Automatic Generation of GUI from VDM++ Specifications

VDM++ GUI Builder

Carlos A. L. Nunes
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
ei05095@fe.up.pt

Ana C. R. Paiva
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
apaiva@fe.up.pt

***Abstract*—The Vienna Development Method is supported by several tools. These tools allow generating Java code from a VDM++ specification but do not generate a graphical user interface (GUI). This paper describes a generic approach and tool to automatically generate a GUI in Java from a VDM++ specification. The generated GUI calls methods of the VDM++ specification, which allows testing the specification itself in order to increase confidence that it is an accurate description of the intended behaviour. This GUI may evolve to interact with the already supported generation code in Java (for the API) in order to obtain a complete application from a VDM++ specification based on a fully automatic code generation process.**

Keywords-Formal Methods; Graphical User Interfaces; Vienna Development Method; Automatic Code Generation

I. INTRODUCTION

In the development of a VDM++ specification, interaction with the underlying model is usually done by the use of an interpreter – VDMTools [1] or VDMJ [2]. Although current tools provide an API to externally use the interpreter [1, 2], they offer little more than a way to establish the connection. As this stands, in order to create a Graphical User Interface (GUI) to interact with a VDM++ specification, a developer is forced to design and implement it from the ground up, and also create the necessary "glue" between the VDM interpreter/tool and the GUI.

Using automatic code generation techniques from a formal specification, this research work puts forward an approach that allows users to interact with a VDM++ specification through an automatically generated GUI. Enabling the developer to execute and test the VDM++ specification without the direct use of an interpreter.

Additionally, the generated GUI may be considered as an evolutionary prototype and be connected with the API code generated by current tools, in the following steps of the development process, in order to provide a complete application obtained by a fully automatic code generation process.

This paper is organized as follows: Section II introduces related work; Section III presents basic concepts related to the context of this work; Section IV describes the GUI generator tool and its approach; Section V presents a case study; Section VI discusses the results of a case study; and Section VII presents conclusions and future work.

II. STATE OF THE ART

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 user 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.

A. 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 [3].

Two examples of such tools, currently in use, are the Glade interface builder [4] and the interface builder component of the NetBeans integrated development environment [5].

B. 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, instead of functionality [6].

Examples of user interface markup languages include UsiXML [7], XAML [8], XUL [9] and SwiXML [10].

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

 ISBN: 978-1-61208-165-6

C. 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 [11] .

D. Formal Language-Based Tools

The motivation behind the use of existing formal method based techniques is a strong emphasis on dialog management. 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 [3].

E. Constraints

"A constraint can be thought of intuitively as a 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 (…)" [12]. This concept can be used to implement several different aspects of a user interface. Two examples of such a tool are Amulet [13, 14] and Subarctic [3].

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, as far as we know, 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 [3].

Nevertheless, constraints are widely used for layout control. NeXTStep, for example, provided a limited form of constraints that could be used to control layout [3]. 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 [15].

F. 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.

Early examples of such tools are UIDE [3] and HUMANOID [16]. 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 case models [17].

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 [3, 18].

G. Summary

The tools or techniques, described above, 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 approach can be considered an "Automatic Model-Based Technique", with the distinguishing features of not relying on a special purpose modeling language, and the removal of unpredictability. Another new aspect is the use of a XML markup language to describe the user interface, giving a greater degree of freedom to make alterations after the automatic generation. No attention to user interface functionality is required from the developer.

 ISBN: 978-1-61208-165-6

III. BASIC CONCEPTS

A. 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 [19]. 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. As for system verification, formal reasoning techniques are used [19, 20] [21, 22].

B. The Vienna Development Method (VDM) Language

The Vienna Development Method is one of the oldest formal methods [23]. Initially the method only possessed a meta-language for specification, but evolved to include the VDM++ specification language. The VDM++ language is an object-oriented version of the VDM-SL formal language. Apart from classes, the VDM++ language includes instance variables, operations, functions, types, operators and expressions. As with the VDM-SL, VDM++ allows the definition of invariants, pre-conditions and post-conditions.

Besides basic types, such as Boolean and numeric, the language includes three collection types – set, seq and map. A set consists of a 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 relating elements of type A with elements of type B [24] [25].

IV. VDM++ GUI BUILDER

The VDM++ GUI builder generates a GUI from a VDM++ specification. VDM++ can be used to model virtually any kind of system. So the GUI generation approach should be generic enough to work on any kind of modelled system.

A. Architecture

The VDM++ GUI Builder is integrated with the tools developed in the context of Overture Tool Project [26] – an open source project to develop a set of high quality formal modelling tools, built on top of the Eclipse Platform [27].

As such, the VDMJ engine [2] 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 [10]. This engine is used to render the GUI elements from a XML description generated by the VDM++ GUI Builder. This tool was chosen because it is specifically designed for Java applications and possesses a very simple mechanism for UI element search. The tool optionally assigns an id for each UI element, which can be used in runtime mode to retrieve the corresponding UI element.

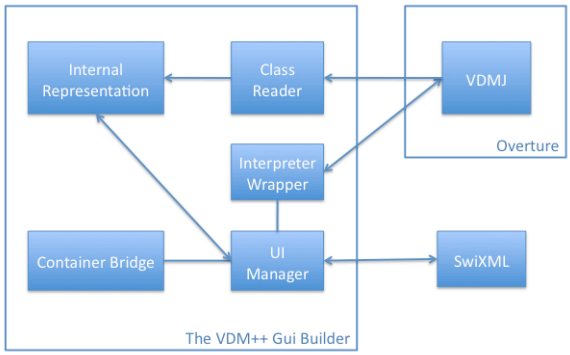

Figure 1. Diagram of the architechture

As shown in Figure 1, 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. This module relies heavily on VDMJ to extract such information. Even though, this module can be replaced with another one in order to use the VDM++ GUI Builder with other tools different from the ones available within the Overture project.

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.

B. 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 it does not require an extension to the VDM++ grammar. The annotations take the form of "--@name=value" or "--@name" and are handled separately by the approach.

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

 ISBN: 978-1-61208-165-6

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.

C. 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 (Figure 8), and another to show all the class instances created in each moment of the execution, for debugging purposes (Figure 10).

Apart from annotations, the GUI elements are generated from the analysis of the signatures of the methods of the underlying class.

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 (Figure 2). In cases where the parameter is a class, the generated GUI provides a combo box with the class instances created until that moment.

Figure 2. Example of a generated window from the "Dining" VDM++ example (//overture.svn.sourceforge.net/)

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.

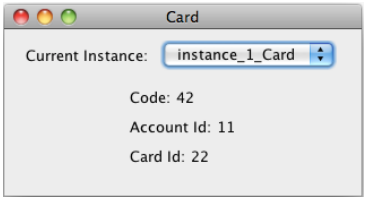

Figure 3. Example of a window generated from the class Card (with 'check' annotated methods) of the Dispenser system used in the case study.

All windows generated from VDM++ specification classes have a drop-down list. This list (labelled "Current Instance" in Figure 3) contains all the instances of such class.

Such list also contains a "new" option to allow the construction of new instances. This 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 (Figure 4) when there is a constructor with arguments.

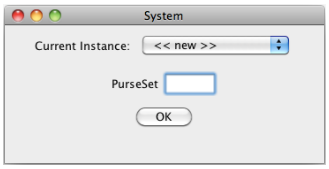

Figure 4. Example of a window generated from the class System of the "ElectronicPurse" specification found in //overture.svn.sourceforge.net .

D. 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.

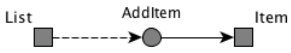

Figure 5. Graph representing the dependencies of simple list system.

The above graph (Figure 5) 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 5). This operation requires the existence of an "Item" instance to be enabled (dependency represented by a solid arrow in Figure 5).

Extending the previous example, so that a "List" requires a "Person", would generate the dependency graph in Figure 6 which means that it will be possible to construct List instances only after creating Person instances.

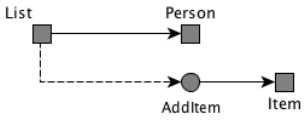

Figure 6. Graph representing the dependencies of the extended system.

 ISBN: 978-1-61208-165-6

V. CASE STUDY

The Overture Project provides several examples of VDM++ specifications (//overture.svn.sourceforge.net/). In order to evaluate the approach, including the use of annotations, the Cash Dispenser system was selected. The specification describes a system that allows the withdrawal money from accounts using a card and a till. The system keeps record of issued cards, cardholders and current accounts, and can issue card statements to the cardholders. The VDM++ specification used in this experiment includes the class "SimpleTest" used as a test case. Since this class does not specify additional behaviour of the system being modelled, the "--@nowindow" annotation was added to it.

The following figure depicts the dependencies that the specification has, according to the previously described approach. Note that in Figure 7, only classes, operations and functions with dependencies are represented.

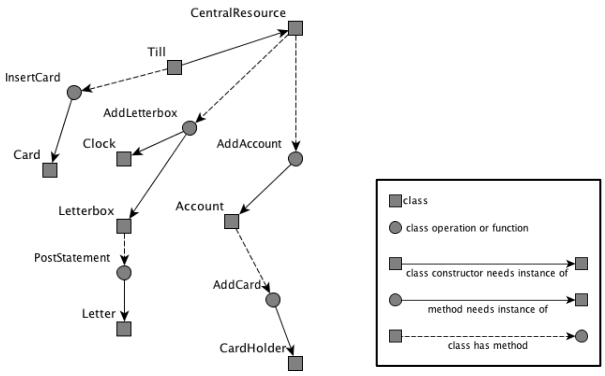

Figure 7. The Cash Dispenser system dependency graph.

The generated main window is shown in Figure 8. The Till button is initially disabled because there is a dependency between Till class and CentralResource class (represented by a solid arrow in Figure 7) which means that an instance of CentralResource in needed in order to construct a Till instance.

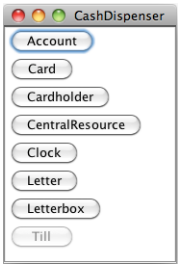

Figure 8. The main window with the Till button disabled

An instance of CentralResource class is immediately constructed when opening the corresponding window (Figure 9) 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 7. "AddLetterBox" method is enabled after creating instances of "Clock" and "Letterbox" classes. "AddAccount" method is enabled after constructing instances of the "Account" class.

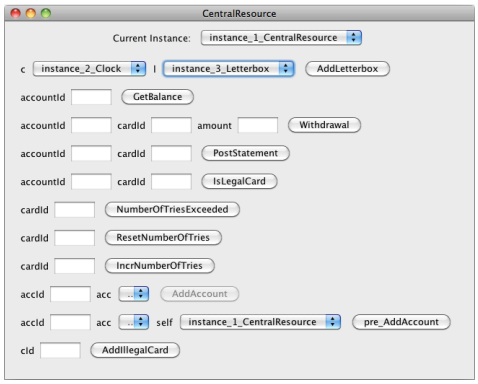

Figure 9. The "CentralResource" window with AddLetterBox and AddAccount buttons disabled

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.

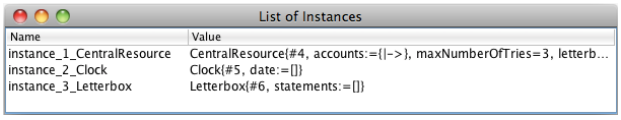

Figure 10. The list of instance window, after creating the instances.

VI. DISCUSSION

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 modelling effort, which could put into question the goal of this research work: generate a GUI from a generic VDM++ specification with minimal additional 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.

VII. CONCLUSION 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 a dependency graph extracted from the analysis of GUI specification methods. The approach achieves this while following the grammar of the VDM++ formal language and

 ISBN: 978-1-61208-165-6

without requiring the user active participation in the GUI generation process.

Furthermore, by adding annotation with additional information to the VDM++ specification, it is possible a better adjustment of the GUI elements generated.

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 in [28], this approach uses a user interface pattern [29] 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.

REFERENCES

[1] C.S.K. Corporation, *The VDM Toolbox API 1.1*, 2008.

[2] N. Battle, *VDMJ Tool Support: User Guide*, 2011.

[3] B. Myers, S.E. Hudson, and R. Pausch, *Past, present, and future of user interface software tools.* ACM Trans. Comput.-Hum. Interact., 2000. **7**(1): pp. 3-28.

[4] D. Aitel, *A beginner's guide to using pyGTK and Glade.* Linux J., 2003. (113): p. 5.

[5] Oracle. *Lesson: Using the NetBeans GUI Builder*. [2011 6/3/2011]; Available from: http://download.oracle.com/javase/tutorial/javabeans/nb/.

[6] J. Bishop, *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. pp. 751-760.

[7] Q. Lambourg, et al., *USIXML: A Language Supporting Multi-path Development of User Interfaces*, in *Lecture Note in Computer Science*2005. pp. 134-135.

[8] Microsoft Corp., *XAML Overview (WPF)*. [29/06/2011]; Available from: http://msdn.microsoft.com/en-us/en-us/library/ms752059.aspx.

[9] M.D.N., *The Joy of XUL*. [28/05/2011]; Available from: https://developer.mozilla.org/en/The_Joy_of_XUL.

[10] W. Paulus, *SwiXML* [03/06/2011]; Available from: http://www.swixml.org/.

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

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

[13] B.A. Myers, 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. pp. 214-215.

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

[15] I. Darwin, *GUI Development with Java.* Linux J., 1999. **1999**(61es): pp. 4.

[16] P. Szekely, 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. pp. 507-515.

[17] A. Rosado 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. pp. 256-270.

[18] J. Nichols, 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. pp. 1283-1292.

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

[20] D. Bjørner, *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. pp. 326-359.

[21] *VDMTools: advances in support for formal modeling in VDM.* SIGPLAN Not., 2008. **43**(2): pp. 3-11.

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

[23] P.G. Larsen 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.

[24] J.S. Pedersen and K.H. Shingler, *Software Development Using VDM*, 1989.

[25] P.G. Larsen, et al., *VDM-10 Language Manual*, 2011.

[26] P.G. Larsen, et al., *Tutorial for Overture/VDM++*, 2010.

[27] A. Wolfe, *Eclipse: A Platform Becomes an Open-Source Woodstock.* Queue, 2003. **1**(8): pp. 14-16.

[28] L. Aversano, et al., *An empirical study on the evolution of design patterns*, in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* 2007, ACM: Dubrovnik, Croatia. pp. 385-394.

[29] A. Granlund, D. Lafrenière, and D.A. Carr. *A Pattern-Supported Approach to User Interface Design Process*. in *HCI International'2001*. 2001. New Orleans: Lawrence Erlbaum Associates.

 ISBN: 978-1-61208-165-6