

A High-Level Visual Test Specification Model for DSVL

M. F. Jaafar*, J. Grundy**, J. Hosking***

* Department of Electrical and Computer Engineering,

*** Department of Computer Science,

The University of Auckland, Auckland 1142, New Zealand.

** Department of Computer Science and Software Engineering,

Swinburne University of Technology, Victoria 3122, Australia.

Emails: mjaa001@aucklanduni.ac.nz, jgrundy@swin.edu.au, john@aucklanduni.ac.nz

Abstract

Domain-Specific Visual Languages (DSVLs) have captured the attention of the programming language world with their simplicity and high-level abstraction. This has encouraged many to use DSVLs as a way to write programs. With little or no programming knowledge, many end-users can program tasks that would be beyond them with conventional programming. Despite their benefits however, DSVLs need validation, just as conventional programs do. Tests are typically done manually and few DSVLs support testing processes inside the language or tool. Motivated by this, we propose a high-level visual test specification model that resides inside DSVL programs. This specification model enables users to design tests within their domains, providing a way to validate their development models.

Keywords: Test Specification Model, Domain-Specific Visual Language, Testing, Meta-Tool, Automated Test Generation.

1 Introduction

Domain-specific visual languages or DSVLs are special types of programming language that use icons and graphical notations to code programs. With their simplicity and high-level abstractions, visual languages have been promoted as better than text-based programming [ref?? – perhaps Shu?]. The use of DSVLs in the programming world can be seen in various fields including the financial, engineering, and medical domains [1].

The reason behind their success is that they combine the power and flexibility of programming languages with the ease of graphical interfaces. They help users who have little or no understanding of programming to express their intentions using high-level representation. Another reason DSVLs have gained momentum is because of the existence of the DSVL meta-tools, tools that help to create DSVL tools. Marama [2] and Microsoft DSL [3] are two examples of these meta-tools. Using these tools, a developer can create new

DSVLs based on the templates and graphical representations provided. Even end-users can now create a DSVL tool and share it with others. Unfortunately, with all these advantages, the validation problem remains open. Testing is still being done manually or with the help of third-party testing tools. End-users are required to use the saved time (while creating the programmes) to create tests.

2 Research Question and Motivation

While there is a growing number of large IDE's for writing codes, there are fewer for testing [4]; this is also an issue in the DSVL domain. Current meta-tools are excellent for supporting the creation of DSVL tools, and yet, fail to assist the user with the verification and validation process. Meta-tools like Marama and Microsoft DSL only support testing to the extent of text-based testing. It is awkward to use text-based testing for DSVL programmes where almost everything is achieved visually. End-users are required to fill in programme codes and test data manually, which means that they have to revert to something that they have moved away from.

Although visualization reduces the complexity in programming language, it brings new problems. Lack of attention or misunderstanding of notational characters may cause unintentional errors [5] or, in this case, unintended tests. Creating test support for DSVLs is not an easy task and, in general, it is a huge concept to start with. DSVLs can be developed for different domains. Specific DSVLs contain attributes that are not presented in other DSVLs. Flexibility in a generic DSVL test support tool is therefore required. Our key research questions are:

- Can DSVL approaches be used to model tests for DSVL programs at high-levels of abstraction?
- Can such DSVL test models be used to generate and run automated test tool scripts?
- What domains can such approaches be applied to?
- Can a DSVL meta-tool be extended to specify and generate such visual approaches in order to test programs created by the DSVL tools implemented?

Testing is a tedious task and requires much effort. Having an IDE that could facilitate testing and its processes would reduce this effort. The need to have a test design tool has been clearly documented in [6]. Even, researchers in [7] have discuss combining requirements engineering and interaction design to help with development processes. It seems that requirements engineering have reached a new level and need visual interaction for assistance. This has motivated us to explore the possibility of assisting the DSVL program testing process since most DSVL users lack any deep programming or testing knowledge.

The main aim of this study is to propose, implement and validate whether a meta-tool can be extended with test specification support for DSVLs built using the meta-tool. We want to extend a current meta-tool’s potential from just domain-specific language and application creation to support the DSVL testing processes.

3 Existing Work

Various methods have been introduced to create tests automatically, either from program codes [8, 9] or development documents [10-12]. Testing tools like JUnit and NUnit focus on textual programming languages like Java and C#. One example for test support in the end-user programming domain is the ability to create tests for spreadsheet applications [13]. Here, the “What you see is what you test” (WYSIWYT) methodology was used to assist test creation. We believe this method is relevant in designing a test specification model for DSVLs, as it is concerned with creating tests from artefacts that users see (in the DSVL case, the development model). With this method, end-users can reuse the development model and specify tests from it.

In more recent examples, tests have been created based on user requirements. The first example is where a test is generated from the viewpoint of an end-user who has created a security requirement [14]. The system has the ability to suggest to the end-user if there is any lack in their system. The same is true in [15], where a model is created from a document specification and then kept as abstract as possible to match the textual specification. Other researchers [16, 17] have derived test cases from a DSVL. The created test is independent of any programming language and is transferable across platforms. We believed that creating test specifications that are independent of a programming language better empowers end-users.

As well as creating test cases, we are interested in exploring visual approaches in test reporting which is an important part of testing. Test reporting is the communication point between end-users and the application created. Existing testing tools provide test report to a certain extent [8]. A report typically consists of the number of pass, fail and unexercised tests. Only a

few exceed this stereotype. For instance, [10] describes a report showing the defects and test paths that were exercised during the test activity. Users are allowed to select the path node and see the failed test details too.

Alternatively, [18] illustrates the testing process with animation. This helps the user to identify improper use of modelling constructs. A recent study by [19] has created another approach for visualizing the test execution. Although this is promising, we believe it should have more interactive capabilities allowing the end-users to select and rerun the fail tests with new data. [20] and [21] suggest that visualization enables the user to understand faults and how to debug the programme. These examples indicate that visualization plays an important role in helping end-users to understand their application more effectively.

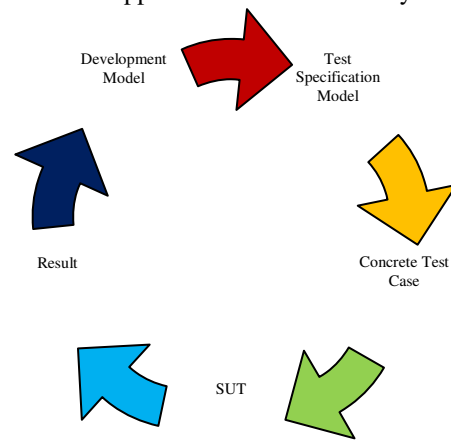


Fig. 1 Overview of testing life cycle in DSVL

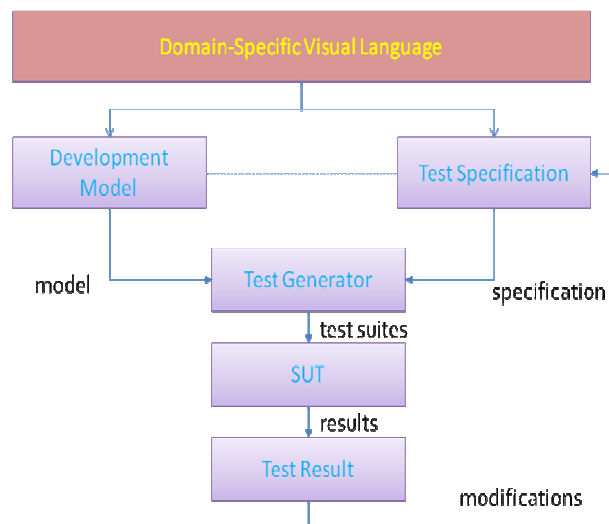


Fig. 2 Framework for creating test from DSVL

4 Proposed Solution

The aim of our research is to design a generic high-level visual test specification meta-tool prototype that allows users (developer and end-users) to create

specific test specification models and tools. By taking this approach, we hope to eliminate the need to use textual testing specifications and scripts. In addition, the same model can be used to examine execution results within the development environment.

Figure 1 shows the life cycle of the proposed testing process for DSVL. The process starts with the DSVL development model and follows by creating the test specification model. Then, concrete test cases are generated and given to the system under test (SUT). Finally, the test results are gathered and visualized within the development model.

Illustrated in figure 2 is the framework for creating test from DSVL. The test should be generated from any type of DSVL development model combined with a test specification. These combinations are then fed to a test generator to create concrete test cases which will be executed by the SUT. The test execution results gathered are fed to the test model specification for result annotations. Within this framework, the test specification models have two functions:

- (i) specifying tests, and
- (ii) annotating test results.

5 Contributions

This research focuses on how a DSVL can be used to support the validation process for other DSVL programs. The framework will provide guidelines for creating a test specification model for a DSVL programme, realised by a DSVL tool developed using a meta-tool. Listed are the expected contributions from this research:

- Modelling and visualizing tests using a DSVL
- Generating and executing concrete tests from DSVL test models
- Extending a DSVL meta-tool to support the testing process

Currently, we have demonstrated that concrete tests can be produced from a DSVL test specification model. We also have confirmed that the test specification model can be used to annotate test results. Our work is ongoing to identify what types of DSVL are suitable for use with our proposed test models.

6 Methodology

This research uses the methodology listed below:

- Conduct a literature review on model-based testing, test generation, and visual test reports to understand current approaches;
- Design initial test specification model and test report layout;
- Implement the test specification model functions inside a meta-tool;

- Generate and support execution of the concrete test cases and test scripts;
- Evaluate the model and accompanying tool using real world examples and representative end-users group.

Iterative and incremental development methodologies are used to prove our framework. This cycle starts with designing the test specification model, implementing it in a DSVL meta-tool and then evaluating it in order to verify the effectiveness of the model. The result is then used in the next iteration development. A complete test model specification will be implemented and evaluated in the last development cycle.

6.1 Designing the Modelling Language

In order to create a test specification model that co-exists with DSVLs, several criteria need to be addressed. To help us with this, we have followed the guidelines provided in [22] and have also conducted a literature review on past methods used to create tests from model-based testing and UML notations (as UML is an example of a DSVL). At the moment, we are investigating several approaches to identify the best method for visualizing test execution results.

6.2 Implementing the Design

Marama has been chosen as the meta-tool to help us prove this framework. It has model generation capabilities, which allow customizable functions and a usable GUI. In addition, Marama supports model integration, which can be used with our test specification model, in order to identify and implement the test specification model. Marama is:

- (i) used to create DSVLs prototypes,
- (ii) used to design and create test specification models, and
- (iii) extended to support test specification creation and test generation and execution visualisation in a DSVL meta-tool.

6.3 Evaluating the Model

In order to demonstrate the validation of the proposed model, we have selected two criteria [23, 24] that are relevant for end-users. They are briefly explained:

Model representation: This is concerned with the representation used for defining the test specifications or contributing towards in the creation of tests.

Usability: This addresses the effort required to learn and use the test specification language provided. We also aim to identify the effectiveness of designing test specifications using the model and tool provided.

We will apply the test specification model with several case study examples and conduct a user survey. For the

case study, the test specification model should be able to be used to:

- (i) generate an intended test, and
- (ii) find errors seeded in the programme.

We will deliberately seed a number of different errors into the programme. All of these errors will be logic errors that can occur in programming. We skip syntax errors because these should be catered for by the language editor and compiler. We expect that the tests generated will be able to find all the seeded faults.

We will also undertake a survey on each DSVLs prototype to obtain end-user feedback on the model and accompanying tool usability. The survey will ask questions related to the ease of use and the support given by the test specification model and accompanying tool.

The evaluation will be conducted in two stages:

- (i) During the initial prototype development. A quick survey will ask end-users about feasibility and practicability of the test specification model and accompanying tool for specifying tests. Results

obtained will be reviewed for a possible significant research improvement.

- (ii) At the last stage of development, when we will fully validate our proposed meta-tool model.

7 Progress

Until now, we have developed two working prototypes of DSVLs to evaluate our test model. The first prototype is MaramaEUC, used for modelling essential use cases. The second prototype is MaramaFB, created for drawing function block diagrams design based on IEC 41699 standard. Both of these DSVL prototypes were developed using the Marama meta-tool.

7.1 MaramaEUC

Essential use cases or EUCs are an extended version of use case but from the user view [25]. They are simpler than UML use case models only requiring users to specify their intention and the possible system response at an abstract level. Thus, it tries to capture requirements without relying on a technology or implementation bias [26].

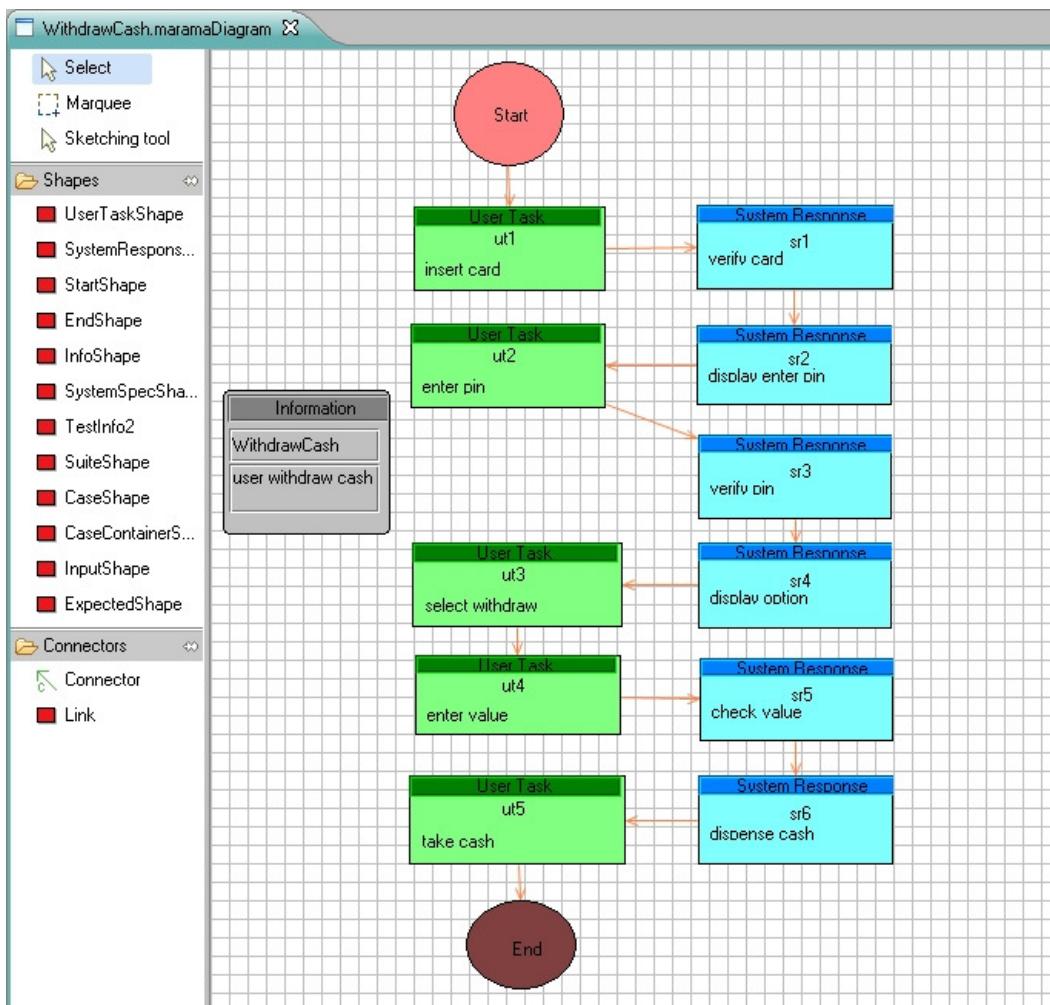


Fig. 3 Drawing essential use case with MaramaEUC

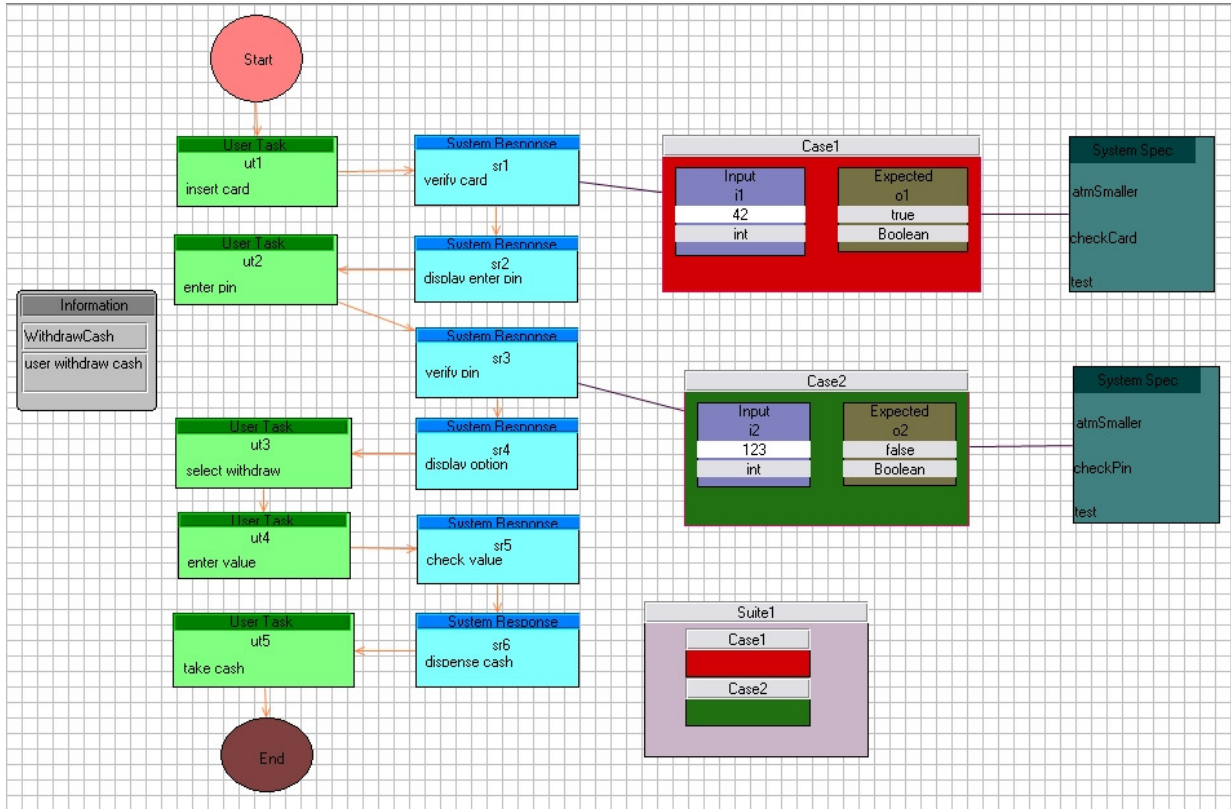


Fig. 4 MaramaEUC with Test Specification Model

MaramaEssential is a visual modelling tool for specifying EUCs. In MaramaEssential, users can create two main entities; User Tasks and System Responses, which can be linked with a connection arrow (to show the flow of process). Figure 3 shows an example of an essential use case model.

In this prototype, our approach was to specify tests with a small set of icons that extend the DSLV “programming” environment and can be used to explicitly annotate the DSLV programme with test specification information. Figure 4 presents the extended version of the essential use case with a test specification model shown alongside (fig. 3).

In figure 4, test specification is conducted by linking a test case icon to the essential use case icon. Test oracles (input and expected output) are specified inside the test case icon. The collection of test cases is placed inside a test suite icon to organize the test. Finally, concrete test cases are generated based on the chosen template. At this stage, JUnit is our main test template, as our initial SUT was implemented using Java. For a complete list of test specification models and examples of generated concrete tests, please refer to the Appendix.

We have mentioned test visualization in our proposed solution. Hence, in the MaramaEUCTest prototype, the test case icon (besides functioning as test case) also functions as the test result reporter. The test case icon changes colour from yellow (the default colour) to green (for a passed test) or red (for a failed test). By doing this, we help to reduce the need to refer to the

text-based test report. Furthermore, we have reused the test specification model to facilitate results reporting.

7.2 MaramaFB

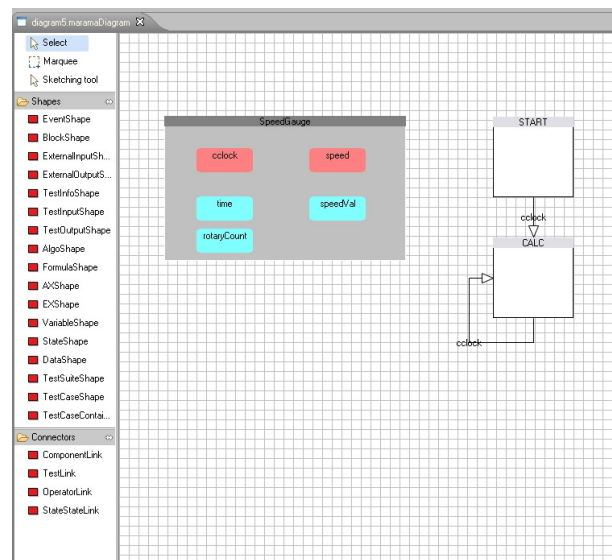


Fig. 5 Drawing function block diagram with MaramaFB

A function block diagram describes functions between input and output variables [27] and is used mainly to describe the programming logic control inside an embedded application. MaramaFB is our version of a function block diagram tool created using Marama.

In MaramaFB, users can draw a block diagram using a provided set of icons, which can be divided into two categories;

- (i) Interface icons – this represent the function block components. They consist of a Block icon (representing a basic block), an Event icon (representing an event), and a Data icon (representing an event variable).
- (ii) Execution Control Chart (ECC) icons – these represent the logic control inside a basic block. This consists of a State icon and a Transition link.

Figure 5 shows an example of a function block diagram model. MaramaFB was chosen as the second prototype since it has different characteristics from the first prototype, MaramaEUC. MaramaEUC is a high-level abstraction model that works with user requirements. MaramaFB works with visual languages that directly specify control logic programming. With different types of domain-specific language, we will be able to observe if the test specification model is applicable to other domains.

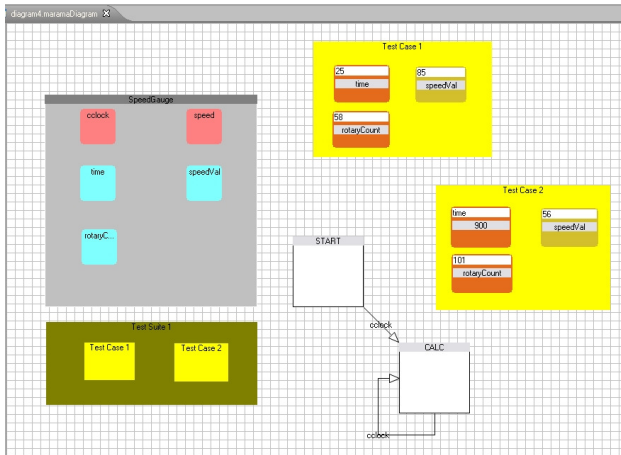


Fig. 6 MaramaFB with Test Specification Model

The same test specification model was used but with some modification to the test oracle. We need to explore the possibility of having more than one input (or expected output) in the test (this was not present in the first prototype). Figure 6 shows MaramaFB with its test specification model.

Another improvement that we are implementing in the second prototype is the effect of changing the test suite icon colour according to the number of test case results. At this point of development, the test suite icon does not contribute anything to the visualization test result. We believe the test suite icon could be extended into something similar to the test case icon. The idea here is to use a simple colour calculation method and change the icon colour based on the number of test cases passed or failed. If the number of test passes is more than the number of test failed, it would have a colour that leans more towards green. It is vice versa if the number of failed test is more.

Currently, we are working on generating and proving that concrete tests can be produced from function block diagrams and test specification models. The process is however more complicated, mainly because each diagram contains more than one input and expected output. To solve this problem, a model checker is added to the test case generator. Model checking is a technique for verifying models based on given formulae specification. The results from the model checker are fed to the test generator and used to produce concrete test cases.

8 Future Directions

We will continue to implement and improve our proposed test specification model on other types of DSVL domains and are keen to explore the possibility of using our test specification model on web-based DSVLs (or embed it inside an existing DSVL). We plan to embed it into MaramaMTE (Middleware Testing Environment) which is a tool for modeling complex software architectures and generating performance test beds [28]. This will prove whether the test specification model can be used on other testing tool or not.

The current prototype uses our built-in test generators. Although the test generator works fine for now, we would like to see whether the test specification model can be used effectively with other test case generator. To achieve this, we are planning to create an export function that converts the test specification model into an XML data file. XML is chosen because most of the current testing tools share data (or specifications) using this format.

As mention earlier, an evaluation will be carried out to examine the possibility of using the test specification model to assists users in creating tests. We will examine both users' opinions on the model and the tool ability to create concrete tests from given specifications. The results will help us to design a generalized high-level visual test specification model meta-tool.

Finally, we would like to explore the prospect of modelling and generating DSVL test support from the DSVL meta-tool. By doing this, it can increased and broaden the scope DSVL meta-tool from just supporting the creation of DSVL to validating its content. This will definitely make DSVLs a pure visual programming language.

9 Conclusions

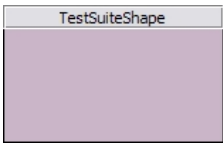
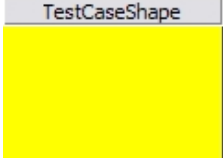
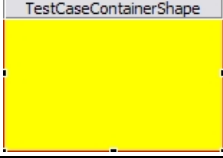
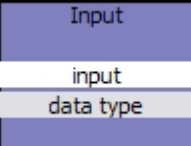
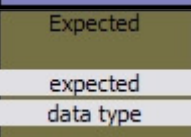
This research aims to create a high-level visual test specification model that can be used by DSVLs' end-users to specify and create tests alongside their DSVL programs. To achieve this, the past model-based approaches to test case specification, generation and visualization were analysed and their strengths and weaknesses are explored. As a result, we have developed two prototypes to help us understand the challenges in developing a visual test annotation. Up to

now, the progress shows that this is achievable and has the potential to succeed.

Appendix

Table 1 shows the test specification model and its functions for MaramaEUC.

Table 1 Test Specification Model

Shapes	Descriptions
	The test suite shape represents test suite in testing. It is a place to group test cases together. It must contain at least one test case.
	The test case shape represents test case in testing. It is place inside test suite to symbolize that the test belongs to the test suite. The shape color can change depends on test execution result.
	Test case container shape is equivalent to test case shape. The different is that it can be used as container for input and expected output.
	Input shape represents the input value for testing. It can take any kind of data. User needs to specify the value and its data type.
	Same like input shape, expected shape represents the expected output value for testing.

```

package test;

import test.atmSmaller;

public class WithdrawCash_Case1 extends TestCase {
    public WithdrawCash_Case1(String name) {
        super(name);
    }

    public static TestSuite suite() {
        return new TestSuite(WithdrawCash_Case1.class);
    }

    atmSmaller objectClass = new atmSmaller();
    public void testcheckCard() {
        Boolean result = objectClass.checkCard(42);
        Boolean expected = false;
        assertEquals(expected, result);
    }
}

```

Fig. 7 Concrete test for Case1

These are the examples of the concrete test cases and test suite created from our built-in test generator, based on the test specification model in Figure 3. Figure 7 and 8 show the generated concrete test cases.

```

package test;

import test.atmSmaller;

public class WithdrawCash_Case2 extends TestCase {
    public WithdrawCash_Case2(String name) {
        super(name);
    }

    public static TestSuite suite() {
        return new TestSuite(WithdrawCash_Case2.class);
    }

    atmSmaller objectClass = new atmSmaller();
    public void testcheckPin() {
        Boolean result = objectClass.checkPin(123);
        Boolean expected = false;
        assertEquals(expected, result);
    }
}

```

Fig. 8 Concrete test for Case2

```

package test;

import junit.framework.Test;

public class WithdrawCash_Suite1 extends TestSuite {
    static public Test suite() {
        TestSuite suite = new TestSuite();

        suite.addTestSuite(WithdrawCash_Case1.class);
        suite.addTestSuite(WithdrawCash_Case2.class);
        return suite;
    }
}

```

Fig. 9 Concrete test suite for Case1 and Case2

Figure 9 shows the concrete test suites generated.

Acknowledgments

The author would like to thank his supervisors, Professors John Grundy and John Hosking, for their guidance and contributions to the research. In addition, special thanks to Dr Partha Roop for his advice and comment. We would like to extend our gratitude to the Malaysian Ministry of Higher Education, Universiti Putra Malaysia, the University of Auckland and FRST for funding the research.

References

- [1] W. Hui, "Grammar-driven generation of domain-specific language tools," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* Portland, Oregon, USA: ACM, 2006.
- [2] J. Grundy and J. Hosking, "Supporting Generic Sketching-Based Input of Diagrams in a Domain-Specific Visual Language Meta-Tool," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 282-291.
- [3] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools (Microsoft .NET Development Series)*. Boston: Pearson Education, Inc., 2007.
- [4] B. Haugset and G. K. Hanssen, "Automated Acceptance Testing: A Literature Review and an

- Industrial Case Study," in *Conference on Agile, 2008. AGILE '08.*, 2008, pp. 27-38.
- [5] S. Morris and G. Spanoudakis, "UML: an evaluation of the visual syntax of the language," in *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, 2001, p. 10 pp.
- [6] A. Hartman, M. Katara, and S. Olvovsky, "Choosing a Test Modeling Language: A Survey," in *Hardware and Software, Verification and Testing*, 2007, pp. 204-218.
- [7] H. Kaindl, L. Constantine, O. Pastor, A. Sutcliffe, and D. Zowghi, "How to Combine Requirements Engineering and Interaction Design?," in *International Requirements Engineering, 2008. RE '08. 16th IEEE*, 2008, pp. 299-301.
- [8] A. J. S. Mills, "JUnit Testing Utility Tutorial," 2005, p. 6.
- [9] C. Poole, "NUnit Cookbook." vol. 2009, 2005.
- [10] B. Hasling, H. Goetz, and K. Beetz, "Model Based Testing of System Requirements using UML Use Case Models," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, 2008, pp. 367-376.
- [11] D. Arnold, J.-P. Corriveau, and W. Shi, "Scenario-Based Validation: Beyond the User Requirements Notation," in *21st Australian Software Engineering Conference (ASWEC 2010)* Auckland, New Zealand: Computer Society Press, 2010, p. 75.
- [12] P. Baker and C. Jervis, "Early UML Model Testing using TTCN-3 and the UML Testing Profile," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, 2007, pp. 47-54.
- [13] M. Burnett, A. Sheretov, R. Bing, and G. Rothermel, "Testing homogeneous spreadsheet grids with the "what you see is what you test" methodology," *Software Engineering, IEEE Transactions on*, vol. 28, pp. 576-594, 2002.
- [14] J. Romero-Mariona, "Secure and Usable Requirements Engineering," in *IEEE/ACM International Conference on Automated Software Engineering* Auckland, New Zealand: IEEE, 2009, p. 4.
- [15] J. Ernits, M. Kaaramees, K. Raiend, and A. Kull, "Requirements-driven model-based testing of the IP multimedia subsystem," in *Electronics Conference, 2008. BEC 2008. 11th International Biennial Baltic*, 2008, pp. 203-206.
- [16] C. Yuhong, J. Grundy, and J. Hosking, "Experiences integrating and scaling a performance test bed generator with an open source CASE tool," in *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, 2004, pp. 36-45.
- [17] B. Ngoc Bao and J. Ross, "DSL Bench: applying DSL in benchmark generation," in *Proceedings of the 1st workshop on Model Driven Development for Middleware (MODDM '06)* Melbourne, Australia: ACM, 2006.
- [18] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-driven development using UML 2.0: promises and pitfalls," *Computer*, vol. 39, pp. 59-66, 2006.
- [19] J. Grundy, Y. Cai, and A. Liu, "SoftArch/MTE: Generating Distributed System Test-beds from High-level Software Architecture Descriptions," *Automated Software Engineering*, vol. 12, pp. 5-39 pp., January, 2005 2005.
- [20] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. Fisher, II, and M. Main, "End-user software visualizations for fault localization," in *Proceedings of the 2003 ACM symposium on Software visualization* San Diego, California: ACM, 2003.
- [21] A. J. James, H. Mary Jean, and S. John, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering* Orlando, Florida: ACM, 2002.
- [22] H. Alan, K. Mika, and P. Amit, "Domain specific approaches to software test automation," 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* Dubrovnik, Croatia: ACM, 2007.
- [23] M. Sarma, P. V. R. Murthy, S. Jell, and A. Ulrich, "Model-based testing in industry: a case study with two MBT tools," in *Proceedings of the 5th Workshop on Automation of Software Test* Cape Town, South Africa: ACM, 2010.
- [24] A. Sinha and C. Smidts, "HOTTest: A model-based test design technique for enhanced testing of domain-specific applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, pp. 242-278, 2006.
- [25] R. Biddle, J. Noble, and E. Tempero, "Essential use cases and responsibility in object-oriented development," in *Proceedings of the twenty-fifth Australasian conference on Computer science - Volume 4* Melbourne, Victoria, Australia: Australian Computer Society, Inc., 2002.
- [26] L. L. Constantine and L. A. D. Lockwood, "Usage-centered engineering for Web applications," *Software, IEEE*, vol. 19, pp. 42-50, 2002.
- [27] M. Karaila and T. Systa, "Applying Template Meta-Programming Techniques for a Domain-Specific Visual Language--An Industrial Experience Report," in *29th International Conference on Software Engineering, 2007. ICSE 2007.*, Minneapolis, MN, USA, 2007, pp. 571-580.
- [28] C. Yuhong, G. John, and H. John, "Synthesizing client load models for performance engineering via web crawling," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* Atlanta, Georgia, USA: ACM, 2007.