FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO

FEUP

Automatic Generation of Graphical User Interfaces From VDM++ Specifications

Carlos Alberto Loureiro Nunes

Integrated Masters in Informatics and Computing Engineering

Supervisor: Prof. Ana Paiva

July, 2011

Automatic Generation of Graphical User Interfaces from VDM++ Specifications

Carlos Alberto Loureiro Nunes

Integrated Masters in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Rui Pedro Amaral Rodrigues (Ph.D.)

External Examiner: Manuel Alcino Pereira Cunha (Ph.D.)

Supervisor: Ana Cristina Ramada Paiva Pimenta (Ph.D.)

July, 2011

Abstract

This document describes the research work done under the subject of “Automatic Generation of Graphical User Interfaces from VDM++ Specifications”. The project contributes to the field of Software Engineering, more specifically to the areas of Formal Methods and Model-based development. The goal of this work is to provide a useful tool to assist in the formal methods area of software engineering. That is, the main objective of this research work is to develop an approach of automatic graphical user interface generation from VDM++ specifications (a formal mathematical model), and a tool that implements it. Besides being capable of interacting with the underlying mathematical model, the generated user interface must be easy to use. As such, the user interface elements and their layout were studied in detail.

To put the developed tool into context, related work is described. This includes related development tools, and a description of their approaches to the main problems they focus on.

Despite of the large quantity of research done in the broad field of graphical user interfaces, research on the particular subject of this work is very limited or not well known. As such, it was essential to combine and adapt existing user interface development techniques to achieve the goals of this project.

In order to evaluate the flexibility and suitability of the solution, a Java prototype tool, integrated with existing VDM++ tools, was developed. This prototype was used in a case study for result analysis.

The conclusions of this work although focusing on VDM++ graphical user interface generation, can be, in part, extended to automatic graphical user interface generation in general.

Finally, as possible future work, more creative automatic graphical user interface generation approaches are presented.

Resumo

Este documento pretende descrever o trabalho realizado sob o tema de investigação "Automatic Generation of Grapical User Interfaces From VDM++ Specifications" ("Geração Automática de Interfaces Gráficas a partir de Especificações VDM++"). O presente projecto insere-se no campo de Engenharia de Software, mais especificamente na área de Métodos Formais e "Model-based Development". O seu intuito é proporcionar uma ferramenta útil para assistir no processo de desenvolvimento de software na área de métodos formais. Por outras palavras, o objectivo deste trabalho de investigação, tal como o tema indica, é formular uma abordagem (e uma ferramenta que a implemente) de geração automática de interfaces gráficas a partir de uma especificação VDM++ (um modelo matemático formal). Esta interface gráfica além de ser capaz de interagir com o modelo matemático subjacente, deve igualmente garantir uma usabilidade elevada e fácil interpretação. Como tal a apresentação dos diferentes elementos gráficos e a sua disposição no todo da interface foram as características estudadas com a maior cautela.

Para contextualizar a ferramenta desenvolvida, são apresentados alguns exemplos de ferramentas de desenvolvimento relacionadas, descrevendo as suas abordagens principais, incluindo os problemas de desenvolvimento focados nesses exemplos.

Apesar da grande quantidade de investigação na área de interfaces gráficas, em geral, a investigação no âmbito restrito deste projecto é reduzida ou pouco divulgada. Como tal foi essencial a conjunção de diferentes abordagens aplicadas em projectos com propósitos diferentes que recorressem a soluções adaptáveis ao projecto em questão. Visto só ser possível atingir o objectivo proposto através da incorporação de diferentes soluções de problemas paralelos.

Para demonstrar a flexibilidade e aplicabilidade da solução construída, foi implementada um protótipo funcional, em Java, integrado com ferramentas VDM++. Este protótipo foi num caso de estudo na análise de resultados.

As conclusões apresentadas focam, em particular, o caso de estudo considerado segunda a abordagem proposta. Estas conclusões inferidas podem ser, em parte, estendidas à geração automática de interfaces gráficas em geral.

Por fim, são apresentadas, como possível trabalho futuro, abordagens de geração de interfaces gráficas mais criativas do que abordagem apresentada.

Acknowledgements

To my supervisor, Professor Ana Paiva for her guidance throughout the duration of this research work.

To the professors of the Integrated Masters on Informatics and Computing Engineering course for their teachings.

To my family and friends that along the years have one way or another, small or large, contributed to making me capable of accomplishing this research work.

Carlos Alberto Loureiro Nunes

Contents

List of Figures

Abbreviations and Symbols

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
DHTML	Dynamic HTML
GTK	Gimp ToolKit
GUI	Graphical User Interface
IDE	Integrated Development Environment
IR	Intermediate Representation
JAXB	Java Architecture XML Binding
JML	Java Modelling Language
LLVM	Low Level Virtual Machine
RTF	Rich Text Format
UML	Unified Modelling Language
VDM	Vienna Development Method
VICE	VDM In a Constrained Environment
W3C	World Wide Web Consortium
XML	eXtendedMarkup Language

Chapter 1

Introduction

This dissertation describes research work done under the topic of "Automatic Generation of Graphical User Interfaces From VDM++ Specifications", introducing related work, the main problems to address, goals, and defining an automatic generation approach.

1.1 The Project

The growing complexity of hardware and software systems means that the introduction of errors is also more likely to occur. These errors, in some cases, can lead to catastrophic results, such as substantial losses of money and/or time, and even the loss of human life. One of the objectives of software engineering is to allow the development of software with a low fault rate, despite its complexity. A method to achieve this goal is the use of formal methods.

The subject of formal methods is more extensively introduced in chapter 2, but, very briefly, formal methods are development methods that focus on the system specification and verification phases. These methods typically include a formal language used to construct a model of the system to implement. This research work deals with one of these formal languages, VDM++.

This research work puts forward an approach to allow the automatic generation of a graphical user interface from a VDM++ specification, capable of interacting with the underlying model.

1.2 Motivation and Context

The automatic generation of a graphical user interface (GUI) from a VDM++ specification is a high utility feature in software development. It can be used in iterative GUI development, generating a functional interface from a specification, and is equally useful in the debugging process, discarding the use of a VDM++ interpreter to interact with the specification. As this stands, the main motivation for this work is automatic generation.

Another motivation for this work is the study and analysis of the different possible solutions that in the context of this research include a wide array of methodologies and technologies. Not overlooking that the automatic generation of graphical interfaces from VDM++ specifications has been the subject of little research by the community. One of the main difficulties in this transformation of text elements into visual elements is the fact that this formal modelling language possesses no element oriented for graphical interface design. As such, the main challenge of this work is to explore different methodologies of inferring graphical user interfaces and/or possible extensions to the specification language in order to include descriptive elements of a graphical user interface (GUI).

1.3 Objectives

The main goal of this work is to define an automatic generation of graphical user interfaces from a VDM++ specification approach. Producing an implementation, in order to evaluate the approach, and proceed to its analysis. Others goals are integrating the graphical user interface with existing tools, and multiplatform support.

As such the goals can be described as:

VDM++ Specification Analysis – That is, the approach must be able to conduct an adequate analysis of the specification in order to obtain the necessary information for generating the GUI.

GUI Definition – The design of the graphical user interface, its components and layout, must suit the VDM++ specification. And must be carried out using only information collected from the specification, not relying on any external guidance.

GUI Generation – Functional code describing the graphical user interface is automatically generated.

GUI Interaction with the Model – The generated user interface is able to interact with the underlying model of the VDM++ specification.

1.4 Document Structure

This document is divided in five chapters, starting by this introductory chapter. The next chapter presents the bibliographical revision. Introducing basic concepts, as well as related work to help establish what are the main contributions this research work has to offer.

The third chapter describes the main problem, an approach to it, and a possible implementation that is the basis of the tool (VDM++ GUI Builder) produced by this work.

The fourth chapter presents a case study of the approach, using the VDM++ GUI Builder on a VDM++ specification. Including a description of the case study, results and their discussion, ending with a set of extracted conclusions.

The fifth and final chapter, exposes the main conclusions and possible future directions of the research work.

Chapter 2

Bibliographical Revision

This chapter aims to describe related work. As such, it focus on methodologies and tools that can contribute to a better understanding of the basic concepts of this work, and provide the starting point to this dissertation.

2.1 Introduction

In order to explore the topic of "Automatic Generation of Graphical User Interfaces From VDM++ Specifications" this chapter introduces three main subjects: existing practices of graphical user interface development; formal methods, and associated tools; and automatic code generation.

As the research work focus on graphical user interface development it is important to conduct a survey on existing techniques and tools, to ascertain if there are previous approaches that can serve as guidance, or that can be adapted to satisfy the objectives of this work.

The specification language in which this work focuses on is VDM++. The features of the language will be presented and its main elements described. This is useful in order to know which elements the language possesses that can be used for GUI development, and see the need for automatic GUI generation processes for the VDM++ formal method.

To assist in exploring the main problems and approaches to automatic GUI generation, the topic of automatic code generation is introduced, as there are similarities in terms of problems.

2.2 Graphical User Interface Development Tools

The development of Graphical User Interface (GUI) is, currently, tied to the use of tools and techniques that support the design and implementation of the interfaces. These tools and techniques vary according to the main problem they focus on and use different approaches in order to achieve the common goal of assisting the developer.

2.2.1 Graphical Toolkits

Window managers in graphical environments provide a basic programming model to draw and refresh graphics, and to collect input data. However, programming at this level is very costly in terms of time, and very tedious. Furthermore, requiring each developer to create user interface elements from the ground up makes it very difficult to maintain a level of consistency.

Graphical toolkits address these issues by providing a user interface element library (widgets) and a framework to manage the interface operations constituted by these components. By using an existing graphical user interface framework and a library of reusable components, the graphical user interface development process becomes easier than building user interfaces from the ground up [1].

However, they do not free the developer of designing and adding the desired functionality to the interface. There are currently several graphical toolkits, developed in the context of the graphical environment of a operation system or a programming platform. Examples include: GTK [2];the graphical toolkit provided by the Qt development platform [3]; Java Swing [4], the toolkit provided by the Java platform; Cocoa [5], the MacOS X environment graphical toolkit; and the Windows environment graphical toolkit, Winforms [6].

Despite the large number of existing graphical toolkits, with their own library of user interface elements, each with numerous options or attributes, there are common categories, such as: buttons, text boxes, labels, combo boxes, and check boxes.

2.2.2 Interactive Graphical Tools

Also called GUI builders, this type of tool makes it possible to “drag and drop” interface components into place, in order to create windows and dialogs. Leaving to the developer the task of coding the actions associated to a given interface. In this manner the developer can instantly see the final result. Something that is not always straightforward when coding the GUI.

This kind of tool gained its momentum with the NeXT Interface Builder [1]. Two examples of such tools, currently in use, are the Glade interface builder [7] and the interface builder component of the NetBeans integrated development environment [8].

2.2.3 Graphical User Interface Markup Languages

Conventional programming methods to develop a GUI use a specific programming language, and often lead to the creation of repetitive, sometimes error prone, and frequently complex code. User Interface Markup Languages address these problems by describing the GUI in a markup language, usually dialects of XML. Relying on sub-applications to interpret and transform the GUI description into program code. This approach, besides reducing the amount of written code, makes it easier for the developer to concentrate on user interface design (layout and interface elements), instead of functionality [9].

Examples of user interface markup languages include UsiXML, XAML, XUL and SwiXML.

However these languages still rely on the developer to insert functionality using a more conventional approach.

2.2.3.1 UsiXML

UsiXML (User Inteface eXtensible Markup Language) is a XML language capable of describing textual, graphical, audio and multimodal interfaces. That is, applications with different interaction techniques, types of use and computing platforms can be described independently of platform specifics [10, 11].

In brief the language has the following characteristics:

- It is directed to inexperienced programmers, or even non-programmers. Which, of course, does not exclude it from being used by experienced developers.
- The language keeps record of the interface essentials independently of physical characteristics.
- UsiXML describes the user interface elements using a high abstraction level. Thus allowing the development of graphical interfaces in several toolkits. That is, an interface described in UsiXML can be easily used with any toolkit that implements this language, as it has no toolkit specific characteristics.

2.2.3.2 XAML

XAML is a declarative annotation language, based on XML, created by Microsoft. Applied to the programming model of the .NET platform, it is used to simplify the development of graphical user interfaces. It separates the user interface logic from execution logic.

The language directly represents object instantiation. This differentiates it from other user interface markup languages that are, typically, interpretative in nature, and are not explicitly tied to one development platform – there is nothing intrinsically cross-platform in XAML.

It is not expected that developers actually write XAML; the language is more the output notation of Microsoft GUI builder tools [9, 12-14].

2.2.3.3 XUL

XUL (XML User interface Language) was created by the Mozilla Corp. with the goal of facilitating and accelerating the development of their browser. It is a XML language and was developed with the main goal of assisting in the creation of portable graphical applications. The language possesses a set of interface components, such as text boxes, buttons, sliders, and a number of supporting technologies to help with the construction of XUL applications [15].

In short, the language has the following main characteristics:

- XUL is based on existing W3C standards. Applications that use XUL also use technologies like HTML 4.0, JavaScript 1.5, XML 1.0 and CSS 1 and 2.
- The language was designed not to depend on a specific platform. As XUL provides abstraction of user interface elements, it is possible to port a XUL application to several computing platforms, without the need of code alteration or recompiling – on the condition that the platform is supported by XUL.
- XUL separates presentation logic from application logic. The layout of XUL applications can be altered independently of its application logic or definition.

2.2.3.4 SwiXML

"SwiXML, is a small GUI generating engine for Java application and applets. Graphical User Interfaces are described in XML documents that are parsed at runtime and rendered into javax.swing.objects" [16]. SwiXML is also the name given to the XML language that is used to describe the graphical user interfaces. The language has the following characteristics:

- SwiXML focuses completely on the Java Swing graphical toolkit. The XML tag names closely resemble Java Swing classes.
- Despite supporting almost all Swing objects, the size of this tool is approximately 40 Kbyte.
- User interface behaviour has to be coded in Java. SwiXML, although providing methods to ease the task of adding functionality to the interface, such as action generators, focuses only on GUI generation.

2.2.4 Formal Language-Based Tools

The motivation behind the use of existing formal method based techniques is a strong emphasis on dialog management [1]. Which for example, in typical graphical installation user interfaces is indeed a very important aspect. However, outside of this user interface style, dialog management by the system does not contribute to having a shortest path between windows.

Other problems with this kind of tool are the difficulty of expressing unordered operations, thus the interface would have a very rigid sequence of required actions; and the need for the developer to learn a new special purpose language.

2.2.5 Property Models

Graphical user interfaces usually possess dependencies between values manipulated by the user interface, that lead to conditionally enabled GUI elements. The implementation of this aspect of a user interface is time consuming and once again leads to repetitive code. This is the problem property models address.

By maintaining an explicit model of dependencies between parameters of a command, property models can then be used by reusable algorithms to implement enabling or disabling of user interface elements.

But as stated, the model needs to be explicitly defined, requiring the use of a special purpose language or similar construct [17].

2.2.6 Constraints

"A constraint can be thought of intuitively as restriction on a space of possibilities (…) Mathematical constraints are precisely specifiable relations among several unknown (or variables), each taking a value in a given domain)" [18]. This concept can be used to implement several different aspects of a user interface. Two examples of such a tool are Amulet [19, 20] and Subarctic [1].

Relying on constraints, a user interface designer can, for example, easily define that a line has to be attached to a button. In the same way, the colour, position and size of an object can be derived from a relationship with another object expressed by a constraint. At the end, a constraint solver is used to find a solution.

These types of systems offer a simple and declarative specification for implementing a user interface, however, according to the bibliography, they are not used beyond research environments. One of the reasons for this is the inherent unpredictability of the resulting user interface.

The solver will try to find a solution that satisfies all constraints. When several solutions exist, the solver may find one that was not expected by the interface designer.

Another difficulty lies in the debugging of a set of constraints, as locating the bug may not be easily done. A related problem is the need by some solvers, to build the set of constraints in a particular form (for example, in a linear form), or the need for the developer to know some details of how the solver works. Also, it can prove to be difficult to master the declarative programming paradigm of constraints as most developers are used to imperative programming languages – in which, the way to approach problems is different.

Nevertheless, constraints are widely used for layout control. NeXTStep, for example, provided a limited form of constraints that could be used to control layout [1]. This form of constraints gained a fair share of usage as the results were more predictable to developers, and was also easier to use. The Java platform also makes use of constraints in the form of layout managers [4].

2.2.7 Automatic Model-Based Techniques

The goal of these tools is to free the developer from GUI implementation details, allowing him to focus on developing functionality. The motivation for this kind of tools may be the rapid development of quality user interfaces; endowing programmers with little to no experience in building user interfaces, the capacity to create high quality user interfaces; automatically creating user interfaces suited for a wide range of platforms, without the need of additional work; integrating the user interface development process into a larger, formal based development process [21].

Early examples of such tools are UIDE [1] and HUMANOID [22]. These systems used heuristic rules to select the suitable elements and layout, as well as other details of the user interface specified by the model. A more recent example of an automatic model-based technique, generates user interfaces from UML domain and use cases models [23].

A common disadvantage in the use of these techniques is the degree of unpredictability. When heuristics are involved, the final result of the user interface specification may be difficult to predict. Another common disadvantage is the need to learn a special purpose modelling language. And due to the inherent difficulty of automatically generating user interfaces, this kind of tools typically place significant limitations on the type of user interfaces they can produce. This usually leads to the generated user interface being not as good as one created by more common programming techniques [1, 24].

Although not strictly an automatic model-based technique, it is relevant to introduce in this section the naked objects architectural pattern [25]. The main idea of naked objects is to encapsulate all business logic into domain objects. That is, all information and all behaviours that can be applied to, for example, a book, should be contained in the same object. Thus allowing the automatic generation of a graphical user interface just by exposing the domain objects to the user. This technique, although not requiring the use of models to generate the user interface, conditions the architecture of the underlying system.

2.3 Formal Methods

Formal Methods, in the context of software engineering, are a set of mathematical based languages, techniques and tools to specify and verify systems, in order to develop reliably systems despite their complexity [26]. The use of formal methods does not guarantee correctness, but can reveal system inconsistency, ambiguity and omissions that otherwise could pass undetected.

For specifying the system and its properties in great detail a formal method uses a specification language, with mathematical based syntax and semantics. The properties of the system can include functional behaviour, performance characteristics or internal structure. These specification languages, unlike most programming languages do not translate directly into executable code, and are to be used in both system and requirement analysis, and system design. They are meant to describe a system at a much higher level than common programming languages such as Java or Python.

As for system verification, formal reasoning techniques are used. These techniques can verify that the defined properties remain true in the specified system. A well established approach for system verification is model checking [26]. This technique consists on creating a finite model of the system – allowed by the use of a formal specification language – and check if a given property holds true in the model. The verification is executed as an exhaustive search of the possible states of the system. The search always terminating, as the model is finite [26, 27].

Formal methods are "particularly desirable in safety-critical applications such as Air Traffic Control, medical systems, railway signalling and many others" [28].

2.3.1 Vienna Development Method (VDM)

The Vienna Development Method is one of the oldest formal software development methods in existence [29]. Initially developed in the IBM laboratories of Vienna, Austria in the 1970s, the method specified a software development model based on: the collection of requirements; the creation of an abstract model of the desired system; and subsequent software implementation. In order to specify the abstract model, a meta-language was also defined [30].

As the method evolved, it grew to include a set of techniques and tools based on the VDM-SL formal language, in turn created from the previously defined modelling meta-language.

VDM has a long history of successful industrial application, as well as a successful track as a research tool [27, 31, 32]. As a recent example, VDM was successfully used in the development of firmware for a Smart Card IC Chip for mobile phones [33].

2.3.1.1 VDM++

The VDM++ formal specification language is itself an object-oriented version of the VDM-SL formal language. Due to its relevance to this work it will be described in greater depth than the VDM-SL formal language.

As with VDM-SL, VDM++ allows the definition of invariants, pre-conditions and post-conditions. Also including the following characteristics: classes, instance variables, operations, functions, types, operators and expressions.

An example of a VDM++ specification modelling a cash dispenser system can be found in annex A.

The language has five basic types: boolean, character, numeric (that in turn can be divided into natural, positive natural, integers and real), quote (analogous to enumerations in languages like Java) and token (analogous to a struct).

VDM++ also includes three collection types – set, seq and map. With each collection possessing a distinct definition syntax and set of operators (see annex B). Avoiding entering into excessive detail, the set type consists of unordered collection without repeated elements of the same type; a seq consists of an ordered collection of elements, allowing repetition; and a map is a finite function relation of elements of type A with elements of type B (analogous to hashtables in Java) [32, 34].

2.3.1.2 VDM Tools

In order to support the development and analysis of formal models using VDM, several tools were developed. This section describes two of such tools, Overture Tools, an open-source solution distributed under the GPLv3 license, and VDMTools, the leading commercial solution.

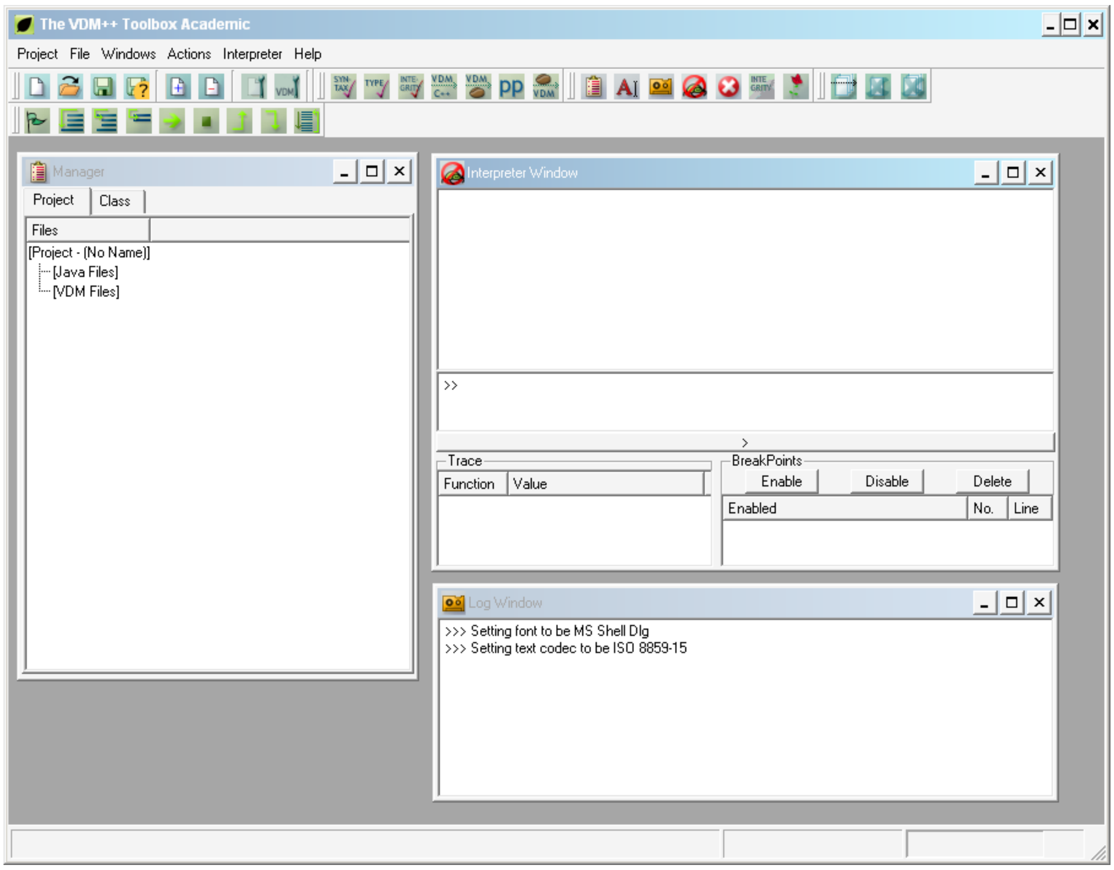
2.3.1.3 VDMTools

The VDMTools tool is in fact a set of tools, that as previously stated, support the development and analysis of system models, specified in the formal language of the Vienna Development Method. Three dialects of the language are supported: VDM-SL, VDM++, and an extension of VDM++ with features to support the modelling and analysis of distributed real-time systems (referred to as VICE or VDM-RT) [35, 36].

The set includes tools for automatic model verification and validation before its implementation. These range from the traditional syntax and type checking tools, to an interpreter to execute the specified model, conducting automatic consistency verification during execution. The interpreter also supports interactive debugging, through the use of: breakpoints; VDM expression evaluation; call stack inspection; instance variable checking.

VDMTools supports as input data VDM++ specifications embedded in Microsoft Rich Text or LaTeX documents. Using these formats it is possible to include descriptive text without the use of a special syntax (like the use of "//" in Java to signal a comment line). Simple ASCII text files are also supported, but in this case it is necessary to use a comment syntax to mark descriptive text.

In figure 1, is depicted the graphical user interface of the tool. Users usually use the graphical user interface instead of the command line interface, also available. The figure shows the interpreter and log viewer windows at the right, and the specification manager window at the left.



```
>>> Setting font to be MS Shell Dlg
>>> Setting text codec to be ISO 8859-15
```

Figure 1: The VDMTools (version 8.1) Interface.

To describe in greater detail the tools included in VDMTools [37], the following list includes a brief description:

- **Specification Manager** – The manager keeps track of the current state of the classes of the specification.
- **Syntax Checker** – Checks if the syntax of the specification follows the VDM language definition.
- **Type Checker** – The tool checks for inappropriate uses of values and operators, and shows all places where execution error might occur.
- **Interpreter rand Debugger** – The interpreter allows the execution of all executable VDM constructs. From simple value constructors, as sequence enumerations, to more elaborate constructions such as exception handling. Thus allowing the use of testing techniques to assist in validation of the specification. Furthermore an executable specification can be the base for a functional prototype. The debugger

supports the usual features present in debuggers for more common programming languages, including breakpoints, stepping, instance variable inspection, and call stack inspection.

- **Integrity Examiner** – The integrity examiner extends the static checking characteristics of VDMTools, by checking the existence of potential sources of internal inconsistency or specification integrity violation. These checks include type invariants, pre-conditions and post-conditions, sequence limits and map domains.
- **Test Facility** – This test module allows the application of a test suite to the specification. Information about test coverage is automatically stored during execution and later presented, showing which parts of the specification where more frequently executed and which were left outside of the test coverage.
- **Automatic Code Generator** – This tool allows the automatic generation of C++ and Java code, from a VDM++ specification. The generator is able to successfully produce executable code from 95% of all VDM constructs. Once a specification has been tested, the generator can be used to rapidly obtain a implementation.
- **Corba Compliant API** – VDMTools provides a Corba API [38] that allows that other programs access a running VDMTools process. This allows the external control of components of VDMTools, such as, for example, the interpreter.
- **Rose-VDM++ Link** – This tool links UML to VDM++. That is, allows the use of UML notation to define a model in structural terms, while the formal notation is used to define functionality.
- **Java to VDM++ Translator** – The translator allows the conversion of Java applications to VDM++ specifications.

As a side note, it is important to refer the existence of a "lite" version of VDMTools, which although possessing fewer features, can be freely downloaded from the official site, after a simple registry.

The tool supports Windows (2000, XP or superior), Linux and MacOS X (10.4 or superior).

2.3.1.4 Overture Tool Project

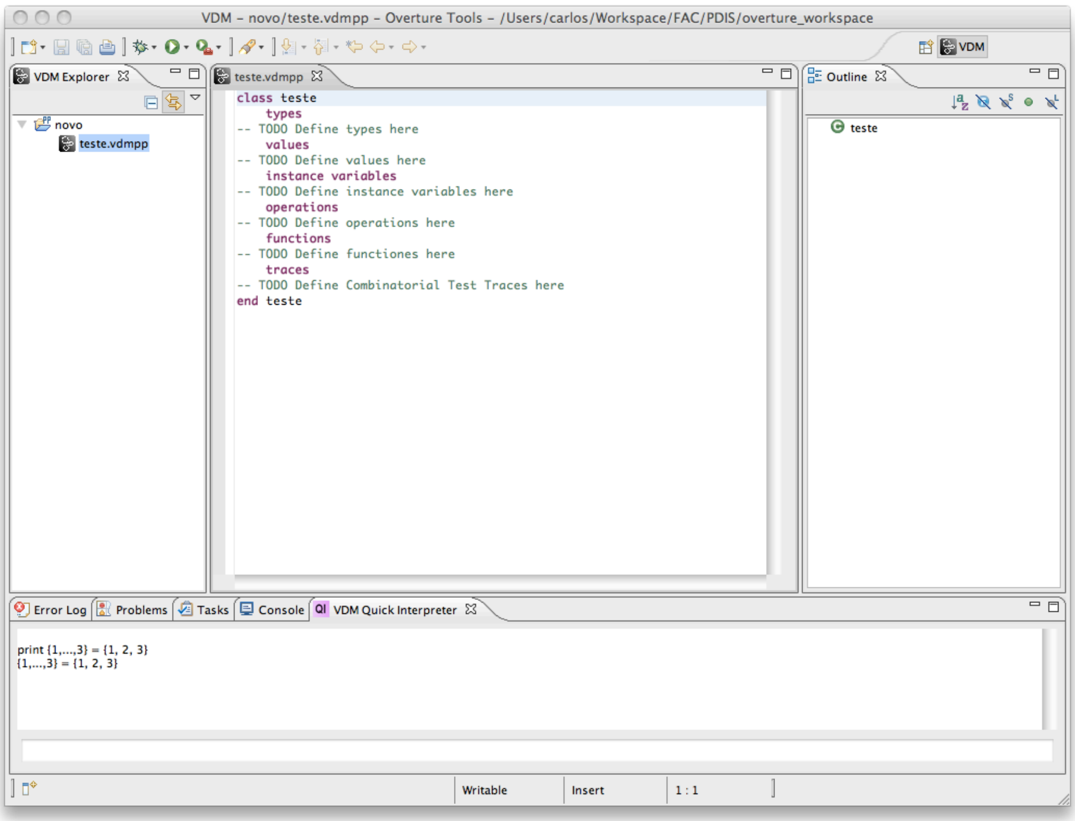
The, open source tool, Overture is currently being developed by a community of volunteers on top of the Eclipse platform [39]. The main goals of the development effort are: produce a high quality industrial grade tool, that can support the creation and development of precise models in any VDM dialect; foster an environment of tool and VDM dialect experimentation and modification.

The tool supports the specification and analysis of computational systems in VDM-SL, VDM++ and VDM-RT/VICE. Providing debugging and testing functionality, in order to support established software engineering practices such as the use of test cases.

By default the VDM specifications are saved to simple ASCII text files, although LaTeX files with embedded specifications are supported as an input format [40].

The underlying VDM engine of Overture, VDMJ [41], provides a simple command line interface, making the use of the graphical user interface provided by Overture (figure 2) not strictly necessary.

The Overture GUI is basically the Eclipse standard GUI, following the same layout conventions: project explorer at the left of the window, code editor at the centre, and code outline view on the right (figure 2).

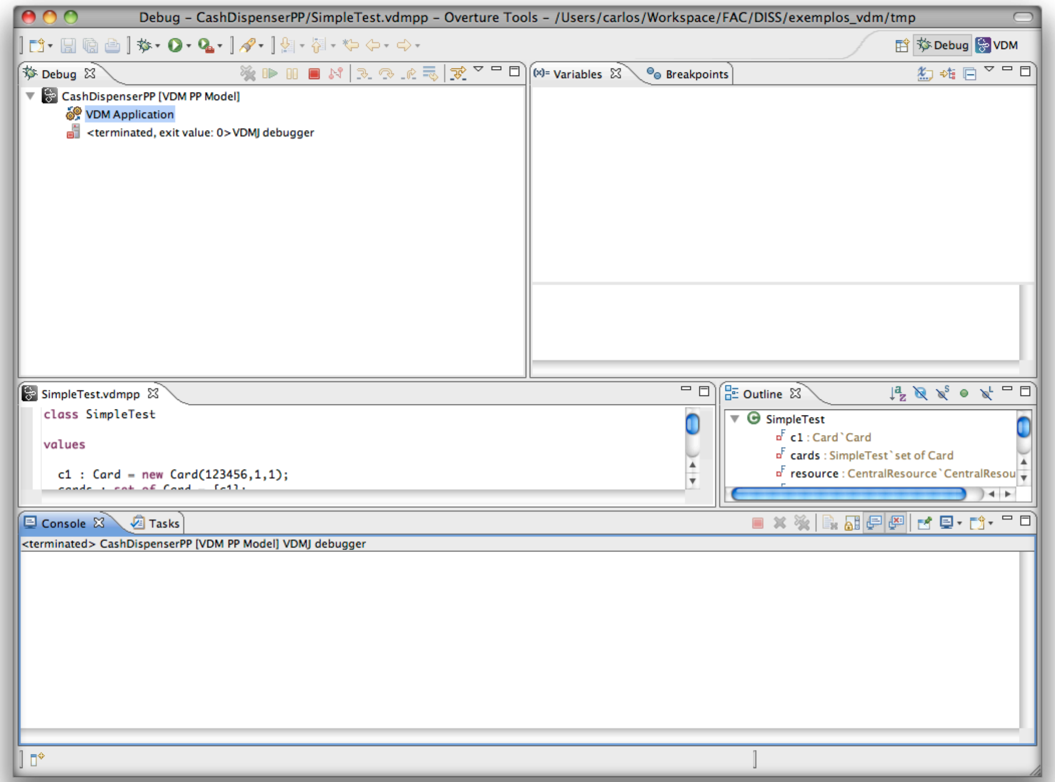


```
class teste
    types
-- TODO Define types here
    values
-- TODO Define values here
    instance variables
-- TODO Define instance variables here
    operations
-- TODO Define operations here
    functions
-- TODO Define functiones here
    traces
-- TODO Define Combinatorial Test Traces here
end teste
```

```
print {1,...,3} = {1, 2, 3}
{1,...,3} = {1, 2, 3}
```

Figure 2: The OvertureTools (version 1.0) Interface

The Overture debug interface also mimics the Eclipse debug perspective (figure 3). Developers with Eclipse experience should have no problem in adapting to the Overture interface.


Debug – CashDispenserPP/SimpleTest.vdmpp – Overture Tools – /Users/carlos/Workspace/FAC/DISS/exemplos_vdm/tmp
Debug
VDM
Debug
Variables
Breakpoints
SimpleTest.vdmpp
Outline
Console
Tasks

CashDispenserPP [VDM PP Model]
VDM Application
<terminated, exit value: 0>VDMJ debugger

```
class SimpleTest

values

  c1 : Card = new Card(123456,1,1);
```

SimpleTest
c1 : Card`Card
cards : SimpleTest`set of Card
resource : CentralResource`CentralResou

<terminated> CashDispenserPP [VDM PP Model] VDMJ debugger

Figure 3: The OvertureTools GUI in debugging mode.

To further describe the features of Overture [42], the following functionality list includes a brief description.

- **Structure Definition Using UML** – The tools allows the translation to a VDM specification of an imported UML specification. Also allowing exporting a VDM specification to UML.
- **Syntax Checking of VDM Models** – Checks if the syntax of a specification follows the rules of VDM dialect.
- **Type Checking VDM Models** – Checks the types used in the specification, namely checking if there is consistency in their use.
- **Interpreter and Debugger** – Besides allowing the execution of VDM constructs, Overture provides functionality, such as setting up breakpoints, stepping and instance variable value checking.
- **Test Coverage** – Overture offers support for the definition and execution of test cases, including compiling information about the test coverage of executed tests. This information can be exported to pdf format.

- **Proof Obligations** – The tool automatically generates proof obligations for VDM models. That is, generates boolean expressions that describe the restrictions in several points of the model that must be maintained in order to prove that the specification is consistent (no error will occur during execution if all restrictions are uphold).
- **Combinatorial Testing** – With the goal of increasing the level of automation in the testing process, the tool allows the use of regular expressions that can be expanded to create a set of individual tests.
- **Remote Control** – This feature enables the remote control of the Overture interpreter. That is, it is possible to delegate the control of the interpreter to an external program, through the use of a purposely-built java class.

Overture is a completely Java based tool so, in principle, supports every platform that possesses a Java virtual machine, including Windows, Linux and MacOS X.

As with VDMTools, or even more so, Overture can be described as a set of tools. "Overture uses a plug-in architecture consisting of components that supply core functionality and components that interact directly with the user through the Eclipse GUI" [40]. That is, the Eclipse platform blends all the tools into one development suite. These tools have complex relationships between them, and are often not completely integrated. For example, Overture has a purposely-built parser, created using the JFlex [43] and Byacc/J [44] tools – the topic of parsing is further described in section 2.3. But the VDMJ interpreter, which instead relies on an internal parser, does not use it. And in turn the VDMJ internal parser is also used for syntax highlighting.

An interesting aspect of Overture is that parts of it were developed using VDM++. From VDM++ specifications, Java classes were generated using the code generators of VDMTools. The UML generators of Overture, for example, were developed using this method.

Development plans for Overture include the restructuring of the Overture AST [45]; generation of C++ and Java code; and GUI generators. Work on links to JML [46, 47] and Alloy [48] is in progress [40].

2.4 Automatic Code Generation

The subject of automatic code generation is extensively studied in the context of compilers. These programs are capable of transforming a program description in a programming language, into executable code for a target platform, performing this translation efficiently. Compilers are also capable of detecting lexical, syntactic and even semantic errors. Furthermore, they can deduce information from source code.

As it stands, an automatic generator of graphical user interfaces from a VDM++ specification can be described as a compiler. The specification assuming the role of source code, and the graphical user interface the role of produced executable code.

2.4.1 Compiling Stages

The compiler process, that is, the process of translating program source code into executable code in a given target architecture, can be split into four different stages: lexical analysis, syntactic analysis, semantic analysis and code generation. The first three stages of the compiling process gather the necessary information for code generation. These stages provide the necessary understanding of the input data, and are responsible for the creation of an intermediate representation of the program.

An additional stage is also common, but not explicitly required for the compiling process: code optimization. This stage takes place before code generation, and serves to produce better quality code. "Better" can mean faster, or for example more compact or even that consumes less of a specific resource. This optimization is done by deep program analysis and transformation.

As a side note, the processes of a compiler are also partially present in interpreters. The difference lies in the final stage, in which the results of the program execution are presented instead of executable code [49].

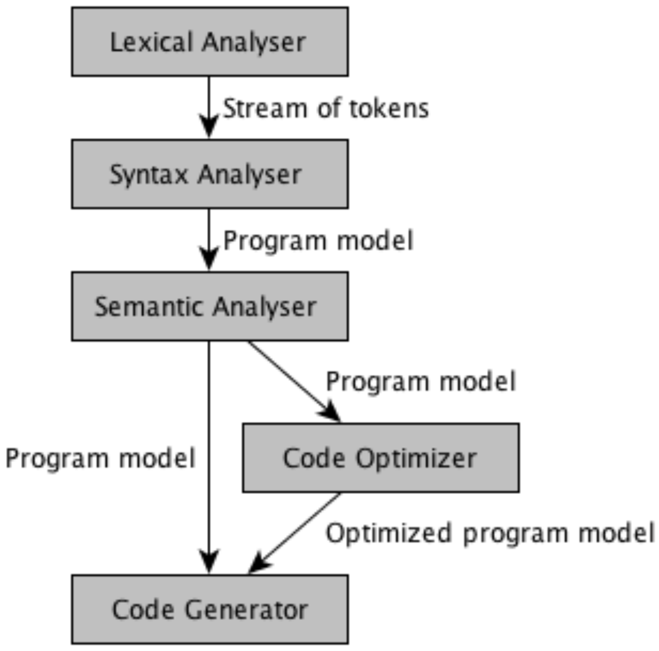


Figure 4: Compiling process diagram.

2.4.1.1 Lexical Analysis

Lexical analysis (also called scanning) is the first stage of the input data understanding process. During this stage the stream of input characters is received and transformed into strings, with an assigned syntactical class. That is, during this stage, the input symbols are grouped to form words, and a set of rules is applied in order to ascertain if the word is valid according to the language of the source code. If the word is valid, a syntactical class is attributed to it.

Most of the necessary work to create a lexical analyser (scanner) is done automatically. This problem – specifying and recognizing patterns in strings – has a mathematical formulation in the form of regular expressions. In fact, "this is a classic example of the application of theoretical results to solve an important practical problem" [49]. Regular expressions can be directly transformed into recognizers, finite automata, with the ability of finding specific patterns in a string of characters.

Tools like Jflex [43] and JavaCC [50] allow the efficient construction of lexical analysers from a specification, taking advantage of the mathematical theory linking regular expressions to finite automata.

2.4.1.2 Syntactic Analysis

The process of syntactic analysis (also called parsing) is responsible for syntax recognition. That is, checking if the program follows the syntax of the language of the source code. This stage works with an abstract version of the program, in the form of a string of tokens produced by the lexical analysis. If the string of tokens forms a valid program, a concrete model of the program is built. And is later used in the compiling process.

This stage has common characteristics with the lexical analysis. Specifically, syntactic analysis is another area in which the study of mathematical formulation lead to the creation of efficient syntactic analysers (parsers). And, as with scanners, there is a large collection of tools that ease the construction of this kind of analysers, performing most of the necessary work automatically [49]. Examples of such tools include Byacc/J [44] and JavaCC [50] – JavaCC generates a scanner as well as a parser.

2.4.1.3 Semantic Analysis

The goal of the compile process is to translate the source code of a program into a form that can be executed in a target platform. This translation is only possible after gathering a large amount of information regarding the program itself. "It must know how values are represented and how they flow between variables. It must understand the structures of the computation. It must analyse how the program interacts with external files and devices" [49]. All this information can be inferred from source code analysis. However, this requires a deeper analysis than scanning or parsing. That is, in this stage of the compiling process the meaning of the information contained in the program is analysed. This is also the last stage of input data understanding of the process.

2.4.1.4 Code Generation

This is the last stage of the compile process, and the most complex. During this stage the compiler, usually by using an intermediate representation created by previous stages, generates corresponding code targeted to the desired platform. Selects operations of the target platform in order to implement each existent operation in the intermediate representation. And also selects the order of the operations in a way that maximizes execution efficiency.

Currently, this is one of the main area of focus of the compiling process [49].

2.4.2 Internal Representation

As introduced in previous sections, compilers are organized into a series of distinct stages. Thus arises the need for a internal representation – an intermediate representation or **IR** [49]– of the code being analyzed and translated. IR is consumed and/or produced along the compiler process. And many compilers use more than one IR during the compilation.

This representation has to be sufficiently expressive to store all the useful and/or necessary facts that might have been collected between compiling stages. Including facts that have no representation in the source code, but are instead inferred from it.

Choosing an appropriate IR for a compiler requires knowledge concerning the source code language or languages, and of the target platform. The kind of information that is necessary to store, analyse and use [49]. Thus a compiler for an object oriented programming language might need to store information about class hierarchy while an equivalent construct is not needed in a functional programming language.

For example, it might be more convenient for a compiler that wishes to translate source code to a very similar programming language to use an IR more close to the source. While for a compiler that wishes to translate the source code into Java bytecode, might want to use a representation that more closely resembles the instruction set of the target.

Another aspect to focus on when choosing and/or designing an IR are practical considerations. The IR should provide simple and efficient ways of performing the operations required by the compiler, or at least the more frequent operations.

An IR can take, for example, the form of: a hashed symbol table [51], a table to store names – variables, defined constants, procedures, etc. – providing efficient methods to look them up; a abstract syntax tree (AST) [52], a tree representation of the structure of the source code, but not every existing detail, with the nodes representing constructs present in the source code; a directed acyclic graph, a compact version of the AST, where identical subtrees are reused.

A practical example of a elaborate IR can be found in the LLVM compiler framework [53].

2.5 Conclusions

The tools or techniques described in the graphical user interface development, focused on a specific aspect or problem within GUI development. For this work, the main problems are user interface design, defining the look and feel; assigning functionality to the interface, and automatic GUI generation. As the basis for the GUI generation process is a formal model, this research can be considered an addition to the field of “Automatic Model-Based Technique” of GUI development, with the distinguishing features of not relying on a special purpose modelling language, and the removal of unpredictability. As described, VDM++ does not focus on interface development. In fact, the language has no GUI development oriented features – it can be used, and is used, to model any kind of computational system. But this does not exclude the use of other techniques, such as user interface markup languages for GUI description.

An important aspect to retain from the field of automatic code generation is the need of an appropriate parser for the VDM++ language. The complexity of the language means that, for example, simple line-by-line reading is not adequate to extract information about classes, functions, operations, etc. This also means that the use of an appropriate representation of VDM++ specifications, an IR, is convenient.

As for VDM++ tool support, there are currently two major tools. Both provide methods to delegate interpreter control to external applications. But due to the closed nature of VDMTools, adapting its functionalities to non previously intended purposes is inherently difficult.

Finally, current user interface development techniques and tools concern themselves mostly on assisting the developer. But not on removing the user from the GUI development process.

Chapter 3

VDM++ GUI Builder

This chapter contains a description of the problem statement, the approach and its implementation.

3.1 Problem Statement

Using more common GUI development techniques (not considering possible software engineering practices) interface construction has the following order of events:

- The developer chooses a graphical toolkit attending to the application requirements, such as platform dependencies (if the application is to operate in the Windows environment, platform specific toolkits such as Cocoa are not an option) and/or personal preferences.
- Having all the tools necessary for GUI development laid out, the developer analyses the system, in order to define the user interface.
- Possessing enough knowledge about "what the system does", the developer using personal experience and/or analysing GUIs for similar systems, comes up with an appropriate design. That is, the number of windows, and their respective interface elements. The choice of interface elements is not as simple as it might seem. Consider a system that keeps a list of lists, its graphical representation is not straightforward. Another important aspect the developer has to consider is layout. For example, it might be desirable to group together similar interface elements, such as buttons.
- The actual construction of the interface can be through traditional programming, the use of an interactive GUI builder tool, the use of a graphical user interface markup language, or a mixture of all three.

- With the user interface created comes the task of adding the required functionality. This includes window management (window calls), assigning action to buttons, and general action to user interface events.

Given that all these tasks involve direct actions of the developer, the objective is to find automatic equivalents. This means:

- The graphical toolkit must not be a limiting factor in terms of platform support.
- Automatic analysis of the VDM++ specification, in order to extract the required information for GUI definition.
- From the required information, automatically infer an appropriate interface. "Appropriate" meaning an interface that is capable of interacting with the executable constructs of the underlying VDM++ specification.
- Write the inferred interface so that it can be rendered.
- Automatically add functionality to the rendered GUI.

3.2 Approach

The approach described in this section (also used for the construction of the GUI generation tool) was developed focusing on one principle, the generated interface should be generic enough to work on any kind of system modelled in VDM++. As a VDM++ specification can model virtually any kind of system this is a very relevant concern.

3.2.1 Architecture

In architectural terms, the approach (whenever possible and convenient) takes advantage of existing tools. This includes VDM++ tools as well as interface development tools.

The GUI generator is integrated with the tools developed in the context of the OvertureTool Project – an open source project to develop a set of high quality formal modelling tools, built on top of the Eclipse Platform. As such, the VDMJ engine is used to execute and evaluate VDM instructions, as well as providing the bulk of the information about the VDM++ specification necessary by the GUI generator.

The other major external tool (not part of Overture) used is the SwiXML Engine. This engine is used to render the GUI elements from a XML description generated by the GUI builder. This tool was chosen because it is specifically designed for Java applications and possesses a very simple (and fairly suited for this approach) mechanism for UI element search. The tool optionally assigns an ID for each UI element. This information is stored in a table that can be easily used on runtime to lookup user interface elements. Another benefit of its use is offloading GUI rendering concerns to an established external tool.

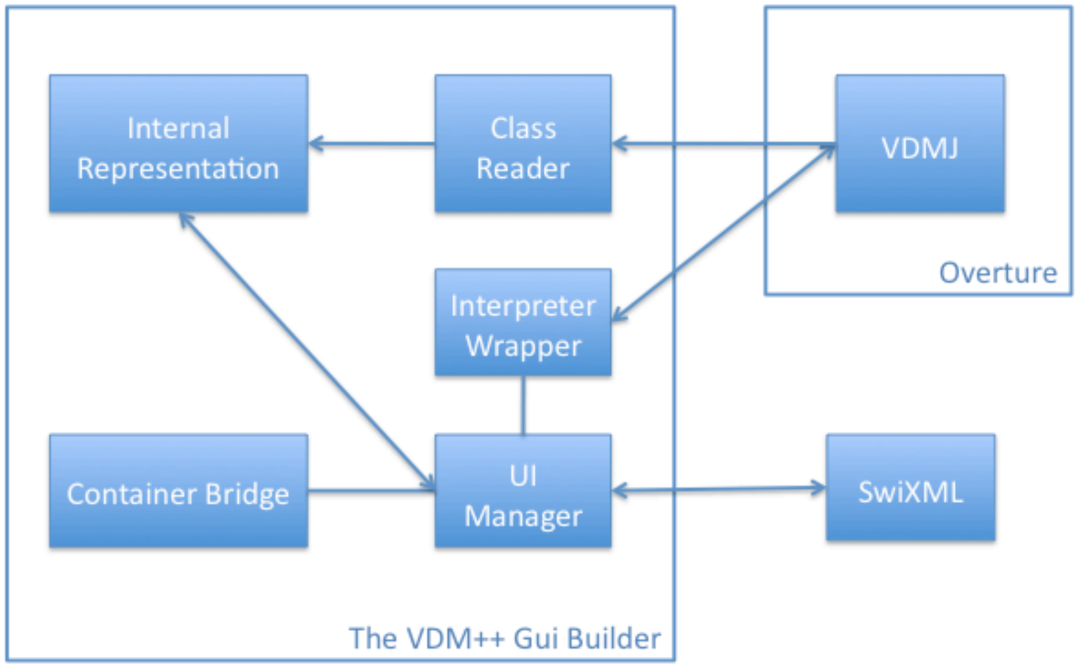


Figure 5: High level architectural diagram

As shown in figure 5, the architecture has five major modules:

- The interpreter wrapper.
- The class reader.
- The UI manager.
- The container bridge.
- The internal representation.

The Interpreter Wrapper serves to establish a link between the external VDMJ engine and the VDM++ GUI Builder. It allows calling VDM++ specification methods and retrieving the result.

The Class Reader is used to collect/maintain an internal representation of the information about the VDM++ classes inside the specification, for instance, their operations, functions, constructors and other elements, tailored for the purposes of GUI generation. As stated, this module relies heavily on VDMJ to extract such information. However, due to the architectural design it can be replaced, without requiring an overhaul of the remaining modules, by a new module that does not rely on an external tool, or one that relies on a more suited tool of the OvertureTool project (for example, the currently in development common AST).

The UI Manager is used to create the windows of the GUI, and serves as an intermediary to the functionality of the underlying VDM specification during runtime.

The Container Bridge, serves as a backend to a window. Basically providing actions during runtime to the events of the user interface and a wrapper for a generated window.

Finally, the Internal Representation is an internal depiction of the VDM++ specification from which the GUI will be generated.

3.2.2 Annotations

In order to provide extra information not extractable from a pure VDM++ specification, some annotations were defined. These annotations are written within VDM++ comments (starting with "--") so that an extension to the VDM++ grammar is not required. The annotations take the form of "--@name=value" or "--@name".

The annotations are intended for VDM++ classes, operations and functions. There are two specific annotations for methods (operations or functions), "--@press" and "--@check=<value>" and one for classes "--@nowindow". The press annotation is intended to identify methods that describe possible user action, and "--@check=<value>" is used to retrieve information – <value> is used to name the state variable with the required information. This annotation can only be applied to methods without arguments. As for the "--@nowindow" class annotation, it serves to mark classes that are to be ignored by the GUI generation process. These would be auxiliary classes in the specification that are not converted to windows on the generated GUI.

3.2.3 GUI Generation Strategy

As previously stated, a VDM++ specification is the basis for the GUI generation process. As this formal specification can be used to describe almost any kind of system, and lacks any intentional GUI oriented elements, the generation strategy relies primarily on signature analysis of methods to create the GUI elements.

The GUI generation strategy supports two different generation modes. One ignoring annotations and another one using annotations to guide the GUI generation process.

The strategy assumes that each class is a valid basis for a single window. In a specification with n classes (not annotated with "--@nowindow"), the resulting GUI will have n+2 windows – two additional windows, one with n buttons to give access to the other windows, and another to show all the class instances created in each moment of the execution, for debugging purposes. An exception to this rule is when there is only one class. In this case only the additional instance list instance is generated.

Not relying on annotations, a method will lead to the generation of input data GUI elements for the arguments, a button with the name of the method, and in cases where there is a return value, an output data GUI element. In cases where the parameter is a class instance, the generated UI provides a combo box with the class instances created until that moment.

Relying on extra information provided by annotations, the generation process adopts a different approach. When a method is annotated with "--@press" the generation strategy will be the same as the one previously described. The annotation serves only to explicitly define that the method is to be parsed in the context of GUI generation. If the method is annotated with "--@check=<value>", two labels will be generated. The first label will show the string defined by <value>, the second will have the return value of the corresponding method.

All windows generated from VDM++ specification classes have a drop-down list. This list contains all the instances of the underlying class. Such list also contains a "new" option to allow the construction of new instances. The option leads to the immediate creation of a new instance of the class when it does not have a constructor, or to a new window when there is a constructor with arguments. This new window serving to collect the required input information for the class constructor.

3.2.4 Dependency Graph

There may exist GUI elements disabled at a given time. For example, when a method has a parameter of the type Class X and there is no instance of such class, this method is disabled. In order to address this issue, the approach keeps track of the dependencies of a given method and checks if they are satisfied.

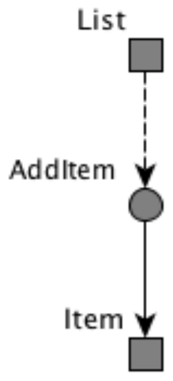


Figure 6: Graph representing the dependencies of simple list system.

The above graph (figure 6) represents the dependencies obtained from a specification of a list. The specification has two classes, "Item" and "List", the latter possessing one operation, AddItem" (represented by a dashed arrow in figure 6). This operation requires the existence of an "Item" instance to be enabled (dependency represented by a solid arrow in 6). Extending the previous example, so that a "List" requires a "Person", would generate the dependency graph in figure 7 which means that it will be possible to construct List instances only after creating Person instances.

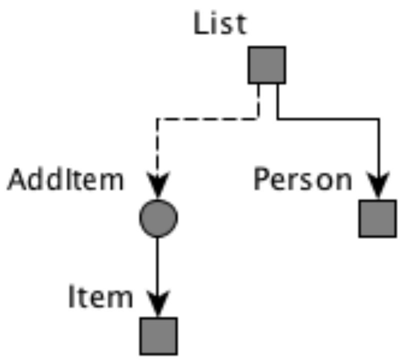


Figure 7: Graph representing the dependencies of the extended list

3.3 Implementation

3.3.1 Class Diagram

Figure 8 represents the UML class diagram of the chosen implementation. The diagram also represents the role of the classes within the high level architecture introduced in section 3.2. The boundary with the name “SwiXML Schema”, although in the diagram containing only one class, represents the XML writer classes. These classes were automatically generated from the SwiXML schema using JAXB.

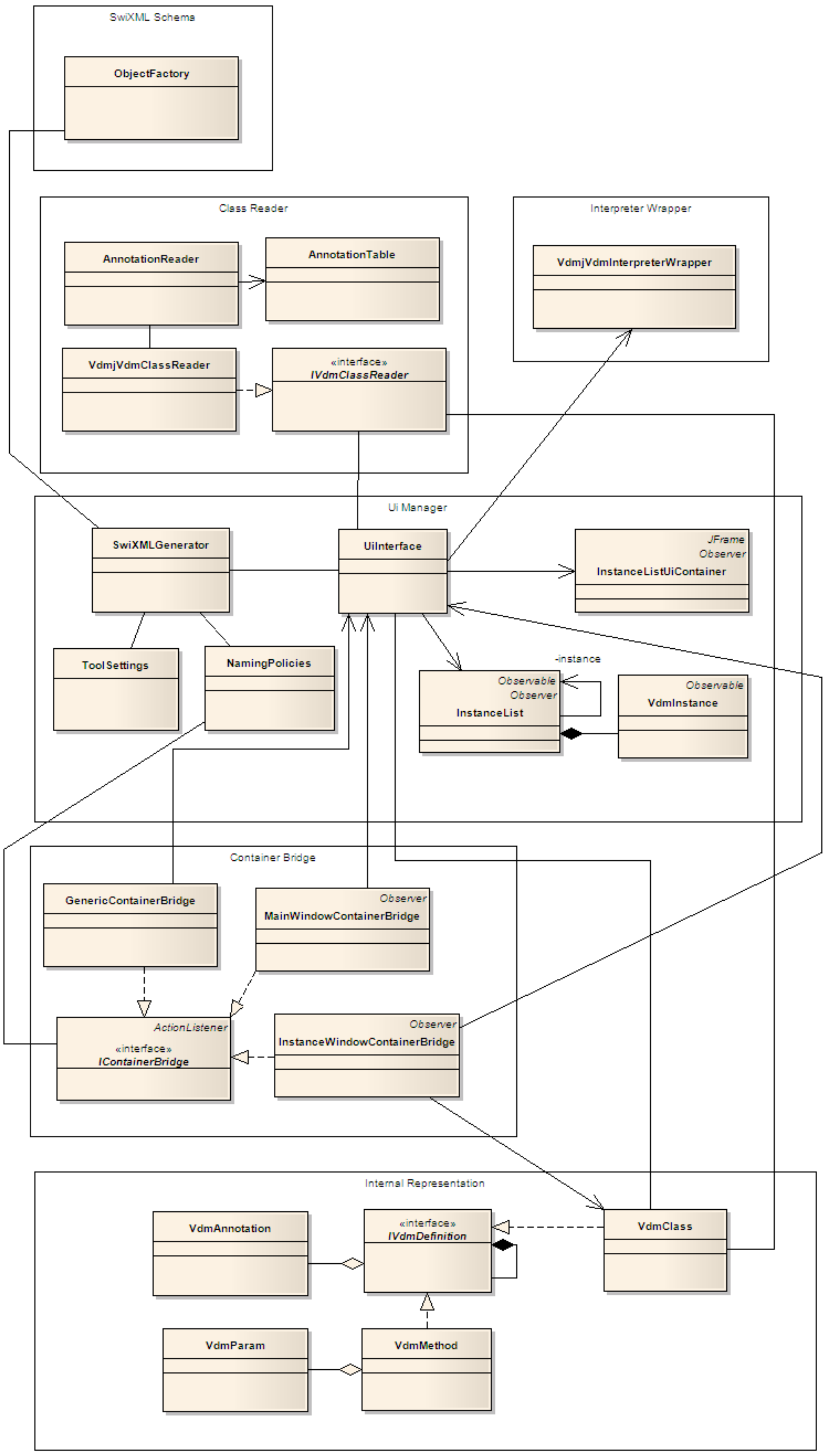


Figure 8: UML class diagram.

3.3.2 *Class Reader and Internal Representation*

To create the internal representation, the implemented class reader extracts information from VDMJ classes and an internal annotation reader – the internal parser of VDMJ does not keep information about comments. The Class Reader is completely reliant on VDMJ for class, operation, function, and type information. In fact, discarding the annotations, it closely functions as an adapter for the VDMJ internal parser.

The internal representation of the VDM++ specification follows a hybrid tree and list structure. The classes (instances of VdmClass) are stored in a list, each class in turn holds internal definitions (children), in this case its methods, operations and/or functions, definitions (instances of VdmMethod).

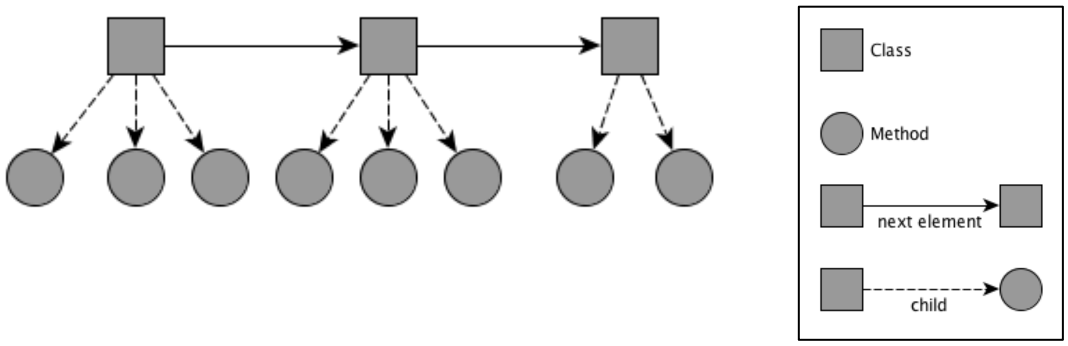


Figure 9: Structural diagram of the internal representation.

Methods store information about their return type, if whether they are a class constructor, and a list of their parameters – parameters are defined in terms of their given name and type. Classes store information about the existence of a class constructor. Both classes and methods hold a list of their annotations, if any.

3.3.3 *GUI Description Generation and Naming Policies*

When generating a new interface, the GUI description is written to SwiXML files, which are later used to render the interface. To write these files, the tool takes advantage of the features of XML languages, more specifically, the XML schema. A XML schema is a description of the language, which can be used by tools, like JAXB, to create custom writers. As such the SwiXML schema was used to generate purposely built classes to write the SwiXML files, guaranteeing total adherence to the rules of the markup language.

The generated description also contains information regarding the intended function of the user interfaces elements, in the form of an ID. These IDs follow a strict naming policy, hard coded in the "NamingPolicies" class. This is necessary as the GUI description is decoupled from the program itself. In fact, it is possible to manually write a GUI description, or even to change a previously generated one, that will still be functional, as long as proper

IDs are given to the user interface elements. An example of a generated XML description can be found in Annex C.

The Naming policies contain ID rules for instance list elements, input data elements, windows, buttons, element containers and output data elements.

Note that, as previously stated, this generation process does not include the “Instance List Window”. This window is a component of the tool itself, not of the generated interface.

3.3.4 Assigning Functionality

The user interface description is decoupled from the rest of the application; so just using the “SwingEngine” of SwiXML to render the generated XML descriptions would result in a series of windows lacking any functionality. To endow the user interface with functionality, a container bridge uses the ID map feature of SwiXML. Using this map it is possible to retrieve the user interface elements according to their intended function, through the naming policies.

Having retrieved the user interface elements, the container bridge proceeds to assign them Java event listeners.

3.3.5 Instance List and Instance Window

As explained in more detail section 3.2, the instance list keeps track of the class instances created over the course of user interface interaction. This purpose reflects itself on the implementation details.

The Instance List is implemented as a Singleton – a design pattern, which ensures that the class will only have one instance. As multiple windows will use the instance list it is necessary to guarantee that there is only one global list, and provide a global entry point for it.

To keep track and inform the necessary parts of the application of a change in the list the Observer pattern is used –a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. When the list is changed, all dependent objects are automatically notified. Using this implementation, for example, the combo boxes of a container are automatically updated when a new element is created.

3.3.6 Interpreter Wrapper

The interpreter wrapper is basically implemented as an adapter for the internal classes of VDMJ. Although support for VDM-SL and VDM-RT is not strictly necessary in the scope of this work, the wrapper can also be used for formal specifications in these dialects.

The adapter does not provide all of the functionality of VDMJ, limiting itself to the more relevant functions in the context of this approach. Namely the wrapper supplies methods to parse and check VDM specifications, basically “loading” specifications; execute and interpret

VDM commands; and create global instances of VDM objects or types, for example instantiating a VDM class.

For the sake of simplicity, the use of the wrapper does not require the use of internal VDMJ types or classes– the wrapper relies on Strings to pass information. The internals of VDMJ are completely isolated by the “Interpreter Wrapper”.

Chapter 4

Case Study

4.1 Description

The Overture Project provides several examples of VDM++ specifications (http//overture.svn.sourceforge.net/). One of these examples is the Cash Dispenser system. In order to evaluate the approach, including the use of annotations, the Cash Dispenser system (Annex A) was selected as the subject of this case study.

The modelled system uses cards and tills to dispense money. The till is connected to a central resource that validates the card and the withdrawal operation. Cards are associated to a cardholder and an account. The system is also capable of issuing account statements, black list cards and store information about account transactions. The specification models this system using eight classes: Account; Card; CardHolder; CentralResource; Clock; Letter; Letterbox and Till. The following list contains a description of each class retrieved from the specification.

- Account – "This class models an account. A number of cards held by individual cardholders are associated with an account. An account has a balance and records transactions."
- Card – "This class models physical cards. Each card has a code, a card id and an account id stored on it. The class provides operations to create a card and to read information stored on a card."
- CardHolder – "This class models a cardholder's name and address. This information is used to post an account statement. The class provides standard read/write operations."
- CentralResource – "This class models the central resource. We assume there is only one central resource in the system, though many tills can be connected to this. The central resource holds the accounts, ids of illegal cards, and connections to a clock and a letterbox."

- Clock – “This class models a clock which maintains a date.”
- Letter – “The class models account statements posted to cardholders.”
- Letterbox – “This class stores account statements sent to cardholders.”
- Till – “This class models a till. A till is connected to a central resource and holds a number of retained cards, which have not been returned to a user of the till. It may hold a current card and it has an attribute to say whether the current card and its PIN code have been validated successfully. In this version of the till we assume that the central resource will always become available within a reasonable time limit.”

The specification also includes a “SimpleTest” class used to test the model.

4.2 Application

The following figure depicts the dependencies that the specification has, according to the previously described approach. Note that in figure 10, only classes, operations and functions with dependencies are represented.

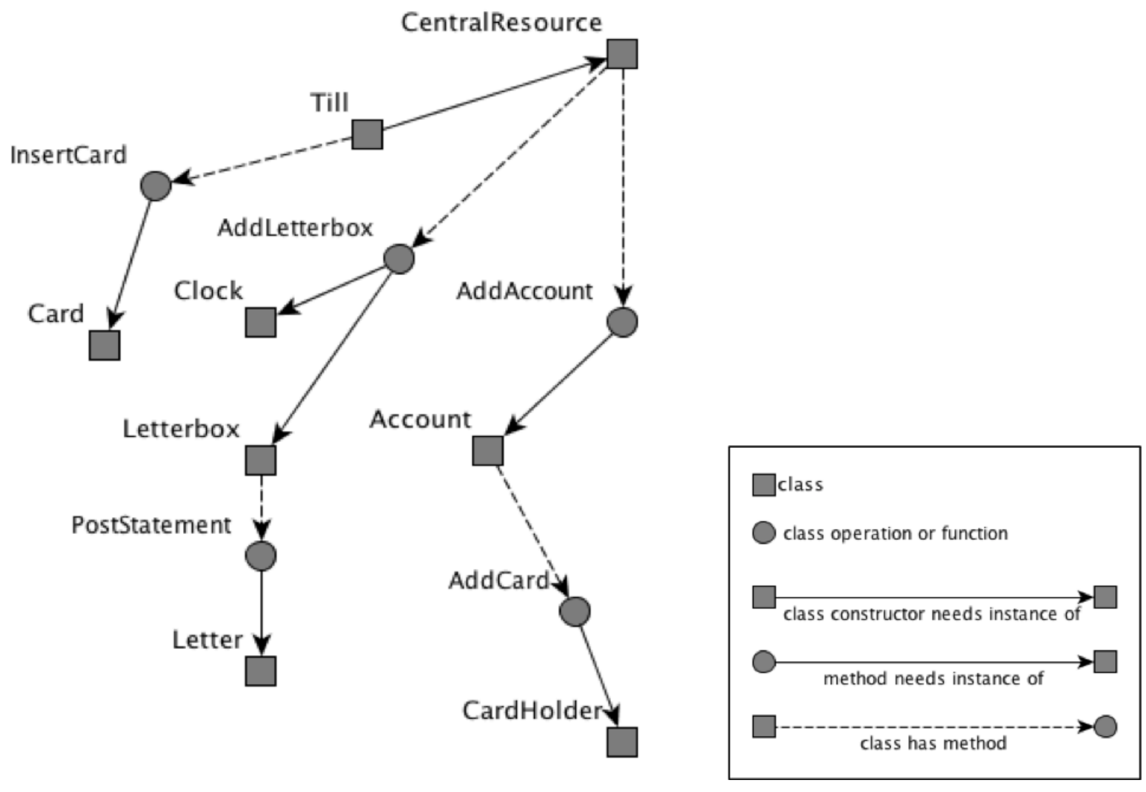


Figure 10: The cash dispenser system dependency graph.

The generated main window is shown in figure 11. The Till button is initially disabled because there is a dependence between Till class and CentralResource class (represented by a solid arrow in figure 10 which means that an instance of CentralResource in needed in order to construct a Till instance.

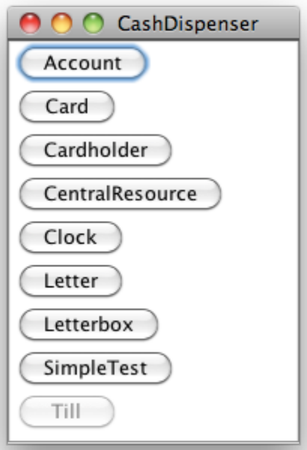


Figure 11: The main window with the Till button disabled

An instance of CentralResource class is immediately constructed when opening the corresponding window (figure 12) because such class has no defined constructor. The window has two buttons disabled, “AddLetterbox” and “AddAccount” – their dependencies are not yet satisfied, as illustrated in figure 10. “AddLetterBox” method is enabled after creating instances of “Clock” and “Letterbox” classes. “AddAccount” method is enabled after constructing instances of the “Account” class.

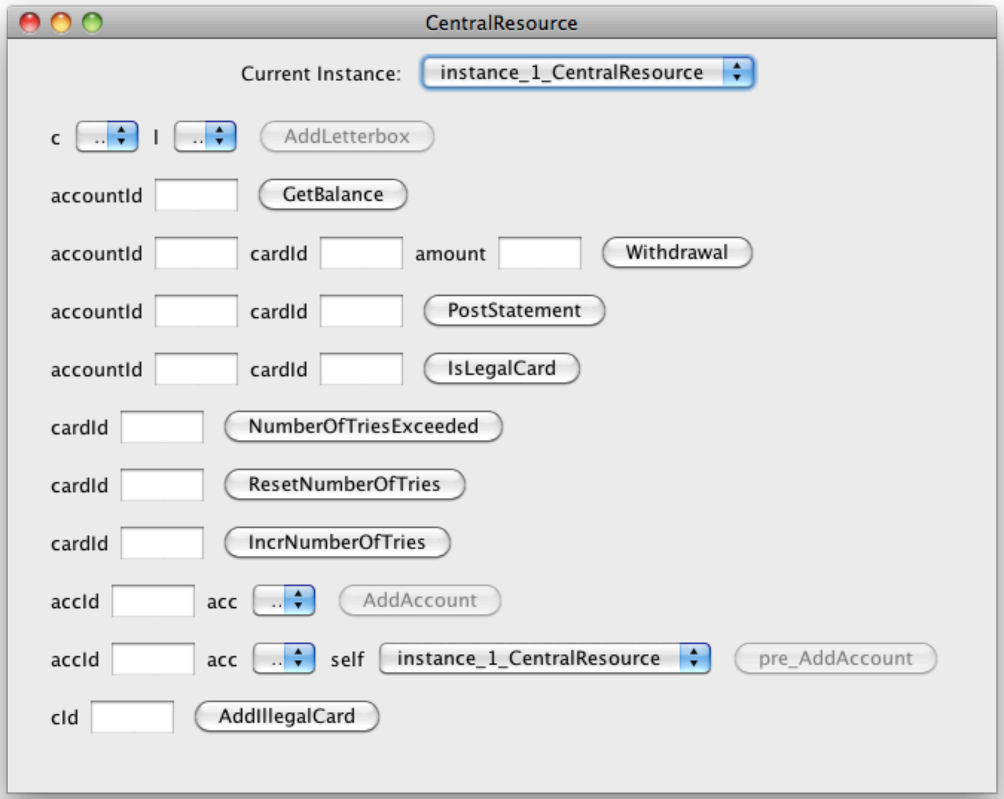
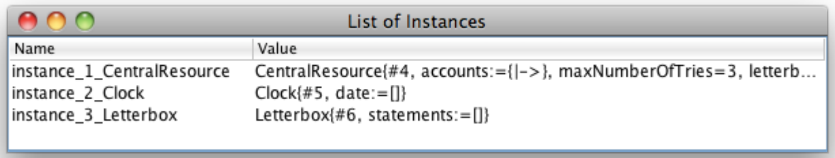

CentralResource
Current Instance: instance_1_CentralResource
c .. l .. AddLetterbox
accountId GetBalance
accountId cardId amount Withdrawal
accountId cardId PostStatement
accountId cardId IsLegalCard
cardId NumberOfTriesExceeded
cardId ResetNumberOfTries
cardId IncrNumberOfTries
accId acc .. AddAccount
accId acc .. self instance_1_CentralResource pre_AddAccount
cId AddIllegalCard

Figure 12: The “CentralResource” window

After creating the “Clock” and the “Letterbox”, the “AddLetterbox” operation becomes enabled, with the appropriate controls now populated with the constructed instances of “Clock” and “Letterbox” classes.


List of Instances

Name	Value
instance_1_CentralResource	CentralResource{#4, accounts:={|->}, maxNumberOfTries=3, letterb...
instance_2_Clock	Clock{#5, date:=[]}
instance_3_Letterbox	Letterbox{#6, statements:=[]}

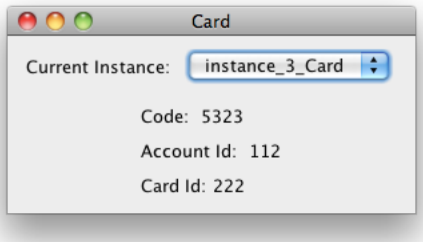
Figure 13: The instance window list, after creating the instances.

4.2.1 *Applying User Stories*

To further evaluate the generation approach and the tool, two user stories were considered:

- The developer wishes to check that the cards are being correctly modelled;
- The developer wishes to check that the modelled system is capable of showing the user an account balance.

For the first user story, the operations of the Card class were annotated with "--@check=<value>", and the remaining classes were annotated with "--nowindow". The user interface was generated in annotation only mode. The resulting interface (not including the Instance List Window) has a single window, "Card" (figures 14 and 15).


Card

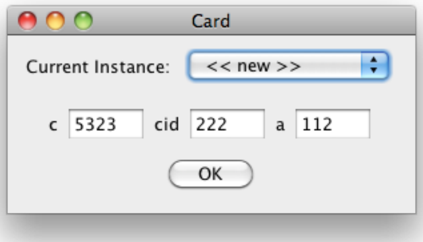
Current Instance: instance_3_Card

Code: 5323

Account Id: 112

Card Id: 222

Figure 14: The card window.


Card

Current Instance: << new >>

c 5323 cid 222 a 112

Figure 15: The card window in constructor mode.

For the second user story, the user interface was generated using no annotations – the user interface can interact with every executable construct of the underlying VDM++ specification. The resulting interface possesses a main window like the one depicted in figure 11. The "SimpleTest" window, for example, is unnecessary, and would justify the use of the "--nowindow" annotation. But this would require annotating the rest of the specification for the annotation generation mode. This is time consuming, and it is doubtful that a developer would choose to do this. To perform the cash withdrawal using the graphical interface, there is no predefined order of user events (the dependencies, figure 10, are not sufficient to enforce a single order). Arbitrarily starting by opening the Account window (figure 16) to create a new Account, its interface possesses no constructor panel – the specification defines a “Create” operation to be used as the constructor, that lead to the interface elements circled in figure 16.

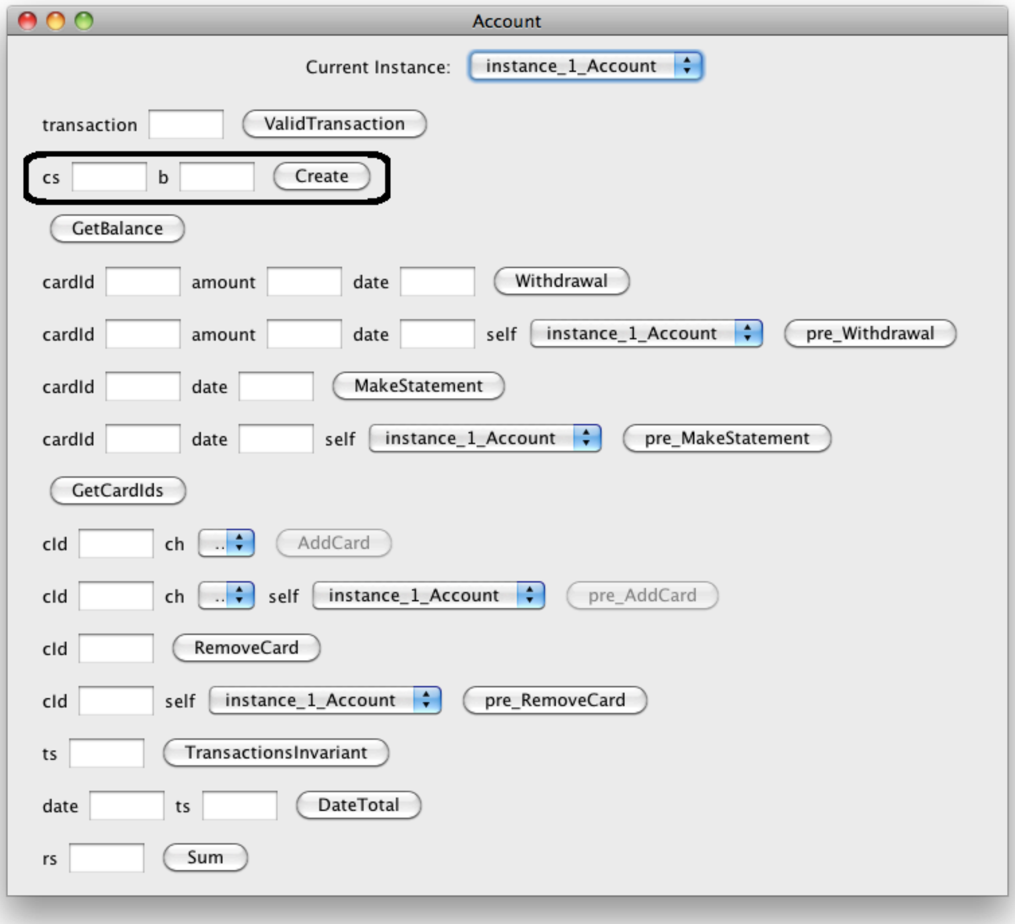

Account
Current Instance: instance_1_Account
transaction ValidTransaction
cs b Create
GetBalance
cardId amount date Withdrawal
cardId amount date self instance_1_Account pre_Withdrawal
cardId date MakeStatement
cardId date self instance_1_Account pre_MakeStatement
GetCardIds
cId ch .. AddCard
cId ch .. self instance_1_Account pre_AddCard
cId RemoveCard
cId self instance_1_Account pre_RemoveCard
ts TransactionsInvariant
date ts DateTotal
rs Sum

Figure 16: The account window, with the ‘constructor’ circled.

Using an empty map, ‘{|->}’, and ‘1000’ as arguments for ‘Create’, the new account became ready to be used. The account still had no associated cards. This was verified by inspecting the list of instances window or by clicking on the “GetCardIds” button. The CardHolder window (figure 17), as with the Account window, displays no constructor panel, only a ‘Create’ button. With a cardholder created, the “AddCard” button in the Account window became enabled, despite no card being present – the cardholder is only a person to whom cards may or may not belong. This is allowed as cards are identified by their id, a simple number. Associating a card with the id “1” to the account yield no error, despite no card being available. This is a property of the underlying VDM++ specification not so much as a defect of the graphical user interface. Cards are sometimes identified only by their id in the modelled system. No verification that the actual card id belongs to a card is made at this point.

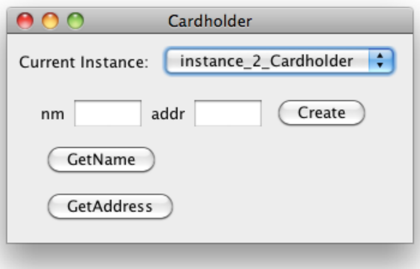


Figure 17: The Card Holder Window

Proceeding, in no particular order, to create a card by opening the corresponding window. The generated 'Card' window (figure 15) possesses a constructor panel, which is used to create a card with the pin "1", id "1" and account id "1". A Till and a Central Resource are still required to continue. As there is no Central Resource the Till window is still disabled. To create a Central Resource it is only necessary to open the corresponding window (figure 12) – the Central Resource class has no constructor. An easily overlooked and necessary step is to associate the previously created account to the central resource, using the AddAccount button. Finally with a card, a card holder, an account and a central resource created, it becomes possible to use a till to dispense money. Opening the Till window (figure 18) the interface requests that the user chooses a CentralResource to which connect the Till.

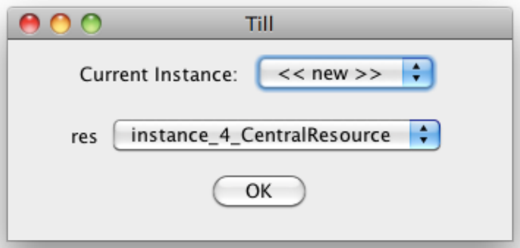


Figure 18: The Till window.

With a CentralResource selected, all operations and functions of a modelled till are accessible through the Till window (figure 19). To check the balance, it is necessary to insert the card, insert the pin and validate it, and finally request a balance check (figure 19). The window does not enforce the order of operations. That is, the "Validate" button, for example, is still enabled even when no card is "inserted".

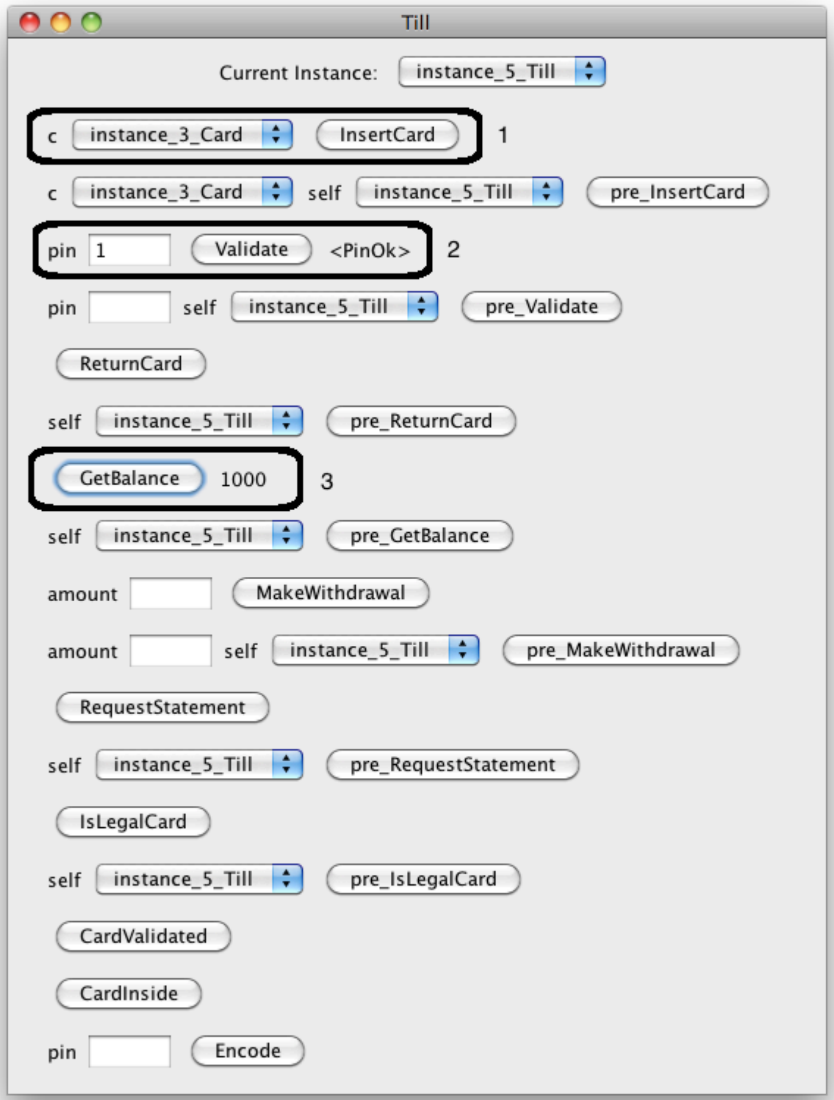

Till
Current Instance: instance_5_Till
c instance_3_Card InsertCard
1
c instance_3_Card self instance_5_Till pre_InsertCard
pin 1 Validate <PinOk>
2
pin self instance_5_Till pre_Validate
ReturnCard
self instance_5_Till pre_ReturnCard
GetBalance 1000
3
self instance_5_Till pre_GetBalance
amount MakeWithdrawal
amount self instance_5_Till pre_MakeWithdrawal
RequestStatement
self instance_5_Till pre_RequestStatement
IsLegalCard
self instance_5_Till pre_IsLegalCard
CardValidated
CardInside
pin Encode

Figure 19: The Till window 'connected' to a central resource.

4.2.2 Manual modification of the GUI

Using the generated XML descriptor files it is possible to manually modify the generated interface. Figure 20 depicts a new layout for a main window generated using no annotations.

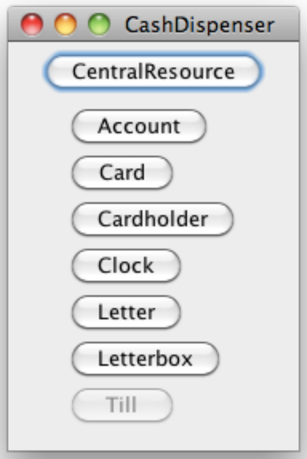


Figure 20: Modified main window.

This is a basic change to the descriptor, which takes only a few seconds to make. The required knowledge to perform this change is very basic – SwiXML layout tags. For a more elaborate change in a descriptor more extensive knowledge of SwiXML is necessary. The modified CentralResource window depicted in figure 21 required approximately 10 minutes to created using a simple text editor. Setting the red background colour, for example, required going through SwiXML examples to understand how it could be done.

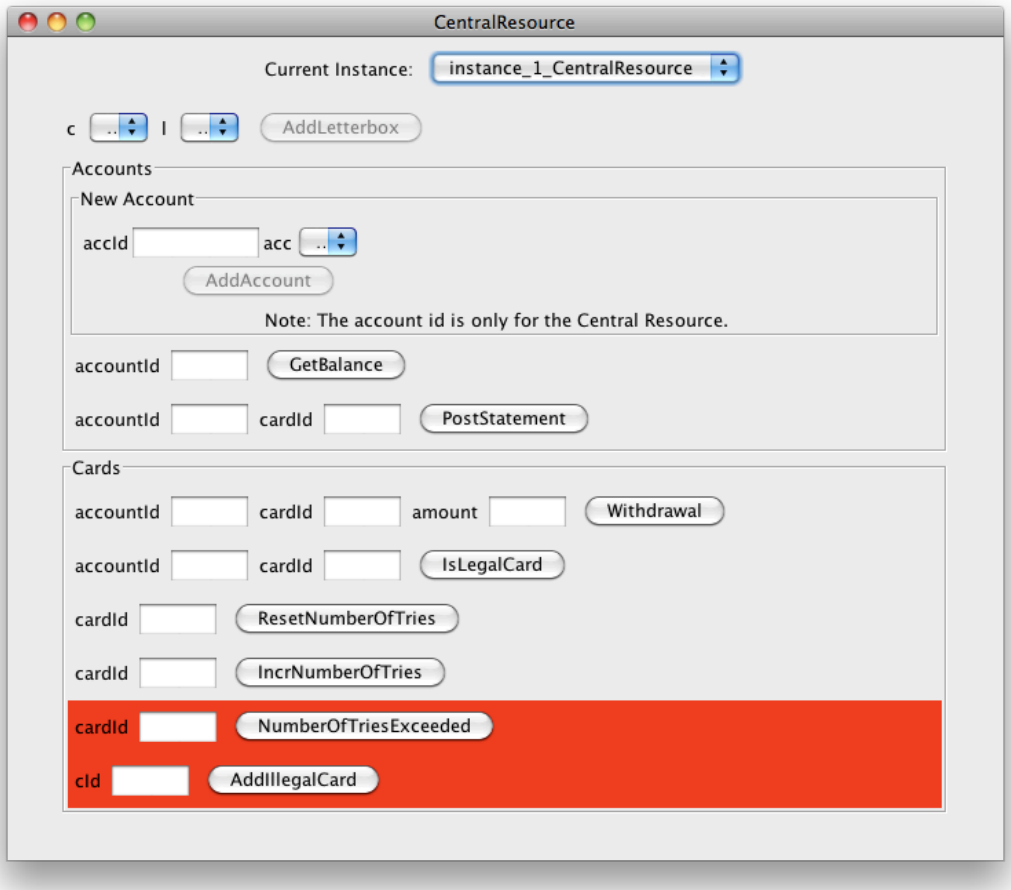

CentralResource
Current Instance: instance_1_CentralResource
c .. l .. AddLetterbox
Accounts
New Account
accId acc ..
AddAccount
Note: The account id is only for the Central Resource.
accountId GetBalance
accountId cardId PostStatement
Cards
accountId cardId amount Withdrawal
accountId cardId IsLegalCard
cardId ResetNumberOfTries
cardId IncrNumberOfTries
cardId NumberOfTriesExceeded
cId AddIllegalCard

Figure 21: Modified Central Resource Window.

4.3 Discussion and Conclusions

As the case study shows, the described approach is able to generate a fully functional GUI to interact with a VDM++ specification, with minimal additional effort from the part of the developer. It enables calling methods present in the specification and displaying the return value. However, the generated GUI is unsophisticated, due to the inherent difficulty of implementing a GUI generation process based on a formal language not specific for GUI modelling (apart from the annotations introduced by the approach). More annotations could be introduced, but they would require additional effort, which could put into question the goal of this research work: generate a GUI from a generic VDM++ specification with minimal effort. The GUI element enabling/disabling previously described can check argument availability but does not validate it. For example, a method that takes as argument a class instance would still be accessible, even if the available instances themselves possessed undefined or invalid required values. But this is not necessarily a limitation of the approach as it could serve to help the developer identify situations where function or operation pre-conditions are missing.

Chapter 5

Conclusions and Future Work

The described approach is able to generate a fully functional GUI from a VDM++ specification. The generated GUI is also capable of enabling/disabling GUI buttons based on the dependencies extracted from the analysis of GUI specification methods. This is achieved while following the grammar of the VDM++ formal language and without requiring the user active participation in the GUI generation process. Furthermore, by adding additional information through the use of annotations to the VDM++ specification, the generated GUI elements become better adjusted to the underlying model.

It is also possible to extract further conclusions from the challenges encountered during the research and development stage.

The subject of this research work can be divided into two main problems, user interface definition and assigning functionality to a user interface. In the course of this work, the two problems were addressed at the same time. That is, the approach to user interface definition was created considering the challenges of functionality assignment. This allowed for a more coordinated approach to user interface generation. But it also meant that the functionality issue conditioned the study of user interface definition. To yield better results in this area of study, the two problems should be treated separately. This may seem counter-intuitive, but it would provide more focus on the specific problems of the two issues.

This work benefited greatly of existing GUI and VDM++ development tools, probably, even to the point of making it possible to achieve the goals of this work. For example, having to directly address the problem of user interface rendering instead of offloading the issue to an external tool, would have diverted time and resources away from the main objectives.
Taking into account the results and features of available VDM++ tools, the approach could be improved in the following ways:

- Adding different user interface patterns to choose from. Based on the design pattern terminology, this approach uses a user interface pattern that focuses on guaranteeing that the GUI will be adequate for a VDM++ specification, whichever it may be. But in terms of an evolving UI prototype, it could be useful to try different interface patterns.

- Taking advantage of the available pre-conditions in a VDM++ specification. The dependencies that check for GUI element enabling/disabling could also be extended to include the evaluation of pre conditions.
- Implementing the connection of the generated GUI with the API code generated automatically by existing VDM tools. VDM Tools are capable of generating Java code from a VDM++ specification. The integration of the GUI with this code would lead to a standalone java GUI application created with no user intervention from a VDM++ specification. This could be achieved by making the UI Manager module aware of the proper VDM methods equivalents in the generated Java code. Thus 'redirecting' the GUI calls to such methods in Java instead of VDM++ methods like what happens now.
- Make the class reader dependent on the Overture AST when the development of this tool is completed. Currently the tool depends directly on VDMJ for extracting class information, but this is not a recommended method. Ideally the tool should use a purposely built Abstract Syntax Tree.
- Using deeper specification analysis, infer which operations or functions can be annotated with "--@check".

References

1. Myers, B., S.E. Hudson, and R. Pausch, *Past, present, and future of user interface software tools*. ACM Trans. Comput.-Hum. Interact., 2000. **7**(1): p. 3-28.
2. Hammersmith, O., *An Introduction to the GIMP Tool Kit*. Linux J., 1998. **1998**(47es): p. 5.
3. Hartness, K., *Graphics and user interfaces in C++ with Qt*. J. Comput. Small Coll., 2005. **20**(4): p. 198-199.
4. Darwin, I., *GUI Development with Java*. Linux J., 1999. **1999**(61es): p. 4.
5. Rogers, M.P., *How sweet it is! a course in cocoa*. J. Comput. Small Coll., 2003. **18**(4): p. 295-307.
6. Corporation, M. *Windows Forms*. 2011 [cited 2011; Available from: http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx.
7. Aitel, D., *A beginner's guide to using pyGTK and Glade*. Linux J., 2003. **2003**(113): p. 5.
8. Oracle. *Lesson: Using the NetBeans GUI Builder*. 2011 6/3/2011]; Available from: http://download.oracle.com/javase/tutorial/javabeans/nb/index.html.
9. Bishop, J., *Multi-platform user interface construction: a challenge for software engineering-in-the-small*, in *Proceedings of the 28th international conference on Software engineering*2006, ACM: Shanghai, China. p. 751-760.
10. Coyette, A., et al., *SketchiXML: towards a multi-agent design tool for sketching user interfaces based on USIXML*, in *Proceedings of the 3rd annual conference on Task models and diagrams*2004, ACM: Prague, Czech Republic. p. 75-82.
11. Lambourg, Q., et al., *USIXML: A Language Supporting Multi-path Development of User Interfaces*, in *Lecture Note in Computer Science*2005. p. 134-135.
12. Corp., M., *Xaml Object Mapping Specification 2006*, 2008.
13. Corp., M., *Xaml Object Mapping Specification 2009*, 2010.
14. Corporation, M. *XAML Overview (WPF)*. 2011 29/06/2011]; Available from: http://msdn.microsoft.com/en-us/en-us/library/ms752059.aspx.

15. Network, M.D. *The Joy of XUL*. 2011 28/05/2011; Available from: https://developer.mozilla.org/en/The_Joy_of_XUL.

16. Paulus, W. *SwiXML - Generate javax.swing at runtime based on XML descriptors*. 2011; Available from: http://www.swixml.org/.

17. Jarvi, J., et al., *Algorithms for user interfaces*, in *Proceedings of the eighth international conference on Generative programming and component engineering*2009, ACM: Denver, Colorado, USA. p. 147-156.

18. Hentenryck, P.V. and V. Saraswat, *Strategic directions in constraint programming*. ACM Comput. Surv., 1996. **28**(4): p. 701-726.

19. Myers, B.A., et al., *The Amulet user interface development environment*, in *CHI '97 extended abstracts on Human factors in computing systems: looking to the future*1997, ACM: Atlanta, Georgia. p. 214-215.

20. Zanden, B.T.V., et al., *Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits*. ACM Trans. Program. Lang. Syst., 2001. **23**(6): p. 776-796.

21. Bowen, J. and S. Reeves, *Using formal models to design user interfaces: a case study*, in *Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI...but not as we know it - Volume 1*2007, British Computer Society: University of Lancaster, United Kingdom. p. 159-166.

22. Szekely, P., P. Luo, and R. Neches, *Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design*, in *Proceedings of the SIGCHI conference on Human factors in computing systems*1992, ACM: Monterey, California, United States. p. 507-515.

23. Cruz, A.M.R.d. and J.P. Faria, *A metamodel-based approach for automatic user interface generation*, in *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*2010, Springer-Verlag: Oslo, Norway. p. 256-270.

24. Nichols, J., D.H. Chau, and B.A. Myers, *Demonstrating the viability of automatically generated user interfaces*, in *Proceedings of the SIGCHI conference on Human factors in computing systems*2007, ACM: San Jose, California, USA. p. 1283-1292.

25. Pawson, R., *Naked objects*, in *Department of Computer Science*2004, Trinity College: Dublin.

26. Clarke, E.M. and J.M. Wing, *Formal Methods: State of the Art And Future Directions*. ACM Comput. Surv., 1996. **28**: p. 626-243.

27. Woodcock, J., et al., *Formal methods: Practice and experience*. ACM Comput. Surv., 2009. **41**(4): p. 1-36.

28. Nami, M.R. and F. Hassani, *A comparative evaluation of the Z, CSP, RSL, and VDM languages*. SIGSOFT Softw. Eng. Notes, 2009. **34**(3): p. 1-4.

29. Larsen, P.G. and J.S. Fitzgerald. *Recent Industrial Applications of Formal Methods in Japan*. in *BCS-FACS Workshop on Formal Methods in Industry*. 2008. British Computer Society.

30. Bjørner, D., *The Vienna development method (VDM): Software specification and program synthesis*, in *Proceedings of the International Conference on Mathematical Studies of Information Processing*1979, Springer-Verlag. p. 326-359.

31. Clemmensen, G.B. and O.N. Oest, *Formal specification and development of an ada compiler - a vdm case study*, in *Proceedings of the 7th international conference on Software engineering*1984, IEEE Press: Orlando, Florida, United States. p. 430-440.

32. Pedersen, J.S. and K.H. Shingler, *Software Development Using VDM*, 1989.

33. Kurita, T. and Y. Nakatsugawa, *The Application of VDM to the Industrial Development of Firmware for a Smart Card IC Chip*. International Journal of Software and Informatics, 2009. **3**(2-3): p. 343-355.

34. Larsen, P.G., et al., *VDM-10 Language Manual*, 2011.

35. *VDMTools: advances in support for formal modeling in VDM*. SIGPLAN Not., 2008. **43**(2): p. 3-11.

36. Larsen, P.G., et al., *Tutorial for Overture/VDM-RT*, in *Overture Technical Report Series*2011.

37. Corporation, C., *VDM Tools User Manual 1.1*, 2010.

38. Corporation, C.S., *The VDM Toolbox API 1.1*, 2008.

39. Wolfe, A., *Eclipse: A Platform Becomes an Open-Source Woodstock*. Queue, 2003. **1**(8): p. 14-16.

40. Larsen, P.G., et al., *The overture initiative integrating tools for VDM*. SIGSOFT Softw. Eng. Notes, 2010. **35**(1): p. 1-6.

41. Battle, N., *VDMJ Tool Support: User Guide*, 2011.

42. Larsen, P.G., et al., *Overture VDM-10 Tool Support: User Guide*, in *Overture Technical Report Series*2011.

43. Klein, G., *JFlex User's Manual*, 2009.

44. Hurka, T. *BYACC/J*. 2008; Available from: http://byaccj.sourceforge.net/.

45. Lausdahl, K. and A. Ribeiro, *Reestructuring of AST in Overture components: Overture-II*, 2011.

46. Leavens, G.T., *Design by Contract with JML*, Y. Cheon, Editor 2006.

47. Leavens, G.T., A.L. Baker, and C. Ruby, *Preliminary design of JML: a behavioral interface specification language for java*. SIGSOFT Softw. Eng. Notes, 2006. **31**(3): p. 1-38.

48. Jackson, D., *Alloy: a lightweight object modelling notation*. ACM Trans. Softw. Eng. Methodol., 2002. **11**(2): p. 256-290.

49. Cooper, K.D. and L. Torczon, *Engineering a compiler*2004: Morgan Kaufmann.

50. Succi, G. and R.W. Wong, *The application of JavaCC to develop a C/C++ preprocessor*. SIGAPP Appl. Comput. Rev., 1999. **7**(3): p. 11-18.

51. Graham, S.L., W.N. Joy, and O. Roubine, *Hashed symbol tables for languages with explicit scope control*, in *Proceedings of the 1979 SIGPLAN symposium on Compiler construction*1979, ACM: Denver, Colorado, United States. p. 50-57.

52. Jones, J., *Abstract Syntax Tree Implementation Idioms*. Pattern Languages of Program Design, 2003.

53. Lattner, C. and V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis \& Transformation*, in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*2004, IEEE Computer Society: Palo Alto, California. p. 75.

Annex A

A VDM++ Specification of a Cash Dispenser System

```
class SimpleTest

values

  c1 : Card = new Card(123456,1,1);
  cards : set of Card = {c1};
  resource : CentralResource = new CentralResource();
  tills : map TillId to Till = {1 |-> new Till(resource)};

instance variables

  clock : Clock := new Clock();
  letterbox : Letterbox := new Letterbox();

types

  public TillId = nat;

operations

public Run : () ==> bool
  Run () ==
    (clock.SetDate("150999");
    let peter = new Cardholder().Create("Peter Gorm Larsen",
"Granvej 24")
    in
       let pglacc1 = new Account().Create({1 |-> peter},5000),
           pglid1 = 1
       in
         (resource.AddAccount(pglid1,pglacc1);
          resource.AddLetterbox(clock, new Letterbox());
          tills(1).InsertCard(c1);
          if tills(1).Validate(123456) = <PinOk>
          then return tills(1).MakeWithdrawal(800);
         );
    );

end SimpleTest

class Account
```

```
instance variables
  cards : map Card`CardId to Cardholder;
  balance : nat;
  transactions : seq of Transaction := [];

  inv TransactionsInvariant(transactions);

values
  dailyLimit : nat = 2000;

types
  public AccountId = nat;
  public Transaction :: date : Clock`Date
                  cardId : Card`CardId
                  amount : nat;

operations
  ValidTransaction : Transaction ==> bool
  ValidTransaction(transaction) ==
    is not yet specified;

public Create : map Card`CardId to Cardholder * nat ==> Account
  Create(cs,b) ==
    (cards := cs;
     balance := b;
     return self);

  public GetBalance : () ==> nat
  GetBalance() ==
    return balance;

public Withdrawal : Card`CardId * nat * Clock`Date ==> bool
  Withdrawal(cardId,amount,date) ==
    let transaction = mk_Transaction(date,cardId,amount)
    in
      if balance - amount >= 0 and
         DateTotal(date,transactions^[transaction]) <= dailyLimit
      then
       (balance := balance - amount;
        transactions := transactions ^ [transaction];
        return true)
      else
        return false
  pre cardId in set dom cards;

  public MakeStatement : Card`CardId * Clock`Date ==> Letter
  MakeStatement(cardId,date) ==
    let nm = cards(cardId).GetName(),
        addr = cards(cardId).GetAddress()
    in
      (dcl letter : Letter := new Letter();
       letter.Create(nm,addr,date,transactions,balance))
  pre cardId in set dom cards;

public GetCardIds: () ==> set of Card`CardId
  GetCardIds() ==
    return dom cards;

public AddCard : Card`CardId * Cardholder ==> ()
```

```
  AddCard(cId,ch) ==
    cards := cards munion {cId |-> ch}
  pre cId not in set dom cards;

  public RemoveCard : Card`CardId ==> ()
  RemoveCard(cId) ==
    cards := {cId} <-: cards
  pre cId in set dom cards;

functions
  TransactionsInvariant: seq of Transaction +> bool
  TransactionsInvariant(ts) ==
    forall date in set {ts(i).date | i in set inds ts} &
      DateTotal(date,ts) <= dailyLimit;

DateTotal : Clock`Date * seq of Transaction +> nat
  DateTotal(date,ts) ==
    Sum([ts(i).amount | i in set inds ts & ts(i).date = date]);

  Sum: seq of real +> real
  Sum(rs) ==
    if rs = [] then 0
    else
      hd rs + Sum(tl rs);

end Account
```

class Card

```
types
  public CardId = nat;
  public Code = nat;
  public PinCode = nat;

instance variables
  code : Code;
  cardId : CardId;
  accountId : Account`AccountId;

operations
  public Card : Code * CardId * Account`AccountId ==> Card
  Card(c,cid,a) ==
    (code := c;
     cardId := cid;
     accountId := a);

  public GetCode : () ==> Code
  GetCode() ==
    return code;

  public GetAccountId : () ==> Account`AccountId
  GetAccountId() ==
    return accountId;

  public GetCardId : () ==> CardId
  GetCardId() ==
    return cardId;

end Card
```

class Cardholder

```
types
  public Address = seq of char;
  public Name = seq of char;

instance variables
  name : Name;
  address : Address;

operations
  public Create : Name * Address ==> Cardholder
  Create(nm,addr) ==
    (name := nm;
     address := addr;
     return self);

  public GetName : () ==> Name
  GetName () ==
    return name;

  public GetAddress : () ==> Address
  GetAddress() ==
    return address;

end Cardholder
```

class CentralResource

```
instance variables
  accounts      : map Account`AccountId to Account := {|->};
  numberOfTries : map Card`CardId to nat := {|->};
  illegalCards  : set of Card`CardId := {};
inv dom numberOfTries union illegalCards subset
    dunion {acc.GetCardIds() | acc in set rng accounts};

  letterbox     : Letterbox;
  clock         : Clock;

inv forall acc1,acc2 in set rng accounts &
          acc1 <> acc2 =>
          acc1.GetCardIds() inter acc2.GetCardIds() = {};

values
  maxNumberOfTries : nat = 3;

operations
  public AddLetterbox : Clock * Letterbox ==> ()
  AddLetterbox(c,l) ==
    (clock := c;
     letterbox := l);

  public GetBalance : Account`AccountId ==> [nat]
  GetBalance(accountId) ==
    if accountId in set dom accounts then
      accounts(accountId).GetBalance()
    else
      return nil;
  public Withdrawal : Account`AccountId * Card`CardId * nat ==>
bool
  Withdrawal(accountId,cardId,amount) ==
    if IsLegalCard(accountId,cardId) then

accounts(accountId).Withdrawal(cardId,amount,clock.GetDate())
    else
      return false;

  public PostStatement : Account`AccountId * Card`CardId ==> bool
  PostStatement(accountId,cardId) ==
    if IsLegalCard(accountId,cardId) then
      (letterbox.PostStatement

(accounts(accountId).MakeStatement(cardId,clock.GetDate()));
       return true)
    else
      return false;

  public IsLegalCard : Account`AccountId * Card`CardId ==> bool
  IsLegalCard(accountId,cardId) ==
    return
      cardId not in set illegalCards and
      accountId in set dom accounts and
      cardId in set accounts(accountId).GetCardIds();

  public NumberOfTriesExceeded : Card`CardId ==> bool
  NumberOfTriesExceeded(cardId) ==
    return numberOfTries(cardId) >= maxNumberOfTries;

  public ResetNumberOfTries : Card`CardId ==> ()
  ResetNumberOfTries(cardId) ==
    numberOfTries(cardId) := 0;

  public IncrNumberOfTries : Card`CardId ==> ()
  IncrNumberOfTries(cardId) ==
    numberOfTries(cardId) := numberOfTries(cardId) + 1;

public AddAccount : Account`AccountId * Account ==> ()
  AddAccount(accId,acc) ==
    atomic
    (accounts := accounts ++ {accId |-> acc};
     numberOfTries := numberOfTries ++
                      {cId |-> 0 | cId in set acc.GetCardIds()};
     )
  pre accId not in set dom accounts;

  public AddIllegalCard : Card`CardId ==> ()
  AddIllegalCard(cId) ==
    illegalCards := illegalCards union {cId};

end CentralResource
```

class Clock

```
types
  public Date = seq of char;

instance variables
```

```
  date : Date := "";

operations
  public SetDate : Date ==> ()
  SetDate(d) ==
    date := d;

  public GetDate : () ==> Date
  GetDate() ==
    return date;

end Clock
```

class Letter

```
instance variables
  public name : Cardholder`Name;
  public address : Cardholder`Address;
  public date : Clock`Date;
  public transactions : seq of Account`Transaction;
  public balance : nat

operations
  public Create: Cardholder`Name * Cardholder`Address *
Clock`Date *
          seq of Account`Transaction * nat ==> Letter
  Create(nm,addr,d,ts,b) ==
    (name := nm;
     address := addr;
     date := d;
     transactions := ts;
     balance:= b;
     return self);
```

class Letterbox

```
instance variables
  statements : seq of Letter := [];

operations
  public PostStatement : Letter ==> ()
  PostStatement(letter) ==
    statements := statements ^ [letter];

  public GetLastStatement : () ==> Letter
  GetLastStatement() ==
    return statements(len statements)
  pre statements <> [];

end Letterbox
```

class Till

```
instance variables
  curCard : [Card] := nil;
  cardOk : bool := false;
  retainedCards : set of Card := {};
  resource : CentralResource;

  inv curCard = nil => not cardOk;

operations
  public Till: CentralResource ==> Till
  Till(res) ==
    resource := res;

public InsertCard : Card ==> ()
  InsertCard(c) ==
    curCard := c
  pre not CardInside();

public Validate : Card`PinCode ==> <PinOk> | <PinNotOk> |
<Retained>
  Validate(pin) ==
    let cardId = curCard.GetCardId(),
        codeOk = curCard.GetCode() = Encode(pin),
        cardLegal = IsLegalCard()
    in
      (cardOk := codeOk and cardLegal;
       if not cardLegal then
         (retainedCards := retainedCards union {curCard};
          curCard := nil;
          return <Retained>)
       elseif codeOk then
         resource.ResetNumberOfTries(cardId)
       else
         (resource.IncrNumberOfTries(cardId);
          if resource.NumberOfTriesExceeded(cardId) then
            (retainedCards := retainedCards union {curCard};
             cardOk := false;
             curCard := nil;
             return <Retained>));
       return if cardOk
              then <PinOk>
              else <PinNotOk>)
  pre CardInside() and not cardOk;

public ReturnCard : () ==> ()
  ReturnCard() ==
    (cardOk := false;
     curCard:= nil)
  pre CardInside();

public GetBalance : () ==> [nat]
  GetBalance() ==
    resource.GetBalance(curCard.GetAccountId())
  pre CardValidated();

public MakeWithdrawal : nat ==> bool
  MakeWithdrawal(amount) ==
    resource.Withdrawal
      (curCard.GetAccountId(),curCard.GetCardId(),amount)
  pre CardValidated();

  public RequestStatement : () ==> bool
  RequestStatement() ==
resource.PostStatement(curCard.GetAccountId(),curCard.GetCardId()
)
  pre CardValidated();

public IsLegalCard : () ==> bool
  IsLegalCard() ==
    return

resource.IsLegalCard(curCard.GetAccountId(),curCard.GetCardId())
  pre CardInside();

  public CardValidated: () ==> bool
  CardValidated() ==
    return curCard <> nil and cardOk;

  public CardInside: () ==> bool
  CardInside() ==
    return curCard <> nil;

functions

Encode: Card`PinCode +> Card`Code
  Encode(pin) ==
    pin; -- NB! The actual encoding procedure has not yet been
chosen

end Till
```

Annex B

VDM++ Collection Operators

Set Types

Operator	Name	Signature
e in set s1	Membership	A * set of A →bool
e not in set s1	Not membership	A * set of A → bool
s1 union s2	Union	set of A * set of A → set of A
s1 inter s2	Intersection	set of A * set of A → set of A
s1 \ s2	Difference	set of A * set of A → set of A
s1 subset s2	Subset	set of A * set of A → bool
s1psubset s2	Proper	subset set of A * set of A → bool
s1 = s2	Equality	set of A * set of A → bool
s1<> s2	Inequality	set of A * set of A → bool
card s1	Cardinality	set of A → nat
dunionss	Distributed	union set of set of A → set of A
dinterss	Distributed intersection	set of set of A → set of A
powerss	Finite power set	set of A → set of set of A

Sequence Types

Operator	Name	Signature
hd l	Head	seq1 of A→ A
tl l	Tail	seq1 of A→ seq of A
concll	Distributed concatenation	seq of seq of A→ seq of A
l ++ m	Sequence modification	seq of A * map nat to A→ seq of A
l(i)	Sequence index	seq of A * nat1→ A
l1 = l2	Equality	(seq of A) * (seq of A)→ bool
l1<> l2	Inequality	(seq of A) * (seq of A)→ bool
len l	Length	seq of A→ nat
elems l	Elements	seq of A→ set of A

inds l	Indices	seq of A→ set of nat1
l1 ˆ l2	Concatenation	(seq of A) * (seq of A)→ seq of A

Mapping Types

Operator	Name	Signature
dom m	Domain	(map A to B) → set of A
rngm	Range	(map A to B) → set of B
m1munion m2	Map union	(map A to B) * (map A to B) → map A to B
m1 ++ m2	Override	(map A to B) * (map A to B) → map A to B
merge ms	Distributed merge	set of (map A to B) → map A to B
s<: m	Domain restrict to	(set of A) * (map A to B) → map A to B
s<-: m	Domain restrict by	(set of A) * (map A to B) → map A to B
m :> s	Range restrict to	(map A to B) * (set of B) → map A to B
m :-> s	Range restrict by	(map A to B) * (set of B) → map A to B
m(d)	Mapping apply	(map A to B) * A → B
inverse m	Map inverse	inmap A to B → inmap B to A
m1 = m2	Equality	(map A to B)*(map A to B) → bool
m1<> m2	Inequeality	(map A to B)*(map A to B) → bool

Annex C

XML Example Description

```
<?xmlversion="1.0" encoding="UTF-8" standalone="yes"?>
<frame xmlns="http://www.swixml.org/2007/Swixml" id="Stack_instance_window" size="320,520" title="Stack">
<panelconstraints="BorderLayout.LEFT">
<panel>
<labeltext="CurrentInstance: "/>
<comboboxid="instance_list_wiget"/>
</panel>
<vboxid="methods_container">
<panelid="panel_Reset_Stack" layout="FlowLayout(FlowLayout.LEFT)">
<buttonid="method_button_widgetReset" text="Reset"/>
<labelid="check_method_return_Reset"/>
</panel>
<panelid="panel_Pop_Stack" layout="FlowLayout(FlowLayout.LEFT)">
<buttonid="method_button_widgetPop" text="Pop"/>
<labelid="check_method_return_Pop"/>
</panel>
<panelid="panel_pre_Pop_Stack" layout="FlowLayout(FlowLayout.LEFT)">
<labeltext="self"/>
<comboboxid="pre_Pop_arg0"/>
<buttonid="method_button_widgetpre_Pop" text="pre_Pop"/>
<labelid="check_method_return_pre_Pop"/>
</panel>
<panelid="panel_post_Pop_Stack" layout="FlowLayout(FlowLayout.LEFT)">
<labeltext="RESULT"/>
<textfieldid="post_Pop_arg0" columns="4"/>
<labeltext="self~"/>
<comboboxid="post_Pop_arg1"/>
<labeltext="self"/>
<comboboxid="post_Pop_arg2"/>
```

```
<buttonid="method_button_widgetpost_Pop" text="post_Pop"/>
<labelid="check_method_return_post_Pop"/>
</panel>
<panelid="panel_Push_Stack" layout="FlowLayout(FlowLayout.LEFT)">
<labeltext="elem"/>
<textfieldid="Push_arg0" columns="4"/>
<buttonid="method_button_widgetPush" text="Push"/>
<labelid="check_method_return_Push"/>
</panel>
<panelid="panel_Top_Stack" layout="FlowLayout(FlowLayout.LEFT)">
<buttonid="method_button_widgetTop" text="Top"/>
<labelid="check_method_return_Top"/>
</panel>
</vbox>
</panel>
</frame>
<?xmlversion="1.0" encoding="UTF-8" standalone="yes"?>
<frame xmlns="http://www.swixml.org/2007/Swixml" id="UseStack_instance_window"
size="320,520" title="UseStack">
<panelconstraints="BorderLayout.LEFT">
<panel>
<labeltext="CurrentInstance: "/>
<comboboxid="instance_list_wiget"/>
</panel>
<vboxid="methods_container"/>
</panel>
</frame>
```