ProB representation on the right with a variable a declared of the record type:

$$\begin{array}{lll} R :: a : \mathbb{N} & \quad & r{:}\, \mathit{struct}(a{:}\, \mathit{NAT}, b{:}\, \mathit{NAT}) \wedge \\ b : \mathbb{N} & & r'a > r'b \end{array} \tag{5}$$

where

$$\mathit{inv}\text{-}R(\mathit{mk}\text{-}R(a, b)) \quad \triangleq \quad a > b$$

The translation becomes less trivial when multiple records are combined if they all include invariants:

$$R :: a : \mathbb{N}$$
$$ b : B$$

where

$$\mathit{inv}\text{-}S(\mathit{mk}\text{-}S(a, b)) \quad \triangleq \quad a > b.x$$

$$B :: x : \mathbb{N}$$

where

$$\mathit{inv}\text{-}B(\mathit{mk}\text{-}B(x)) \quad \triangleq \quad x > 100$$

The ProB representation of the records above is shown below. Note that these additional invariant predicates will have to be included in all places where the record is used:

$$\begin{aligned}&r\colon struct(a\colon NAT, b\colon struct(x\colon NAT))\land\\&r'a > r'b'x\land\\&r'b'x > 100\end{aligned} \tag{6}$$

Strings The VDM language does not have a special representation of strings as such, but represents it as a $[Char]$, whereas ProB has a special `STRING` type denoting all possible strings. This makes it possible to translate the VDM type [Char] into the ProB type `STRING`.

Product Type Product types are semantically the same in both VDM and ProB, only the syntax differs where VDM uses a product constructor $mk\text{-}(...)$ and ProB just uses $(...)$ to define the pairs.

5.2 Translating Definitions

Definitions are used to define functions, operations and a notion of a state. A translation of all definitions in a specification is not needed but operations always require the state definition to be translated.

State State in VDM specifications are defined as the definitions that operations can access. This for VDM-SL is a record as described in Section 2 and for VDM++ is a number of named definitions with a type. The record in VDM-SL may also have an optional initializer, which must not be included in the translation if used with an operation since this has the same effect as specifying the arguments of the operation itself making it unsolvable. Since operations may refer to both pre- and post-state both must be made available in ProB. This is done by translating all state definitions twice with a new and old name.

For classes in VDM++ a specification with an instance variable a may be translated like: $a \in T$ inv $a > 1$ to $(a \in T \land a > 1) \land (\overleftharpoon{a} \in T \land \overleftharpoon{a} > 1)$. Note that the class invariant of the type must be added and that both the current and old state must be typed and constrained by the invariant.

For modules in VDM-SL the same essentially applies but a state is a single record. Thus a state S defined as:

$$\begin{aligned}S \;::\; &a : \mathbb{N}\\ &b : \mathbb{N}\end{aligned}$$

where

$$inv\text{-}S(mk\text{-}S(a, b)) \;\triangleq\; a > b$$

can be translated to ProB as:

$$\begin{array}{ll} \overleftharpoon{r} : struct(a\colon NAT, b\colon NAT) \wedge & (1) \\ r\colon struct(a\colon NAT, b\colon NAT) \wedge & (2) \\ \overleftharpoon{r} = rec(a\colon 2, b\colon 1) \wedge & (3) \\ (r'a > r'b \wedge \overleftharpoon{r}'a > \overleftharpoon{r}'b) & (4) \end{array} \tag{7}$$

The equation above shows how S can be translated. Line 1 is the old state, line 2 is the current/new state, and line 3 is the state initializer[3]. Line 4 is the invariant, note that this has to be added both for the old and new state.

Function and Operation Definitions A function or an operation definition is not translated into ProB but used when the VDM interpreter wish to execute these definitions when declared in an implicit style where the body is left unspecified. The translation will then use the argument and their types in the ProB formula together with the expression from the post-condition as shown in Section 4. The result from ProB must be converted back to an explicit VDM body for the definition.

The Add operation from Section 4 will then result in a body of the form:

$$(atomic(persons\colon= \{mk\text{-}Person(\text{-}128, \texttt{"John Doe"})\});) \tag{8}$$

if the operation has a return value declared then if the solved value if 0 then this is represented by $return\ 0$ at the end of the body. The translation for functions are similar except that it does not specify the $atomic$ block used for assigning state, and the return statement but only includes the resulting expression itself.

The limitation with this approach is that post-conditions that includes operation or function calls cannot be translated. However, if these calls are not recursive there is the possibility for adding a mechanism for in-lining these definitions as well.

5.3 Expressions

This section shown how the expressions used in post-conditions of functions or operations can be translated to ProB. The translation rules uses $[\![]\!]$ to denote the application of the translation rules and $\equiv$ to show the equivalence between VDM and ProB notation. Many of the basic expressions from the VDM language has the same definition in ProB while others like:

$$\begin{array}{ll} \textbf{abs}\ x & \equiv \texttt{max}\{\text{-}[\![x]\!], [\![x]\!]\} \\ \textbf{inds}\ [1,2,3] & \equiv 1\texttt{..size}([\![[1,2,3]]\!]) \end{array} \tag{9}$$

[3] This must only be added if it is not going to be used for evaluation since the initializer must be replaced with a constraint of the current value.

Quantifiers The universal and existential quantifiers maps directly to ProB as shown in Eq. 10 and Eq. 11:

$$\forall x_1 \in S_1, \cdots, x_n \in S_n \wedge pred \equiv \textbf{!}\,(x_1, \cdots, x_n).(x_1 \in [\![S_1]\!] \wedge ... \wedge x_n \in [\![S_n]\!] \Rightarrow [\![pred]\!]) \quad (10)$$

$$\exists x_1 \in S_1, ..., x_n \in S_n \cdot pred \equiv \textbf{\#}(x_1, ..., x_n).(x_1 \in [\![S_1]\!] \wedge \cdots \wedge x_n \in [\![S_n]\!] \wedge [\![pred]\!]) \quad (11)$$

However, the unique existential quantifier ($\exists!$) from VDM does not have a corresponding definition in ProB but can easily be defined universal and existential quantifiers in VDM itself. An example of the unique existential quantification in VDM is shown in Eq. 12 and Eq. 13 shows the rewritten form using both universal and existential quantifiers:

$$\exists!\, x \cdot \in \{1, 2, 3\} \cdot x < 2 \quad (12)$$

$$\exists x \in \{1, 2, 3\} \cdot x < 2 \wedge \forall y \in \{1, 2, 3\} \cdot y < 2 \Rightarrow x = y \quad (13)$$

The unique existential quantification can also be expressed using set operators as shown in Eq. 14:

$$\textbf{card}\,(\{x \mid x \in [\![S]\!] \cdot [\![pred]\!]\}) = 1 \quad (14)$$

Sets and Set Operators The ProB notation for sets is the same as in VDM and therefore a VDM set can be translated as shown in Eq. 15 where each of the set elements are translated:

$$S \equiv \begin{cases} \{\,\} & \text{if } S = \{\,\} \text{ (empty set).} \\ \{[\![e_1]\!], [\![e_2]\!], ..., [\![e_n]\!]\} & \text{if } S = \{e_1, e_2, ..., e_n\}. \end{cases} \quad (15)$$

The simplest for a VDM set-comprehension is defined using an expression for the result, a binding and a predicate to restrict the binding:$\{exp \mid bind \cdot pred\}$. This differs from the ProB definition that has a number of identifiers and a predicate, where each identifier must be present in the predicate: $\{ids \mid pred\}$. The translation shown in Eq. 16 introduces a new identifier for the element that represents the value created by the expression ($x+1$). This identifier is then set equal to the translation of the expression ($[\![x+1]\!]$) where the identifier (x) is obtained from the binding. The binding is translated into existential quantification over the set and restricted according to the predicate ($[\![x > 2]\!]$):

$$\{x + 1 \mid x \in S \wedge x > 2\} \equiv \{x' \mid \#(x).(x \in S \wedge x > 2 \wedge x' = x + 1)\} \quad (16)$$

Maps and Map Operators ProB does not have maps but relations that supports all map operations from VDM. A map can be translated to ProB using the same approach as used for sets where each maplet is translated individually. The map-comprehensions ($\{maplet \mid bind \cdot predicate\}$) is defined like the set-comprehension (Eq. 16) in ProB and therefore the same solution can be used. While the set-comprehension only have a single identifier specified map-comprehensions have multiple like: a, b which corresponds to $a \mapsto b$.

A map value can be thought of as an unordered collection of pairs. The first element in the pair is called a key. All key elements in a map must be unique. If a relation generated from a set-comprehension expression holds this property, the relation can be regarded as a map. Therefore it is necessary to add the following condition at the predicate part of the set-comprehension expression.

$$\begin{array}{l} \forall (k1, v1) \in S, (k2, v2) \in S \cdot k1 = k2 \ \Rightarrow \ v1 = v2 \\ \quad \text{where } S \text{ is a relation (a set of pairs).} \end{array} \tag{17}$$

Eq. 18 shows how a VDM map comprehension can be translated using this approach. The comprehension creates a map with the following elements: $\{9 \mapsto 2, 16 \mapsto 2\}$:

$$\begin{array}{l} \{b * b \mapsto a \mid a \in \{1,2\} \wedge b \in \{3,4\} \cdot a > 1\} \equiv \\ \quad \{pair \mid !(a,b,c,d).((a,b) \in r \wedge (c,d) \in r \wedge a = c \ \Rightarrow \ b = d) \wedge pair \in r\} \\ \quad \text{where} \\ \quad r = \{bb, aa \mid \#(a,b).(a \in \{1,2\} \wedge b \in \{3,4\} \wedge a > 1 \wedge bb = b * b \wedge aa = a)\} \end{array} \tag{18}$$

Sequence and Sequence Operators A VDM sequence can be translated to ProB in a similar faction as sets using the ProB [ ] operator as shown in Eq. 15. While ProB has similar sequence operators none or these have been taken into account yet in this work. Sequence-comprehensions are not supported in ProB and are therefore not included here.

$$S \equiv \begin{cases} [\,] & \text{if } S = [\,] \text{ (empty sequence).} \\ [[\![e_1]\!], ..., [\![e_n]\!]] & \text{if } S = [e_1, ..., e_n]\,. \end{cases}$$

Conditions The translation of the VDM if-expression has only been partly specified as for the cases where $typeof(exp1) \in \mathbb{B} \wedge typeof(exp2) \in \mathbb{B}$:

$$\textbf{if } test \textbf{ then } exp1 \textbf{ else } exp2 \equiv (\neg [\![test]\!] \vee [\![exp1]\!]) \wedge ([\![test]\!] \vee [\![exp2]\!]) \tag{19}$$

6 Case Study – A Soccer Specification

This section illustrates the new ability to interpret implicit VDM definitions can be used for a soccer specification [15] that originally was created by Yves Ledru.

The specification models the rules for the substitution of players during a soccer game. The purpose of the example is to model rules and to check whether the actions quoted below follow the rules. The reason for developing this specification was to illustrate that a violation to the rules was made during the 1994 World Cup in a match between Italy and Norway. To model this particular problem, a further rule is needed, that is, that the referee may exclude one of the players (including the substitutes).

The type `player` is introduced as a renaming for natural numbers and to denote the number of players.

$$player = \mathbb{N}_1$$

The state of the referee's book is defined as a Record named R_Book.

$$\begin{array}{lrcl} R\_Book :: & on\_field\_players & : & player\text{-}\mathbf{set} \\ & potential\_substitutes & : & player\text{-}\mathbf{set} \\ & goalkeeper & : & player \\ & nb\_gk\_subs & : & \mathbb{N} \\ & nb\_fp\_subs & : & \mathbb{N} \end{array}$$

$$\begin{array}{l} \mathbf{inv}\ (mk\text{-}R\_Book(ofp, ps, gk, ngk, nfp)) \triangleq (\mathbf{card}\ opf) \leq 11 \wedge \\ \quad (ngk \leq gk\_subs\_max) \wedge \\ \quad (nfp \leq fp\_subs\_max) \wedge \\ \quad gk \notin ps \wedge ofp \cap ps = \{\,\} \end{array}$$

$$\begin{array}{l} \mathbf{init}\ ofp = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\} \wedge \\ \quad ps = \{12, 13, 14, 15, 16\} \wedge \\ \quad gk = 1 \wedge ngk = 0 \wedge nfp = 0) \end{array}$$

where `player` is a type defined as the type **`nat1`**. The `R_Book` has numbers of on field players, numbers of potential substitutes, the number of goalkeeper, the number of goalkeeper substitutions already performed, and the number of field players substitutions already performed respectively. These data have to modify by operations.

The initial state has values as the number of `on_field_player` is eleven, the initial number of `potential_substitutes` is five, the goalkeeper has number 1, the number of goalkeeper substitutions is 0, and the initial number of field player substitutions is also 0.

The implicit definition of the interesting operations are presented below. The operation `RED_CARD` takes a number of player who has to exclude. If a player gets a red card, the player may be any of the team players, that is, he/she is in one of both `on_field_players` and `potential_substitutes` (the pre-condition). The post-condition states that the player is no longer in both sets.

The operation `CHANGE_GOALKEEPER` takes a player number as an argument and switch the player to a new goalkeeper. In this case, the set of `on_field_players` is referred whether the player is on the field, and the variable `goalkeeper` is updated.

The operation `SUBSTITUTION` takes two arguments, that is, the substitution of a player by another. The pre-condition states that the outgoing player is on the field, the substitute is in the set `potential_substitutes`, and the constraints on the maximum numbers of players. The post-condition states that `on_field_players`, `potential_substitutes`, and `nb_gk_subs` or `nb_fp_subs` are updated.

$$
\begin{array}{l}
\mathit{SUBSTITUTION}\ (pl\colon player, subs\colon player)\\
\textbf{ext wr}\ on\_field\_players \quad : \ player\textbf{-set}\\
\quad \textbf{wr}\ potential\_substitutes \ : \ player\textbf{-set}\\
\quad \textbf{wr}\ goalkeeper \quad : \ player\\
\quad \textbf{wr}\ nb\_gk\_subs \quad : \ \mathbb{N}\\
\quad \textbf{wr}\ nb\_fp\_subs \quad : \ \mathbb{N}
\end{array}
$$

$$
\begin{array}{l}
\textbf{pre}\ \cdots\\
\textbf{post}\ on\_field\_players = \overleftarrow{on\_field\_players} \cup \{subs\}/\{pl\}\\
\qquad \wedge\ potential\_substitutes\\
\qquad\quad = \overleftarrow{potential\_substitutes}/\{subs\}\\
\qquad \wedge\ \textbf{if}\ pl = \overleftarrow{goalkeeper}\\
\qquad\quad \textbf{then}\ ((goalkeeper = subs) \wedge (nb\_gk\_subs = \overleftarrow{nb\_gk\_subs} + 1)\\
\qquad\qquad\qquad \wedge\ (nb\_fp\_subs = \overleftarrow{nb\_fp\_subs}))\\
\qquad\quad \textbf{else}\ ((goalkeeper = \overleftarrow{goalkeeper}) \wedge (nb\_gk\_subs = \overleftarrow{nb\_gk\_subs})\\
\qquad\qquad\qquad \wedge\ (nb\_fp\_subs = \overleftarrow{nb\_fp\_subs} + 1))
\end{array}
$$

6.1 The execution results

In this case study, two constrains are introduced as FIFA rule as follows: the maximum number of substitution for goalkeeper is one ($gk\_subs\_max\colon \mathbb{N}_1 = 1$) and the maximum number of substitution for field players is two ($fp\_subs\_max\colon \mathbb{N}_1 = 2$). With these limitations it is possible to use the new interpreter described in this paper. Here it is possible to demonstrate that the rules were violated in a match between Italy and Norway and also how alternatively the substitutions could have been made to obey the rules.

In [15], specifications in both explicit and implicit style are provided. The explicit VDM specification is executable and the implicit one is not by the traditional VDM interpreter. The extended VDM interpreter makes it possible to execute the implicit VDM specification. We can obtain the same execution results from two different style of the specifications and interpreters.

7 Limitations with this Approach

The approach presented in this paper clearly shows that integrating a solver with the interpreter enables execution of implicit specifications, but it is also clear that not all VDM constructs can be supported due to the difference ProB and VDM. The following limitations has been observed and violable possible solution have been found fit:

Recursive function calls The solution presented in this paper shows that function calls can be in-lined. However, this will not be possible for recursive calls since it will not be possible to detect how many recursions that should be included. Mutual recursion will further complicate both such a translation but also make it more difficult to detect.

Sequence-comprehension The ProB notation does not allow sequence-comprehensions to be expressed in the available notation. However, it is at the time of writing not clear if this is a limitation it the underlying solver in ProB or in the notation.

Union type The union type is essential to VDM, and to fully support translation of token types these union types must be able to express in ProB. Unfortunately, the union type is currently not expressible in ProB.

8 Related Work

The desire for interpretation of implicit specifications is not new and have been done before using abstract interpretation for the Z language[2], and later for VDM by Fröhlich in her PhD work [5] which enabled interpretation of implicit functions and operations similar to the work presented here. However, this only supported post-conditions that was defined using a pattern with conjuncts of equality expressions and membership expressions. Another approach was taken in [13] which also aimed to improve checks and animation of implicit specification. The approach presented here is less limited because only a subset of the specification needs to be translated for each implicit definition and because the translation is interlinked with the interpreter the scope is generally much smaller since most variables will have a specific value from a type.

9 Conclusion and Future Work

In this paper we have presented the initial work enabling an interpretation of implicitly defined VDM functions and operations using the ProB constraint solver behind the scenes. It is more general than anything which has been done in the past. However, this is still work in progress since we are aware of a number of further improvements in the kinds of constructs that we are able to cover from the language. Using the constraint solver and the transformation between the different representations is naturally less efficient than interpreting a traditional explicit definition but this new works clearly gives values to users who would prefer to express models using an implicit style.

While the work presented here enables interpretation of a subset of the VDM language there is a one important type have not been possible to translate due to API limitations in ProB. The union type is one of the most used types in VDM and is essential and without it the quote type is nearly unusable. It is believed that the union type is possible to present in ProB using a free-type implementation previously made for Z [18]. This essentially allows the construction of a set of different types e.g. a: $\{\mathbb{B}, Q\}$. The character type is another type that needs investigation to determine if it can be represented in ProB. In addition type invariants are not yet incorporated in the translation. These give a special challenge in the sense that they need to be added as additional conjuncts wherever something is supposed to be of the type that has an invariant, including places where no logic expression is present. Thus, most likely the first version taking this into account will ignore the usage of such invariants in non-logic contexts.

The translation of conditions presented in Section 5.3 also needs further investigation since this rule only translated logical conditions. This is too limited since it is common to use conditions on other types. A solution is to introduce a variable (r) that will be assigned based on the truth-value of the test list:

$$(test \;\Rightarrow\; r = exp1) \wedge (\neg\, test \;\Rightarrow\; r = exp2) \tag{20}$$

The translation presented in this paper supports a number of expressions used for the case-study and example not explicitly mentioned in the paper but there are still many expressions where the translation rules have not been specified. This includes the lambda expression, which may be able to provide a better solution for function calls than the current in-lining solution where a function call from a post-condition may have the called function represented as a lambda expression containing the function body and argument constraints. Record expressions and manipulations of these are also only partially supported so far. General patterns has also not yet been covered although we expect these to be relatively easy to handle when they appear in a logical context.

In our future work we expect to attack the limitations presented in Section 7 in order to determine the final limitations that cannot be resolved with the approach presented here. Naturally there will always be implicit definitions that we will not be able to handle from an interpretation perspective but we wish to examine how to get closer to the limit of what can be interpreted.

Acknowledgments. The work reported in this paper has been carried out in the COMPASS project which is funded by the European Commission 7th Framework Program. We would like to thank Michael Leuschel and the rest of his ProB development team for their invaluable input to this work. Hiroshi Ishikawa would like to thank both Aarhus University and Niigata University of International and Information Studies for accepting his one-year-sabbatical.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)
2. Breuer, P., Bowen, J.: Towards Correct Executable Semantics for Z. In: Bowen, J., Hall, J. (eds.) Z User Workshop. pp. 185–209. Springer-Verlag (1994), cambridge
3. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc.
4. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)
5. Fröhlich, B.: Towards Executability of Implicit Definitions. Ph.D. thesis, TU Graz, Institute of Software Technology (September 1998)
6. Fuchs, N.E.: Specifications are (preferably) executable. Software Engineering Journal pp. 323–334 (September 1992)
7. Hayes, I., Jones, C.: Specifications are not (Necessarily) Executable. Software Engineering Journal pp. 330–338 (November 1989)

8. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International, Englewood Cliffs, New Jersey, second edn. (1990), iSBN 0-13-880733-7
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., 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)
11. 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)
12. Larsen, P.G., Pawłowski, W.: The Formal Semantics of ISO VDM-SL. Computer Standards and Interfaces 17(5–6), 585–602 (September 1995)
13. Lausdahl, K.: Translating VDM to Alloy. In: Johnsen, E.B., Petre, L. (eds.) Integrated Formal Methods, Lecture Notes in Computer Science, vol. 7940, pp. 46–60. Springer Berlin Heidelberg (2013), 10th International Conference, IFM 2013
14. 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), ISBN 978-3-642-24558-9
15. Ledru, Y.: Soccer referee’s book (1995)
16. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Keijiro, A., Gnesi, S., Dino, M. (eds.) FME. Lecture Notes in Computer Science, vol. 2805, pp. 855–874. Springer-Verlag (2003)
17. Leuschel, M., Butler, M.: Automatic refinement checking for B. In: Lau, K.K., Banach, R. (eds.) Proceedings ICFEM. Lecture Notes in Computer Science, vol. 3785, pp. 345–359. Springer-Verlag (May 2005)
18. Plagge, D., Leuschel, M.: Validating Z Specifications using the ProB Animator and Model Checker. In: Integrated Formal Methods (IFM 2007), LNCS 4591. pp. 480–500. Springer-Verlag (2007)
19. Plat, N., Toetenel, H.: A formal transformation from the BSI/VDM-SL concrete syntax to the core abstract syntax. Tech. Rep. 92-07, Delft University (March 1992)
20. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)

A Code Generation Platform for VDM

Peter W. V. Jørgensen, Morten Larsen, and Luis D. Couto

Department of Engineering, Aarhus University, Denmark
{pvj,mola,ldc}@eng.au.dk

Abstract. In this paper we describe ongoing work on a code generation platform that simplifies the construction of code generators for VDM in the Overture tool. The platform represents the code generated model as an Intermediate Representation (IR) and assists a code generator in transforming the IR into a structure that is easier to code generate. Since the IR is independent of any target language, a code generator can choose the transformations it needs to obtain the IR it desires. Based on the code generation platform a VDM++ to Java code generator has been developed[1], while early work is currently being made on a C++ code generator. Implementing the Java and C++ code generators has provided useful feedback for the architecture of the code generation platform. This has helped us to generalise the platform structure in order to make it a stronger foundation to use for constructing code generators.

Keywords: VDM, code generation, language paradigms, intermediate representations, tree transformations, extensibility, Java, C++

1 Introduction

When resources have been invested into modelling a system it is desirable to code generate the software implementation or parts of it from the system model to reduce the efforts needed to realise the system. Code generation therefore supports efficient transitioning to the realisation phase. However, most importantly it minimises the chances of introducing inconsistencies in the software implementation that makes it deviate from the system specification due to manual translation of the model into code.

With the existence of many popular target languages it is common for code generators to provide support for multiple target languages in order to target a larger group of users. This can, however, easily lead to duplication of efforts when implementing code generators — especially if the target languages follow the same paradigms such that the rules used to code generate a source language are the same.

Ideally it should be possible to reuse the transformations used to code generate constructs of a source language. As an example, consider the VDM set comprehension {`x|x` **`in set`** `S & pred(x)`}, which constructs a new set from the elements of `S` for which `pred(x)` is true. In imperative languages such as Java and C++ this language construct is non-trivial to code generate since Java and C++ do not have similar constructs included. The same functionality can be obtained in those languages, but it

[1] The Java code generator is available in Overture releases 2.1.0 onwards

requires use of multiple language constructs for iterating over a set, evaluating a predicate on each set member, adding elements to a resulting set and so on.

The potential for different *backends* (a code generator that extends the code generation platform) to use the same transformations is particularly good when the target languages belong to the same paradigm (e.g. they are object-oriented or functional in style). In that case they will have many language constructs in common and thus face many of the same challenges with respect to code generation. When the same transformation can be used by different backends to code generate a source language construct it is beneficial to apply the transformation to the code generator input before it reaches the backend in order to obtain a transformed structure that is easier for a backend to code generate. The idea is therefore to structure a code generator such that it is possible to pick the transformations that will lead to a structure that requires the least effort for a backend to code generate. In this paper we explore this approach to code generation in order to reduce the efforts needed to implement code generation for multiple backends.

The paper is structured as follows. Section 2 describes how IRs support the implementation of code generators or backends. Section 3 explains how tree transformations simplifies the implementation of backends. Section 4 provides an overview of the code generation platform used for implementing the Java and C++ backends. Section 5 and Section 6 describes challenges encountered for the implementation of the Java and C++ backends, respectively. Section 7 describes future plans for the code generation platform. Section 8 describes related work and finally section 9 concludes our work.

2 Intermediate Representations

One approach adopted by compiler developers is to transform the Abstract Syntax Tree (AST) specified in the source language into an IR that preserves the semantics of the input and from which the backend generates code in the target language. The IR helps managing the complexity of the compilation process by being independent of details specific to the source language and the target language. An IR obtained from the VDM AST serves a similar purpose by mitigating the complexity of generating code from a VDM model. This would, for example, enable the code generator to unify VDM functions and operations into the concept of a method as seen in a programming language such as Java. Then the backend only needs to treat a single (language) construct without having to distinguish between functions and operations.

Code generating a VDM construct is easier if equivalent or similar constructs appear in the target language. For example, a set comprehension in VDM is more likely to have an equivalent construct in a functional programming language, which would simplify the task of code generating it. However, for a target language that does not support set comprehensions, code generating this construct is non-trivial. This is not surprising since Java is an imperative language and a set comprehension is a functional concept. Similarly, code generating the object-oriented concepts of VDM to a functional language will require constructs to be code generated that are not naturally expressed in terms of the target language. In general it is difficult to code generate across paradigms since a construct with a strong relation to one paradigm will not be present in other paradigms thus requiring a strategy to translate that construct, which potentially needs to make widespread changes to the IR.

In this work we address the challenges of code generating constructs where no obvious mapping exist. We do this by translating the VDM AST into an IR to which a series of transformations are applied. By transforming language constructs that are difficult to code generate into new (possibly larger) tree structures, based on concepts that are easier to code generate, the implementation of a backend can be simplified. If the backend provides support for code generating the replacement constructs used by the transformations then it follows that the backend already supports code generation for the complex construct. In that case the complexity of the code generation process is comprehended entirely using tree transformations. The advantage of this approach is that the transformations can be made such that they are independent of the target language. This enables other backends to benefit from the same transformations when code generation is implemented for other languages.

3 Tree transformations

In order to show the usefulness of applying transformations to the IR, before handing it to the backend, we will consider a set comprehension as an example of a construct to code generate. Since the set comprehension is a functional concept many target languages, such as those that are imperative in style, need to use several different constructs to obtain the equivalent functionality.

3.1 Code generating the set comprehension

The VDM snippet in Listing 1 shows an example where a set comprehension is used to construct a new set obtained by iterating over the set `S` and selecting the elements for which `pred(x)` is true. Therefore collection comprehensions provide a convenient notation to construct collections from other collections which would otherwise require use of several different constructs in an imperative language such as Java.

```
public f : () -> set of nat
f () ==
let a = {x | x in set S & pred(x)}
in g(a,a);
```

Listing 1. Example of a set comprehension in VDM

Without using the set comprehension the equivalent functionality can be obtained by rewriting the function in Listing 1 into a VDM operations that explicitly specifies the semantics of the set comprehension using an imperative style of writing as shown in Listing 2. Due to the expressiveness of the set comprehension Listing 1 obtains the same functionality as that of Listing 2 using fewer lines of VDM. Although the function in Listing 1 and the operation in Listing 2 are semantically equivalent, the listings represent different challenges for a code generator. The reason for this is that the two VDM snippets use different constructs to obtain the same result.

```
public op : () ==> set of nat
op () == (
dcl setCompResult : set of nat := {};
for all x in set S do
  if pred(x) then
    setCompResult := setCompResult union {x};
(dcl a : set of nat := setCompResult;
 return g(a,a)));
```

Listing 2. Imperative specification of the VDM set comprehension in Listing 1

Therefore, the difficulty of code generating a VDM model depends on the style of modelling and the target language. VDM is a multi-paradigm modelling language, using constructs of both the object-oriented and the functional paradigm, and therefore backends will experience situations where a construct does not have a one-to-one mapping into the target language.

3.2 Transforming language constructs

One approach to simplifying code generation of a VDM model is to have the modeller refine the model such that it uses constructs that are easier to code generate. However, eliminating constructs that are problematic to a code generator using model rewriting, limits the modeller to use only a subset of the source language. This also clutters the model with details used to assist the backend in generating code from the model, thus going against the point to have a model that abstracts away details that do not contribute to obtaining the insight needed.

A better approach is to have this kind of model refinement done at a later stage to make it transparent to the modeller and avoid restricting modelling to only a subset of the source language. This could be done by applying transformations to the IR such that constructs that are problematic to code generate get replaced with other IR constructs in order to obtain a *simplified IR* that is easier to code generate.

The use of transformations is part of a larger platform architecture that is used to construct backends. In section 4 the architecture of this code generation platform is detailed to make it clear how it facilitates the construction of code generators.

4 Architecture of the code generation platform

The code generation platform, shown in Figure 1, takes a VDM++ model as input and use it to construct an IR that represents the generated code. After the IR has undergone a transformation process it is input to a backend that translates it into source code in a target language. To further detail the approach taken to construct code generators this section describes the architecture of the code generation platform and how it interacts with the backend of a target language.

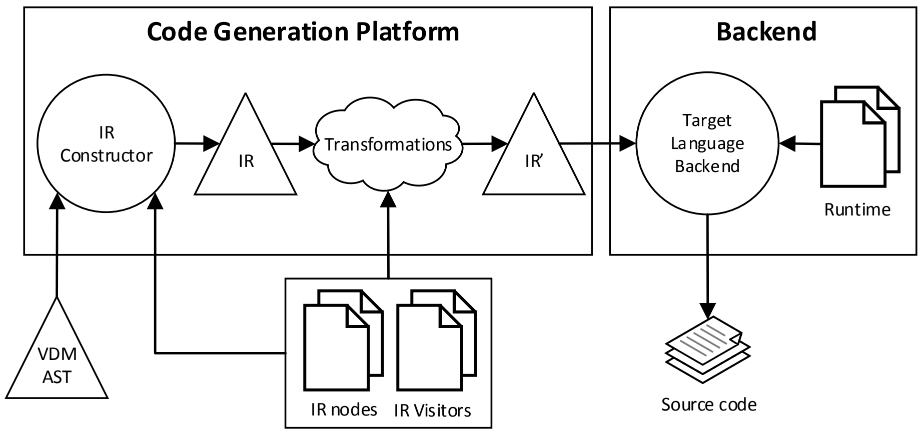

Fig. 1. An overview of the code generator platform architecture

4.1 The intermediate representation life cycle

The IR as first constructed from the VDM AST represents a slightly simplified and extended version of the VDM model. For example, records in the IR are allowed to have methods (unlike records in VDM). The purpose of this will become more clear in subsection 5.1 where it is discussed how VDM records are code generated to Java.

The IR simplifies the tree structure by eliminating or rewriting use of certain operators by replacing them with use of other operators. For example, writing a logical implication on the form $A \Rightarrow B$ where A and B are propositions, is convenient in a mathematical language such as VDM, but since it is a derived and, not elementary operation of boolean logic, this operator is rarely seen in a programming language. Therefore the expression $A \Rightarrow B$ is represented as $\neg A \vee B$ in the IR.

Next, code generation enters the transformation process where constructs that are difficult to code generate (or even unsupported by the backend) are translated into new tree structures that can be code generated. The IR before and after it has undergone the transformation process is denoted `IR` and `IR'` in Figure 1, respectively. Finally, the simplified IR is input to the backend that translates it into a target language.

4.2 The design of the intermediate representation

The IR nodes are generated using the ASTCreator tool [1], which is a SableCC [9] inspired tool. As shown in Figure 2 the ASTCreator takes a description of the AST as input and outputs nodes from which concrete ASTs can be constructed. The generated AST structure uses bidirectional node relations which makes it easier to search the tree both upwards (e.g. finding the enclosing class of a node) and downwards (e.g. looking up type information of child nodes). The nodes also have functionality for making changes to the tree structure, which is needed when nodes must be replaced with new tree structures during the transformation process.

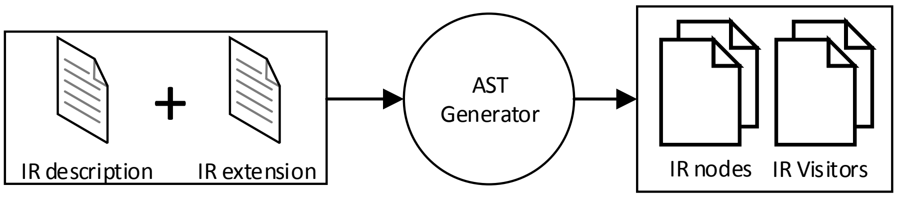

Fig. 2. The ASTCreator produces the IR nodes and visitors based on the IR description

The ASTCreator also produces functionality to traverse the AST. Tree walkers or visitors are implemented using the visitor pattern and play an important role in the current AST architecture used in the Overture tool where they, for example, are used to implement the type checker and the interpreter [6]. Similarly, the `IR Constructor` is a visitor that traverses the VDM AST and constructs the `IR` from it.

The ASTCreator is designed such that it allows optional AST extensions to complement an existing AST description and therefore it is possible to add new nodes to the IR. This design benefits the extensibility of the code generation platform since extensions to the IR can be made to support the implementation of additional transformations.

4.3 The backend

The final step of the code generation process translates IR constructs into source code of the target language. When transformations have simplified the IR then ideally these mappings should be trivial. The process of mapping IR constructs into source code of the target language for the Java backend is done using the template based technology, Apache Velocity [11]. Optionally, the generated code can make use of a runtime. As an example, the Java backend includes a runtime to represent VDM types and implementation for some of the VDM operators such as the sequence modification.

5 Code generating Java from VDM++

Java has fewer language constructs than other object-oriented languages such as C# and C++, which makes it simpler, but also less expressive as a language. Such languages are difficult to code generate since fewer constructs in a target language implies less ways to code generate a source language. This has led to some instructive experiences when implementing the Java backend, some of which we will discuss in this section.

5.1 Code generating value semantics

In VDM records, tuples and collections have copy-by-value semantics (referred to as “value semantics” throughout the remainder of the paper), which is the behaviour where a variable is copied when it is passed as a parameter or appear on the right-hand-side of an assignment. This is also the semantics used for structs in programming languages such as C++ and C#. In Java where there is no direct support for structs (or something similar) the equivalent can be obtained by representing a value type using a class and then have the instances explicitly copied as needed – normally by invoking a method on the instance that does the copying. Therefore, code generating value types to Java require careful attention. To demonstrate this, consider the VDM snippet in Listing 3, where the record type `Vector2D` is used to represent two-dimensional vectors. In this listing two vectors `v1` and `v2` are created with `v2` being a copy of `v1`. Since records have value semantics subsequent modifications to `v1` have no effect on `v2` and therefore the operation would return 1.

```
public op : () ==> nat
op () == (
dcl v1 : Vector2D := mk_Vector2D(1,2);
dcl v2 : Vector2D := v1; -- Copy using value semantics
v1.x := 2;
return v2.x;)
```

Listing 3. Value semantics in VDM demonstrated using records

Had the vectors in Listing 3 been modelled using a class rather than a struct then `v2` and `v1` would have been object references pointing to the same object. Therefore, subsequent modifications to the underlying object using any of the two object references would effect the same object and in that case the operation in Listing 3 would return 2.

5.2 Obtaining the effect of value semantics in Java

To obtain the effects of value semantics using a Java class one can provide a `clone` method and invoke it on the instances when they need to be copied. Therefore, code generating the VDM operation in Listing 3 yields the Java code shown in Listing 4. Note, how the backend invokes the generated `clone` method in order to ensure that the copy of `v1` respects the rules of value semantics. Since the responsibility of the `clone` method is to copy the fields of the associated class it must be generated specifically for each record in the IR.

```
public Number op() {
  Vector2D v1 = new Vector2D(1L, 2L);
  Vector2D v2 = v1.clone();
  v1.x = 2L;
  return v2.x;}
```

Listing 4. Java code generated from Listing 3 demonstrating how the Java backend obtains the effects of value semantics

A record in the IR can have methods (unlike records in VDM, which only have fields) and therefore the clone method can be added as a child to the record node. Similarly, the Java backend adds a method for record comparison based on structural equivalence (field-wise comparison), a method for calculating the hash code of a record (such that it is suited for use in collections) and a method that computes the string representation of the record. Since these methods are added as extensions to records in the IR they are specified solely using IR nodes.

5.3 Code generating functional concepts in Java

The approach of applying transformations to the IR has enabled the Java backend to code generate complex constructs without being aware of their presence in the VDM model. The reason for this is that these constructs are transformed into tree structures composed of IR nodes that are simpler to code generate. In that sense code generating the functional concepts required no extra effort for the implementation of the Java backend since code generation for the simpler constructs was already supported. The result of code generating the set comprehension in Listing 1 is shown in Listing 5.

```java
public static VDMSet f() {
  VDMSet setCompResult_1 = SetUtil.set();
  VDMSet set_1 = S.clone();
  for (Iterator iterator_1 = set_1.iterator();
    iterator_1.hasNext();) {
    Number x = ((Number) iterator_1.next());
    if (pred(x)) {
      setCompResult_1 = SetUtil.union(
        setCompResult_1, SetUtil.set(x));
    }
  }
  VDMSet a = setCompResult_1;
  return g(a, a);}
```

Listing 5. Java code generated from the VDM function in Listing 1

5.4 Iterating over collections

Iterating over collections in a target language is often done using library classes specific to that language. The Java backend does this using the `java.util.Iterator` class, as shown in Listing 5, whereas the C++ backend uses the C++ standard library iterators (`std::iterator`). Since transformations may produce new tree structures that iterate over collections the code generation platform enables transformations to be configured with language specific ways to iterate over collections. Iteration strategies, as they are termed, have been added to the code generation platform as a result of the feedback from implementing the C++ backend, and the Java backend has been updated accordingly such that it also uses the iterator strategies.

Language iterators must implement a language iterator interface that requires implementation of methods to

1. Initialize the iterator (or counter) used to perform the iteration
   - e.g. `Iterator iterator_1 = set_1.iterator();`

2. Build the expression used to determine whether there are more elements to process
   – e.g. `iterator_1.hasNext();`
3. Increment the iterator (or counter) and read the next element
   – e.g. `Number x = ((Number) iterator_1.next());`

The language iteration strategies allows incrementation of the iterator and reading the next element (the third item) to be done in separate steps (increment the iterator and then read the next element), but in Java and C++ it is common to do this in a single step – at least when using the built-in iterator classes. Each method constructs a tree to express the generated code related to that method using IR nodes. Since the trees generated by these methods may want to represent types that are external to the code generation platform the IR offers nodes to represent external constructs of the target language. For example, in order to allow easier integration with a target language the IR offers a construct to represent external types (e.g. the `Iterator` class in Java).

6 Code generating C++ from VDM++

The point of using a code generation platform is that backends of similar target languages can use the same functionality in order to reduce the efforts needed to implement code generation. The implementation of a C++ backend has provided useful feedback for the architecture of the code generation platform and given rise to some future plans that will be covered in section 7. In this section we describe some of the interesting challenges encountered for the implementation of the C++ backend and relates it to the work on the Java backend described in section 5.

6.1 Code generating reference semantics

In VDM classes use reference semantics, and therefore two object references are considered equal if they point to the same object. The same applies for object references in Java, but in C++ the objects must be referred to using pointers in order to obtain reference semantics. A C++ object allocated on the stack, on the other hand, uses structural equivalence (field-wise comparison) to determine equality.

When an object must be shared among multiple methods it is common to put it on the heap and access the object via a pointer or reference. In C++ memory that is allocated on the heap must also be deallocated explicitly by the programmer since C++ does not support garbage collection. In order to address this issue, the C++ backend implements an object reference using a shared pointer from the standard library (i.e. `std::shared_ptr`), which provides reference counting and automatic deletion when no more references for the underlying object exist. In Java, garbage collection is a language feature, and therefore the Java backend does not take memory deallocation into account.

6.2 Code generating functional concepts in C++

The C++ backend uses the same transformations as the Java backend to transform functional VDM language constructs (collection comprehensions, quantified expressions etc.) before they are code generated. Configuring the visitor that performs the transformations of the functional concepts with an iterator strategy and applying it to the set comprehension in Listing 1 yields the generated C++ code shown in Listing 6. Since the transformation takes care of expressing the algorithmic part of evaluating a set comprehension, and this can be reused directly by the C++ backend, the efforts needed for code generating the set comprehension in Listing 1, is only a matter of providing the iterator strategy – given that the simpler constructs can already be code generated by the backend (e.g. the if-statement).

```
vdm::set<int> f() {
  vdm::set<int> setCompResult_1 = vdm::set<int>::
      from_list();
  vdm::set<int> set_1 = S;
  for (vdm::set<int>::iter iterator_1 = set_1->begin();
      iterator_1 != set_1->end();  ){
    int x = *iterator_1++;
    if( pred(x) ){
      setCompResult_1 = vdm::set<int>::set_union(
          setCompResult_1, vdm::set<int>::from_list( x));
    }
  }
  vdm::set<int> a = setCompResult_1;
  return g(a,a);};
```

Listing 6. C++ code generated from the VDM set comprehension in Listing 1

6.3 Representing VDM records in C++

Record values in VDM are copied when assigned from, passed as argument (to a function or an operation), or returned as a value. This is also the behaviour for a class in C++, and therefore this construct is used to represent a VDM record. However, for a C++ class to create values from another class, the declaration of that class must be visible to the compiler (such that the size of the value can be computed), otherwise the class can only be pointed to (using a fixed size pointer). For example, consider a class `R1` that has a field of the class type `R2`. The declaration of `R2` must appear before the declaration of `R1` otherwise the C++ compiler raises an error. A partial solution to this, is to sort the dependencies using a topological sort. This does, however, require a graph with no directed cycles. Another solution is to treat records as classes and use the `std::shared_ptr` type and generate additional code to obtain the effects of value semantics as it was done by the Java backend described in subsection 5.2. Currently the C++ backend uses the first approach where topologically sorted C++ classes are used to represent VDM records.

6.4 Representing VDM collections in C++

The C++ standard library include lists, sets and maps but it lacks some of the functionality needed to fully represent the VDM collections. Therefore, the C++ backend uses a runtime that includes classes to represent the VDM collections (e.g. `vdm::set`) and operations on them. For example, the set union operator is implemented as a method, `set_union`, and used, for example, in the transformation of a set comprehension as shown in Listing 6. To code generate the includes needed in the generated code to access the runtime, the C++ backend uses the external type construct of the IR.

To ensure that the value semantics for VDM collections are preserved, the runtime collections overload the assignment operator and the copy constructor such that a collection gets copied correctly when assigned from, passed to a method or returned as a value. The advantage of this approach is that the C++ compiler becomes responsible for generating the code that copies the collection object the places where it is needed. This is different from the approach used by the Java backend, which needs to analyse the IR in order to find out where the `clone` method needs to be invoked. However, since Java classes use different semantics than C++ classes and Java does not allow operator overloading nor does it use copy constructors, the approach used by the C++ backend cannot be used.

7 Future plans

Looking forward, there are several immediate improvements that can be made to the code generation platform, in terms of expanding the coverage of VDM and adding support for additional target languages. There are also possibilities of looking into how the extensibility and the reuse of transformations can be improved.

7.1 Adding support for new target languages

We may wish to generate code for different languages of different paradigms to further validate the code generation platform architecture by implementing backends based on target languages that are different from Java and C++. Since Java and C++ are both imperative languages that use object-oriented principles, the work presented in this paper focuses on reusing the existing transformations. Adding support for a new language of a different paradigm would provide feedback for the code generation platform, which would lead to further improvements in terms of its extensibility.

7.2 Atomic transformations

One way to add support for new target languages is to continue expanding the code generation platform, by adding transformations and altering the existing ones to facilitate the support for new target languages. However, there are issues with this approach: the constant maintaining of existing functionality to support new target languages indicates a poor platform extensibility. Instead it should be possible to extend the existing functionality that the code generation platforms offers without affecting it.

In order to address this issue, the code generation platform must be re-designed with respect to the way transformations are applied. At the moment each transformation is large and extensive. For example, when transforming the functional elements in preparation for Java code generation, all elements are removed in the same visitor. This occurs at code level where all these transformations are implemented in one visitor.

To understand the consequences of this, consider the case where we wish to code generate for a language `JavaEQ` that is like Java in every regards, except that it contains support for existential quantifiers. In the current architecture this requires either subclassing and overriding the methods in the Java visitor (which immediately locks the new transformations in the Java hierarchy) or duplicating and changing code.

Therefore, we propose use of fine-grained *atomic transformations* that allow constructs in the IR to be transformed one at a time. These transformations would then be grouped in libraries in terms of which types of constructs they replace rather than which programming language they target. From here, we can define composite transformations that support specific programming language as combinations of the atomic transformations. For example, if we consider a transformation to be a relation from IR to IR then we would say that $JavaTrans = ExistsTrans \circ SetCompTrans \cdots$.

7.3 Extensibility of the code generation platform

Use of atomic transformations would also make it easier to maintain existing backends. For example, Java supports lambda expressions as of the recent Java 8 release[5]. In terms of atomic transformation, updating the Java backend to support Java 8 is as simple as removing the lambda expression transformation from the sequence of transformations. This would be significantly simpler (and shorter) than editing the visitor code to remove the transformation. In addition, if this visitor is being used to code generate for another language without lambda expressions, then the code must be split.

In order for this approach to be viable, the atomic transformations would have to respect various properties. Namely any two atomic transformations should be compatible with each other. This means that they alter independent parts of the tree while preserving anything else. An initial approach to this might be to ensure that no two transformation operate on the same node.

As certain transformations push the tree in a particular direction, other transformations no longer become available. This is acceptable since one is not interested in combining transformations arbitrarily but rather do so always with the goal of getting closer to a particular target language. There may also be issues with the order of the transformations. Again, it is not essential that all transformation can be combined in arbitrary ways. Only that there is one way to combine all the desired transformations.

There are some implementation challenges to the atomic transformation approach. Particularly since multiple inheritance is not available in Java (the language in which the transformations are implemented) and therefore the combination of multiple transformations in a single visitor is non-trivial. Finally, there may also be performance considerations when performing multiple small transformations versus a single larger one.

7.4 Code generating trace definitions

Overture supports automated test generation and execution of large collections of test cases that are derived from a trace definition [7]. The trace definition uses a short-hand notation and can be thought of as a regular expression that expands to test cases or sequences of operation calls that match the pattern of the trace definition. When the expansion process is complete the VDM interpreter executes the tests one by one and optimises the process by filtering out tests based on the outcome of other tests (e.g. a test will fail if it starts with a sequence of operation calls that it known to cause a test to fail). In addition, Overture offers different techniques to make reduced sets of tests as representative as possible – a technique known as shape reduction.

This future plan item aims to produce and execute code generated from a trace in order to make test execution more efficient. This can be done by expanding the trace into tests that are code generated and executed, instead of having them executed in the VDM interpreter. However, for this to be of any value it should make up for the time spent producing and executing the code generated tests. This approach would also allow use of existing techniques available for test filtering and shape reduction. Another approach is to code generate the trace directly such that when the generated code is executed it will expand and execute all the tests matching the pattern of the trace. However, this approach needs new ways to do test filtering and shape reduction, since it must be done during execution of the code generated trace.

8 Related work

This section describes existing work on code generation for VDM and approaches to constructing code generators for cases where expressing the source language in terms of the target language is non-trivial.

8.1 VDM code generation

VDM code generation was developed for VDMTools [10] in the nineties with support for both Java and C++, and has primarily been used to code generate prototype implementations rather than final production code. In the late nineties the Java code generator was extended to support code generation for the concurrency mechanisms of VDM++ [8]. Unfortunately, there is no scientific literature available to document the approach used to construct the VDMTools code generation feature.

VDMTools supports code generation for a larger subset of VDM compared to the current code generation platform described in this paper. However, VDMTools does, for example, not support code generation for a functional concept such as a lambda expression, which is non-trivial to express in earlier versions of Java where this is not supported. Code generation of lambda expressions (for earlier versions of Java) is achieved by the Java backend described in this paper by applying transformations to the IR.

8.2 The DMS Software Reengineering Toolkit

The DMS Software Reengineering Toolkit is a commercial set of tools for program analysis and transformations [3]. It contains tools for lexer and parser generation, functionality to pretty print an AST and specify program transformations, termed "transforms", using source rewrite rules. Rules are written in the DMS's Rule Specification Language and typically have the form $LHS \rightarrow RHS$ **if** *condition*. A rule is interpreted such that when a part of the program matches *LHS* it gets replaced by *RHS* if the condition is true. For example, a rule can be specified such that it replaces an assignment statement on the form `v = v+1` with the incrementation `v++`. To support the specification of rules, patterns use language syntax categories (e.g. both *LHS* and *RHS* must be of the statement syntax category) and it also allows use of metavariables to match with variables and expressions in the source language.

Transforms and source rewrite rules work at the concrete syntax level and therefore this approach differs from that used by the code generation platform described in section 4. Here transformations are applied to the IR at the abstract syntax level using visitors generated using the ASTCreator tool, where a case must be implemented to match a constructs in the IR. For example, a case can be implemented to match a set comprehension but there are also cases that allows categories of IR nodes such as statements, expressions and numeric binary expressions to be matched. Visitors have the potential to make changes all over the IR including adding new types and definitions as done by the Java backend during code generation of records as explained in subsection 5.2.

8.3 Code generating a logic language

Research has been done on translating logic languages such as Prolog [4] into imperative languages. In logic languages programs are expressed as logical formulas or horn clauses. Queries can be made about a program from which the interpreter will try to construct a proof. An example of a Prolog to Java translator is found in Prolog Café [2].

In order to be able to execute a code generated Prolog program, there must exist a runtime to support the generated code, and take over the role of the Prolog interpreter. This runtime is therefore responsible for trying to construct a proof that meets a given query. Since the Prolog interpreter is based on a highly efficient implementation and makes use of sophisticated algorithms to implement the traversal of the search tree, the implementation of such a runtime (or at least one that is efficient) is a complicated task.

The approach to use transformations to mitigate the complexity of code generating a logic language to a language such as Java can be expected to be of limited value compared to the case where code generation is more naturally done by expressing a source language using an IR. The reasons for this is that most of the challenges of code generating a logic language such as Prolog to Java involves the implementation of the runtime that constructs the proof.

9 Conclusion

The code generation platform presented in this paper supports construction of different backends and reduces the efforts needed to code generate a source language to multiple target languages. It achieves this by representing the generated code as an IR and subjects it to a series of semantic preserving transformations in order to obtain a tree structure that is easier for a backend to code generate. Since the IR is independent of any source and target language backends facing similar challenges during the code generation process, can use the same transformations to simplify their implementation.

To validate the architecture of the code generation platform a VDM++ to Java backend has been developed for the Overture tool, and work is currently being made on a VDM++ to C++ backend. Java and C++ are imperative languages and therefore both backends can use the same transformations in order to obtain an IR that is easier to code generate. To demonstrate this, it was shown how a set comprehension was transformed into a larger tree structure based on IR constructs of an imperative nature.

Applying transformations to an IR has proven useful for implementing the Java and C++ backends, but the the approach also supports code generation for other source languages. Therefore we hope that the work presented in this paper will be useful for others working with code generation.

Acknowledgements

This work is partially supported by the EU Framework 7 Integrated Project "Comprehensive Modelling for Advanced Systems of Systems" (COMPASS, Grant Agreement 287829). For more information see `http://www.compass-research.eu`.

References

1. The ASTCreator website (2014), `https://github.com/overturetool/astcreator`
2. Banbara, M., Tamura, N., Inoue, K.: Prolog Cafe : A Prolog to Java Translator System. Lecture Notes in Computer Science, vol. 4369, pp. 1–11. Springer (2005)
3. Baxter, I.D., Pidgeon, C., Mehlich, M.: DMS: Program Transformations for Practical Scalable Software Evolution. In: Proceedings of the 26th International Conference on Software Engineering. pp. 625–634. ICSE '04, IEEE Computer Society, Washington, DC, USA (2004)
4. Blackburn, P., Bos, J., Striegnitz, K.: Learn Prolog Now!, Texts in Computing, vol. 7. College Publications (2006)
5. Gosling, J., Joy, B., Steele, G., Brach, G., Buckley, A.: The Java Language Specification Java SE 8 Edition. Oracle America, Inc. (March 2014)
6. Jørgensen, P.W., Lausdahl, K., Larsen, P.G.: An Architectural Evolution of the Overture Tool. In: The Overture 2013 workshop (August 2013)
7. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 278–285. SEFM '10, IEEE Computer Society, Washington, DC, USA (September 2010), `http://dx.doi.org/10.1109/SEFM.2010.32`, ISBN 978-0-7695-4153-2
8. Oppitz, O.: Concurrency Extensions for the VDM++ to Java Code Generator of the IFAD VDM++ Toolbox. Master's thesis, TU Graz, Austria (April 1999)
9. The SableCC website (2014), `http://www.sablecc.org/`
10. SCSK: VDMTools website. *http://www.vdmtools.jp/en/* (2007)
11. The Apache Velocity website (2014), `http://velocity.apache.org/`

Introducing the Overture Architecture Guide

Luís Diogo Couto

Aarhus University
Department of Engineering
`ldc@eng.au.dk`

Abstract. Overture is an open source IDE for VDM with an extensible, plug-in based architecture. However, it currently faces significant challenges to its developer resources. The number of active developers is small and it is difficult to attract new developers, in part because the Overture code base is large and complex, and therefore challenging to learn. This is further complicated by a lack of documentation throughout. This paper presents an initial effort to address these challenges in the form of a guide to the architecture of Overture. The guide itself is a living document, being maintained in the developer wiki. This paper presents key sections of the guide as well as issues encountered during its production and possible solutions.

1 Introduction

Overture is an open source IDE for VDM where extensibility is a main focus and goal [3]. This allows interested Overture users to contribute to the tool both by maintaining and improving the existing code base but also by adding brand new functionality. In order to attract these contributions it is essential that new developers are able to orient themselves in the code base.

However, the Overture code base is rather large (nearly 400k lines of code), spread across multiple components and implementing many different functionalities. It is challenging for new developers (and even experienced ones) to orient themselves in the code base. This issue is further compounded by a general lack of documentation across the project, both in terms of code comments and annotations and external documents such as design specifications or API descriptions.

An initial effort into improving the technical documentation of Overture has been the production of an Overture architecture guide. The guide itself is meant to be a living document, evolving along with the tool. As such, it is maintained in the developer wiki and is available at `https://github.com/overturetool/overture/wiki/Architecture-Guide`.

The guide is primarily aimed at new developers who need to familiarize themselves with Overture. But it should be of use to anyone that needs information regarding the architecture of Overture. It is meant to be an authoritative source on the architecture and should be kept up-to-date as the architecture changes. The focus is on the interactions between the various components of Overture so their descriptions are written in terms of what they require and provide. Less attention is paid to the internal architecture of the components.

This paper introduces the current, work-in-progress version of the Overture architecture guide. It aims to attract the attention of the broader Overture community in order to both assess the quality of the guide and discuss ways to improve the architecture of Overture and address the developer resource challenges.

This paper presents significant excerpts of the architecture guide, with minimal alterations to reflect the change of format. Also, throughout the paper, including the guide excerpts, suggestions are made to address various architectural issues uncovered during the production of the guide. These suggestions are highlighted in boxes as follows:

Suggestion 0: Improve the architecture of Overture.

The remainder of this paper is structured as follows:

– section 2 presents excerpts of the guide that give a brief high-level overview of Overture and describes the two perspectives that can be taken on the architecture.
– section 3 presents excerpts of the guide describing the architecture of the core components of Overture.
– section 4 discusses various architectural issues uncovered while producing the guide.
– section 5 concludes the paper.

2 Overview

This section presents an overview of the Overture architecture. Two different perspectives can be taken on the architecture and the components (also called modules) of Overture:

Source Code Perspective where each architecture module corresponds to a Maven [1] module. Maven is the build system used for Overture and it provides a series of conventions for organising source code and related files that Overture follows.

Functionality Perspective is a more conceptual perspective that blurs the lines between modules and simply considers distinct, high-level functionalities.

The following excerpts show the guide's descriptions of both perspectives. It is worth noting however, that most of the architecture guide follows the source code perpective.

2.1 Source Code Perspective

The Overture code base is grouped in two main sets of (Maven) modules:

- Core Modules implement and provide the various functionalities of the tool
- IDE Modules are responsible for the User Interface and are heavily based on Eclipse[1] plug-ins.

In theory this should enforce strong separation of concerns — the Core modules implement functionality and the IDE Modules implement the user interface. However, there are a few quirks worth mentioning. The IDE modules are all based on Eclipse and built with Tycho[2] so the set of IDE modules does not implement all user interfaces. It only implements the IDE version. On the other hand, some of the IDE Modules implement functionalities that are not available in the Core set (examples include `uml2` and `latex`). More details can be found in the module overviews.

There are some advantages to separating the code into functionality modules and IDE modules. Namely the fact that most of the actual functionalities of Overture are detached from Eclipse. Therefore, should we ever decide to change the IDE support, migration will be easier.

In general, the development of new functionalities for Overture should follow this pattern: implement the actual functionality in a core module and implement its user interface in the form of an Eclipse Plug-in that provides access to the core module. Any external dependencies of a plug-in should be manged through Maven. If that is not possible due to, for example, Eclipse dependencies that are not built with Maven then the module *cannot* go in the core set.

2.2 Functionality Perspective

In terms of functionality, Overture can be split into core functionality and plug-ins (keep in mind that this is not the same as the core and IDE module sets). The core functionality is implemented by the parser, type checker and AST. The parser and type checker construct and validate an AST that represents the VDM model being worked on. These 3 modules are considered the core because they do not make sense on their own. Every functionality in Overture requires a valid AST to work and these 3 modules are responsible for providing it. In essence, the core provides a way for the plug-ins to interact with a VDM model.

Every other functionality in Overture is conceptually viewed as a plug-in or extension on top of the core. This generally means that almost all functionality depend on the AST. Most of them need not depend on the parser directly. You need the parser present to construct an AST but in general you should not need to declare the parser as a dependency. The type checker is a bit more subtle. In addition to validating the AST, it offers various utilities so it may be necessary.

[1] `http://eclipse.org/`

[2] For more information about Tycho see `http://eclipse.org/tycho/`

3 Core Modules

This section presents the architecture guide's descriptions of the core modules. The guide provides an overview of the architecture of the set of core modules as well as brief individual descriptions of each module. We begin by presenting the overview of the architecture.

3.1 Core Architecture Overview:

The core modules are written in pure Java and all their dependencies are managed exclusively through Maven. This makes building and working on them a fairly straightforward process.
Fundamentally, all modules are built around the Abstract Syntax Tree (`AST`). The `AST` is an in-memory representation of the VDM model being worked on. The `Parser` is responsible for reading a VDM source and constructing its tree. The `Type Checker` is responsible for validating a tree. Practically all use cases for Overture involve a type checked AST. However, most modules should not need to interact with the `Parser` nor the `Type Checker` (but see further below) and instead should have a type checked AST provided to them. The task of constructing this AST should fall to an overall core module and only comes into play when interacting with the user. For example, the `interpreter` should only concern itself with executing an AST. How that AST came to be is of no importance to the interpreter.
Figure 1 presents an idealized view of the modules and their interdependencies. It does not match the Maven modules on a 1 to 1 basis since it groups a few of them (for example `Combinatorial Testing` represents 2 Maven modules). It should give a good idea of the overall architecture of the Overture core.

There are some points worth noting with regards to the overall architecture, as shown in Figure 1:

- Most modules should interact only with the `AST` (either to build or analyze it) and with the `Test Framework`
- The `Test Framework` consists of some utility methods to use VDM sources as inputs and XML files as results. Most modules use it for testing, as expected.
- `Command Line` and `GUI Builder` are isolated because they are not modules in the same sense as the rest. The `Command Line` is just a bash script that calls the main Overture jar and the `GUI Builder` is an externally developed VDM tool that we bundle with Overture
- A few functionality modules need to interact with each other:
  - The `POG` uses the `Pretty Printer` to format its output

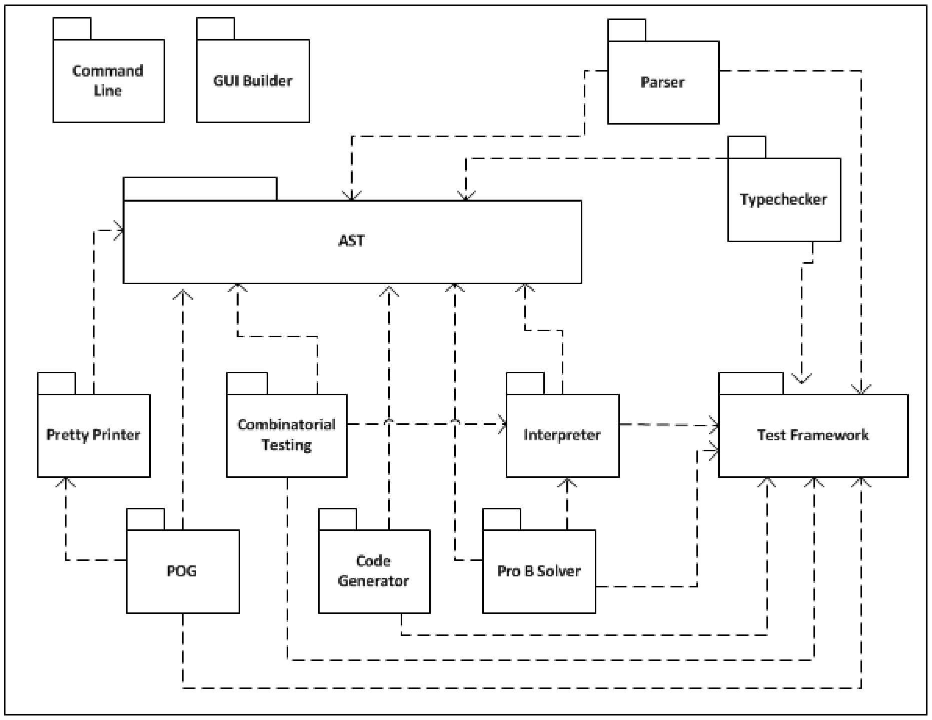

Fig. 1: Architecture of Core Modules

- `Combinatorial Testing` uses the `Interpreter` to execute the generated tests.

While the current architecture is a significant step up from previous efforts, there are still some problems. The biggest issue is a lack of an overall module to coordinate the work between the various functionalities. This is the reason why several module dependencies exist. This issue occurs primarily when exposing functionality to the user. It is significantly less relevant on the IDE side where this kind of coordination code exists. But on the core side the lack of a coordinator introduces several additional dependencies that are not really needed.

3.2 Module Descriptions

For each module, the guide provides a very brief summary of its functionality and purpose along with a description (and rationale) of its most important dependency connections to other modules. Each module description also contains a table listing all of its outgoing and incoming dependencies to other modules. The main purpose of these descriptions is to explain to the reader where each particular module fits in the overall Overture architecture. We present the descriptions of a few core select modules below.

Command Line

Consists of 2 very simple scripts (bash and bat) to run the command line version of Overture. These scripts essentially wrap the call to the main Overture jar. They are not actually connected to any of the modules so their influence on the architecture is none. Still, they are mentioned for completeness' sake.

AST

This is the central module of Overture. The `AST` module provides a series of classes that implement an AST for VDM. It consists mainly of node classes (the `INode` hierarchy) that represent the nodes of the tree and a series of abstract visitors that can be subclassed in order to implement new functionalities. Both the nodes and visitors are generated automatically with the ASTCreator tool from a specification file.
In addition to the auto-generated classes, there are several utility and support classes. The most notable are:

- `AstFactory`, as the name implies, provides a factory for constructing most nodes in the tree. If you need to construct nodes for some reason, you are strongly encouraged to use this.
- Assistants providing a series of generic functionalities.
- Lex classes providing an implementation of the various kinds of tokens present in a VDM grammar.

As for the tree itself, it is very large and complex (around 300 nodes). This is because VDM itself is a rather large language (3 dialects) with a lot of syntax and the Overture tree follows the grammar very closely, practically on a 1-to-1 basis. As such the best way to familiarize yourself with the tree is to study the VDM grammar, available in the VDM language manual that ships with Overture (it can also be found in the `documentation` folder of the repository).

Relation	Modules
Depends on:	None
Dependency of:	`Parser`, `Type Checker`, `Pretty Printer` ,`POG`, `Combinatorial Testing`, `Code Generator`, `Interpreter`, `Pro B Solver`

Table 1: Dependency connections for the `AST` module

Parser

This module is responsible for constructing an AST from VDM source files. The Overture parser is handwritten and while quite efficient, is challenging to maintain. The `parser` provides utility classes and methods to turn an input (`String`, `File`...) into an AST. These are used by various modules and are, in general, problematic since the `parser` should not be depended upon by modules that have no concept of VDM sources. Many of these usages are strictly for testing purposes since the only way to construct any but the most trivial AST is to write a VDM source and parse it. As such we have omitted these connections from the idealized diagram shown in Figure 1.

The parser dependency issue should be alleviated by introducing a framework to directly manipulate the AST for testing purposes. In addition, the dependencies on the parser themselves should be made looser since at the moment they exist as static calls to `parser` methods which makes it harder to replace/change the `parser`. It also makes extending the modules problematic since the static calls cannot easily be hooked into. In general, there should be an intermediate facade module to provide parsing (and type checking) functionalities to all other plug-ins.

Suggestion 1: Create a test framework that allows for direct creation and manipulation of ASTs.

Suggestion 2: Remove public static methods from the `parser` modules.

Relation	Modules
Depends on:	`AST`, `Test Framework`
Dependency of:	`Type CheckerPOG`, `Code Generator`

Table 2: Dependency connections for the `Parser` module

Type Checker

This module is used following the `parser` in almost all use cases for Overture. It is responsible for type checking an AST which consists of validating the types and setting them on all nodes across the tree. Much like the `parser`, there are utility methods for type checking an AST but also for producing a type checked tree from a VDM source. These methods are used in similar contexts as the ones from the `parser` and suffer from the same issues. There is also the fact that it is possible to build a type checked tree directly through the `type checker` or in two steps by using first the `parser` and then the `type checker`. In general, this kind of duplication only serves to muddle things. There should be a single way to perform such a basic and standard task.

Suggestion 3: Introduce a canonical way for modules to obtain type checked ASTs.

Unlike with the `parser`, the `type checker` has additional utility methods that are used by the other modules for non-testing purposes. These methods perform various kinds of functionalities such as, for example, testing if a given union type contains a boolean type. Most of these utilities are exported through the assistant system so they do not pose as much of an extensibility issue. However, from a conceptual point of view, while many of these features are crucial, they are not features of the `type checker` itself but rather of the VDM type system.

The `type checker` should be separated into two modules: a `type checker` module that is only responsible for validating a tree and that would contain most of the visitors and code for traversing a tree; and a `type system` module with the various utilities that would allow a developer to more easily interact with the VDM type system. It's possible that these modules would be tightly coupled and the `type checker` would surely make extensive use of the `type system`. But this would allow other modules to disconnect from the `type checker` which would be important since the other modules have no notion of type checking a tree. For them, all trees are type correct.

Suggestion 4: Separate `type checker` into two modules: one for providing type validation of ASTs and another providing utilities to interact with the VDM type system.

Relation	Modules
Depends on:	`AST`, `Test Framework`, `Parser`
Dependency of:	`Pro B Solver`, `POG`, `Interpreter`, `Code Generator`

Table 3: Dependency connections for the `Type Checker` module

Pretty Printer

This module is used for printing an AST, i.e., converting it to a `String`. The module depends only on the `AST` and can be used by any module that wishes to display a tree to the user. At the moment, that includes only the `POG`. The module is implemented mostly as a series of visitors that walk the tree and produce a `String` for any given node.
Please note however, that this module is very outdated and incomplete and has mostly been discontinued. Its functionality can be (badly) approximated by `toString()` methods in the AST nodes and that is what most modules end up using.

Relation	Modules
Depends on:	`AST`
Dependency of:	`POG`

Table 4: Dependency connections for the `Pretty Printer` module

4 Discussion

This section discusses architectural issues that were uncovered throughout the writing of the architectural guide. Most issues fall in one of two categories: design issues that indicate an unsound architecture and accessibility issues that limit a new developer's ability to navigate the code.

4.1 Design Issues

There are various issues with the architecture of the core modules, in terms of how they depend on each other. Figure 1 shows an idealized design but in fact that design has problems. And it omits several other dependencies that are planned to be removed. The full dependency graph can be seen in Figure 2 and it is more complex and disorganized, particularly the connections between the various plug-in modules.

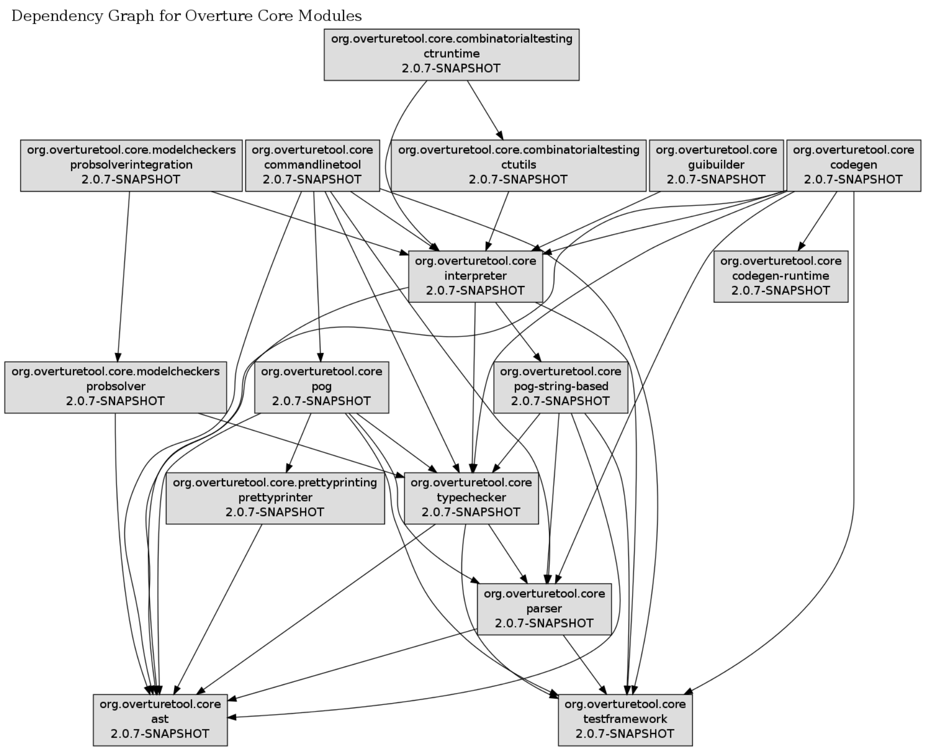
Dependency Graph for Overture Core Modules
org.overturetool.core.combinatorialtesting ctruntime 2.0.7-SNAPSHOT
org.overturetool.core.modelcheckers probsolverintegration 2.0.7-SNAPSHOT
org.overturetool.core commandlinetool 2.0.7-SNAPSHOT
org.overturetool.core.combinatorialtesting ctutils 2.0.7-SNAPSHOT
org.overturetool.core guibuilder 2.0.7-SNAPSHOT
org.overturetool.core codegen 2.0.7-SNAPSHOT
org.overturetool.core interpreter 2.0.7-SNAPSHOT
org.overturetool.core codegen-runtime 2.0.7-SNAPSHOT
org.overturetool.core.modelcheckers probsolver 2.0.7-SNAPSHOT
org.overturetool.core pog 2.0.7-SNAPSHOT
org.overturetool.core pog-string-based 2.0.7-SNAPSHOT
org.overturetool.core.prettyprinting prettyprinter 2.0.7-SNAPSHOT
org.overturetool.core typechecker 2.0.7-SNAPSHOT
org.overturetool.core parser 2.0.7-SNAPSHOT
org.overturetool.core ast 2.0.7-SNAPSHOT
org.overturetool.core testframework 2.0.7-SNAPSHOT

Fig. 2: Dependency Graph for Core Modules

There are two kinds of dependency issues, though they are related and have a common cause: first, multiple modules depend on the same set of modules: `parser`, `type checker` and `ast`. This effectively introduces tight coupling between all three modules. There is some reasoning for this. As we stated before, none of the three functionalities makes much sense on its own and, in general any changes introduced in one of the modules will necessitate changes in the others. Imagine introducing a new type of node in the `ast`. Such a node would require changes to the `parser` (in order to construct the new node) and `type checker` (in order to validate the new node).

This tight coupling between the 3 modules might suggest that they should be merged but that would instead lead to a very large module that would be harder to maintain. But the current design introduces too many dependencies between the modules and even though the `parser`, `type checker` and `ast` all implement a single core functionality, there are multiple points to access it.

One possible solution to this issue is the introduction of a new module that acts as an interface between the 3 core modules and all the remaining ones. Such a module exists in the IDE modules, but not in the core set. The existence of such a core module and subsequent detachment of the `parser` and `type checker` from the rest would also force the removal of hard coded dependencies which, at the moment, greatly hinder the extensibility of the tool and make it much harder to replace the existing `parser` and `type checker` with new versions.

Suggestion 5: Introduce a core module that provides a unified interface to the `parser`, `type checker` and `ast` modules.

The second kind of design issue encountered lies in the interdependency of the functionality modules (i.e., the ones other than `ast`, `parser` and `type checker`). For example, the `combinatorial testing` module depends on the `interpreter`. Most of these dependencies can be justified on a case-by-case basis. In the aforementioned example, the `combinatorial testing` module needs the `interpreter` to execute the texts it generates.

However, these kinds of interdependencies cause problems. Once again, they make it harder to change, extend or replace a given modules. And further, they lack justification from a conceptual point of view. A module should not depend on another module if their functionalities do not intersect. Returning to our example, the responsibility of `combinatorial testing` is to generate tests, not execute them.

All of these extra dependencies have the same common cause: there is no overall module that controls and coordinates the others to present functionality to the user. Each module is more or less responsible for providing a base user interface (in some cases – such as the `POG` – this is done through the `interpreter`). There is a clear lack of a user interface module. This issue does not occur in the IDE modules but if one of the goals for the core modules is to have a basic command-line interface, then there should be a module dedicated to it.

Suggestion 6: Introduce a core module solely dedicated to providing a basic command-line interface.

4.2 New Developer Issues

The main architectural issue that new developers will face has to do with the necessity to use assistants. Assistants are classes that provide centralized access to common functionalities and they are implemented in an extensible way. However, the documentation surrounding them is quite poor. For starters, it is not immediately obvious how a developer hooks into the assistant hierarchy. Furthermore, when constructing a new extension, the need to hook into the assistants is not explicit. Even further, if one hooks into the assistants, it is not immediately clear which methods of which assistants need to be overridden. To answer such questions would require deeper analysis of the code (including the new extension), perhaps with code slicing techniques.

In general, while the current assistant architecture supports extensibility, its usage is not particularly simple. There are two possibilities to address this issue: one is to overhaul the architecture entirely and look to remove the assistants. Another is to greatly increase the assistant documentation to raise further awareness of these issues. On the subject of further documentation, the functionalities of the assistants themselves are also poorly documented. The construction of an assistant API document is underway and looks to remedy that. Finally, it is worth noting that many of these assistant issues only affect language extensions, in the vein of the COMPASS project [2].

Suggestion 7: Revise usage of assistants either through detailed documentation or a new design and implementation.

A final issue worth mentioning is one of terminology. There are two fundamental terms in the Overture developer documentation that are overloaded: *core* are *plug-in*. Both are used to refer to different concepts in different circumstances. In a source code perspective *plug-in* is used in the sense of an Eclipse plug-in: a series of classes and configuration files that implement some kind of user interface in the Eclipse framework. These plug-ins usually correspond to a Maven module. On the other hand, from a functionality perspective, *plug-in* is used to refer to a non-basic functionality of Overture, however many modules it may comprise.

As for the term *core*, from a source code perspective it refers to modules that implement a functionality (as opposed to an IDE interface). Relatedly, when talking about Eclipse plug-is, *core* is used to refer to the imported `jar` file that provides the functionality exposed to the user by the plug-in. From the perspective of functionality, *core* refers to the group of fundamental Overture functionalities: parsing, type checking and AST.

The fact that these two overloaded terms are both central to any discussion of the Overture code base and that they are often used together can lead to confusion when discussing them. For experienced developers, distinguishing between the various meanings is quite intuitive but for a new developer – the kind for whom documentation is primarily intended – it may not be so clear. It might be worthwhile to consider disambiguating the two terms by introducing a few new terms to refer to each specific meaning.

Suggestion 8: Disambiguate terminology for *core* and *plug-in*.

5 Conclusion

In this paper, we have introduced the guide to the Overture architecture, a new effort to improve the existing documentation of Overture. As it stands, the guide is a work in progress and we hope to continue expanding it until it covers the totality of Overture. Even at that point, the major goal for the guide is for it to be continuously updated as the architecture of Overture evolves.

Several issues were encountered during production of the guide. Many of them stem from a lack of existing documentation so there is an expectation that this guide (and others like it) will alleviate various issues as it improves. On the other hand, the guide has uncovered various architectural design issues that may require significant changes to Overture. This paper provides some initial discussion and suggestions on how to address those issues.

Finally, while the intended target audience of the architecture guide are new developers of Overture, we hope it draws the interest of the Overture community at large. The current lack of new developers is a real issue, and part of the reason for it is that the code base is quite challenging. This leads to something of a vicious cycle where the absence of new developers lowers the need for documentation and the absence of documentation makes it harder to attract new developers. We hope that this guide is a first step towards breaking the cycle and that it may help attract new members to the Overture community.

Acknowledgements

This work is partially supported by the EU Framework 7 Integrated Project "Comprehensive Modelling for Advanced Systems of Systems" (COMPASS, Grant Agreement 287829). For more information see `http://www.compass-research.eu`.

References

1. Apache: Maven Homepage. *http://maven.apache.org/* (2014)
2. Coleman, J.W., Malmos, A.K., Larsen, P.G., Peleska, J., Hains, R., Andrews, Z., Payne, R., Foster, S., Miyazawa, A., Bertolini, C., Didier, A.: COMPASS Tool Vision for a System of Systems Collaborative Development Environment. In: Proceedings of the 7th International Conference on System of System Engineering, IEEE SoSE 2012. pp. 451–456 (July 2012)
3. 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`

Concurrency, Rely/Guarantee Thinking and Separation Logic

Cliff B. Jones

School of Computing Science, Newcastle University, NE1 7RU, UK
`cliff.jones@ncl.ac.uk`

Abstract. This is a summary of a talk given at the 2014-06-21 Overture Workshop.

Extended abstract

The talk was largely based on [5] (which originated from an invited talk at SEFM [4] but was almost completely rewritten) — as hinted by the title, that paper argues that "expressive weakness" might not be a fault in specification languages for concurrency. The point being that more expressive notations might be less tractable.

Furthermore, it is argued that it is a mistake to focus too early on specific notations; it is far better to first study the "issues" in concurrency and then try to see how various approaches can help describe and support reasoning about the issues. As the title of the talk suggests, both Rely/Guarantee approaches and Separation Logic were a particular focus.

"Separation" is certainly an issue in concurrency and the extension of John Reynolds' Separation Logic [8] to Concurrent Separation Logic (CSL) [7] provides a compact way of reasoning about separation. In fact, the related issue of "ownership" between concurrent processes is also well served by CSL. In order to make the point about starting from the issues, it was pointed out that VDM descriptions use **rd/wr** clauses in operation descriptions to delineate the "frames" of operations. It is also interesting to compare some of the many papers that provide formal justifications of Simpson's implementation [9] of Asynchronous Communication Mechanisms. Race freedom on the four slots is of paramount importance and exchange of ownership of the slots between the reader and writer processes is delicate. In [1] and [2] an approach is used that actually uses both rely/guarantee argumentation and linearisability; in contrast [6] shows that it is possible to reason about the exchange of ownership at an abstract level. (In passing, that same paper introduces a notation for the "possible values" of a variable to which concurrent processes have write access — this idea is still being worked out.)

The talk also made the point that the form of separation exhibited in the in-place list reversal algorithm in [8] might be conveniently handled by abstraction. With the motto "separation is an abstraction", it is interesting to consider what form of notation might best handle reifications whose task is to preserve the abstraction of separation (this example is touched on in [5] and is still the subject of on-going work).

Lest the impression is created that the talk was just about changing Separation Logic, the majority of the time was spent on recent changes to the way that Rely/Guarantee specifications and reasoning can be recorded. Here, of course, the issue is interference which is central to many concurrent designs. The original five-tuple presentation of Rely/Guarantee specifications is replaced in [3] by **rely** and **guar** statements that can be wrapped around any command — and here, as in the refinement calculus, commands can include specifications.

The material covered is from the (UK) EPSRC project "Taming Concurrency" and the (Australian) ARC project "Understanding concurrent programs using rely-guarantee thinking". I am grateful to all of my colleagues on these projects: Andrius Velykis, Nisansala Yatapanage, Ian Hayes, Rob Colvin, Larissa Meinike and Kim Solin.

References

1. Richard Bornat and Hasan Amjad. Inter-process buffers in separation logic with rely-guarantee. *Formal Aspects of Computing*, 22(6):735–772, 2010.
2. Richard Bornat and Hasan Amjad. Explanation of two non-blocking shared-variable communication algorithms. *Formal Aspects of Computing*, pages 1–39, 2011.
3. Ian J. Hayes, Cliff B. Jones, and Robert J. Colvin. Laws and semantics for rely-guarantee refinement. Technical Report CS-TR-1425, Newcastle University, July 2014.
4. Cliff B. Jones. Abstraction as a unifying link for formal approaches to concurrency. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 1–15, October 2012.
5. Cliff B. Jones, Ian J. Hayes, and Robert J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, (in press), 2014.
6. Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 23(3):289–306, 2011.
7. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.
8. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th LICS*, pages 55–74. IEEE, 2002.
9. H. R. Simpson. New algorithms for asynchronous communication. *IEE, Proceedings of Computer Digital Technology*, 144(4):227–231, 1997.

Abstracts of Other Workshop Contributions

Tools Development Update
Author Joey Coleman
Abstract We will give an overview of the nut and bolts state of how development, and what we're currently working to improve upon.

Design Space Exploration through Co-modelling and Co-Simulation - the Pacemaker Challenge
Authors John S Fitzgerald, Carl Gamble, Peter Gorm Larsen and Martin Mansfield
Abstract We present a study aiming to demonstrate that co-modelling and co-simulation can be used to explore design alternatives in the context of the pacemaker challenge problem. Specifically, we show the use of VDM as a discrete-event formalism modelling the controller, coupled to a continuous time model of the leads and heart environment represented in 20-sim. Possibilities for the exploration of design alternatives through co-simulation are illustrated by examining the change from synchronous to asynchronous pacing modes in the presence of noise.

Teaching with Crescendo
Authors Ken Pierce
Abstract We describe the use of Crescendo, Overture and 20-sim in the design and delivery of an undergraduate course on Real-Time and Cyber-Physical Systems. The underlying goal is to be able to teach Computing Science students the basic elements of real-time control using co-simulation of VDM-RT with 20-sim, instead of direct control of real laboratory robots. Based on the experience in delivering the course, we have developed fresh tutorial material for Crescendo, and piloted it at FM 2014 this year.