chi+med
making medical devices safer

EPSRC Programme Grant EP/G059063/1

Public Paper no. 311

Integrating the PVSio-web Modelling and Prototyping Environment with Overture

Paolo Masci, Luis D. Couto, Peter G. Larsen & Paul Curzon

Masci, P., Couto, L. D., Larsen, P. G., & Curzon, P. (2015).
Integrating the PVSio-web modelling and prototyping environment with Overture.
Proceedings of the 13th Overture Workshop, 33–47.
(GRACE Technical report 2015-06, ISSN 1884-0760.)

PP release date: 24 May 2015

file: WP311.pdf

UCL UCLIC Swansea University Prifysgol Abertawe FITLAB Queen Mary University of London CITY UNIVERSITY LONDON EPSRC Engineering and Physical Sciences Research Council

Integrating the PVSio-web modelling and prototyping environment with Overture

Paolo Masci[1,⋆], Luis Diogo Couto[2], Peter Gorm Larsen[2], and Paul Curzon[1]

[1] School of Electronic Engineering and Computer Science
Queen Mary University of London, United Kingdom
{p.m.masci,p.curzon}@qmul.ac.uk

[2] Department of Engineering
Aarhus University, Denmark
{ldc,pgl}@eng.au.dk

Abstract. Tools are needed that overcome the barriers preventing development teams using formal verification technologies. We present our work integrating PVSio-web with the Overture development and analysis environment for VDM. PVSio-web is a graphical environment for modelling and prototyping interactive systems. Prototypes developed within PVSio-web can closely resemble the visual appearance and behaviour of a real system. The behaviour of the prototypes is entirely driven by executable formal models. These formal models can be generated automatically from Emucharts, graphical diagrams based on the Statechart notation. Emucharts conveniently hides aspects of the formal syntax that create barriers for developers and domain experts who are new to formal methods. Here, we present the implementation of a VDM-SL model generator for Emucharts. An example is presented based on a medical device. It demonstrates the benefits of using Emucharts to develop a formal model, how PVSio-web can be used to perform lightweight formal analysis, and how the developed VDM-SL model generator can be used to produce a model that can be further analysed within Overture.

Keywords: Prototyping, VDM-SL, PVSio-web.

1 Introduction

Formal verification technologies can help developers to discover design problems early in the development process of safety critical systems. These technologies, however, usually require significant mathematical sophistication, and many developers perceive this as a barrier that overweighs the advantages of using such tools.

PVSio-web [1,2] is a new research tool developed to ease the use of formal methods technologies when developing safety-critical *interactive* systems, i.e., ones that involve interaction between devices and human users. It provides a graphical environment that allows developers to rapidly generate interactive prototypes resembling the visual appearance and behaviour of the real system (see Figure 1). Underneath the interface, the

⋆ Corresponding author.

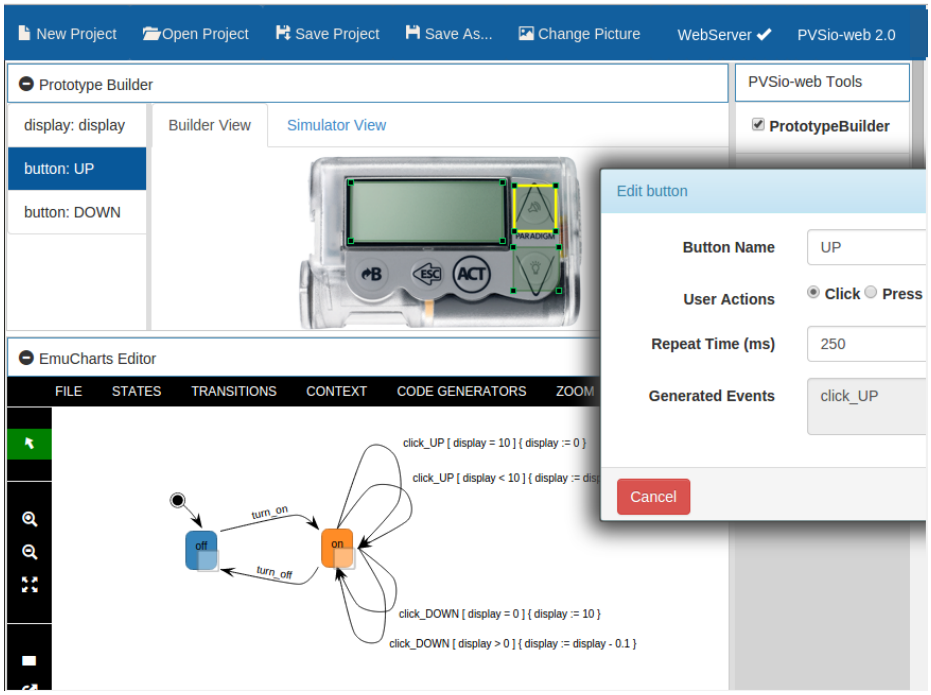

Fig. 1: Screenshot of the PVSio-web graphical environment while creating an interactive prototype based on a formal model.

tool uses advanced formal methods technologies for modelling and analysis. PVSio-web has been successfully used to demonstrate previously undetected design flaws in medical devices [3], and to clarify the causal relationships between user interface issues and software defects [4].

In its current implementation, PVSio-web builds on the PVS [5] theorem proving system for modelling and analysis. However, the architecture of PVSio-web is general, and allows one to link the environment with other formal methods tools.

We report on our work on integrating PVSio-web [1] with the Overture [6] development and analysis environment for VDM. This benefits both PVSio-web and Overture users. PVSio-web users gain direct access to an extensive set of tool features and case studies developed by the VDM community. Overture users gain the modelling and prototyping functionalities offered by PVSio-web, which enable: *validation of formal models* with domain specialists before starting a verification process; *demonstration of formal analysis results* to domain specialists in a way that is easy to comprehend; *lightweight formal analysis* of user interfaces based on user-centered design methods. Here, we focus on the integration of a core PVSio-web tool, the Emucharts editor, with Overture. Using the Emucharts editor, developers can specify the behaviour of a PVSio-web prototype using graphical diagrams, and automatically generate executable PVS models from these diagrams. We have successfully extended the Emuchars editor

to enable generation of VDM models. This basic integration already opens several exciting options, including automatic translation of VDM models from/to PVS, as well as means to explore the behaviour of VDM models using PVSio-web prototypes. The contributions are:

- A new PVSio-web extension for generating executable VDM specification language (VDM-SL) models.
- An example application based on a medical device. A formal model of the device is specified using a graphical Emucharts diagram that hides aspects of the formal syntax; then, a device prototype based on the Emucharts is generated that enables lightweight formal analysis; finally, a VDM-SL model is generated from the same Emucharts, enabling full formal analysis in Overture.

Related Work. VDMPad [7, 8] is a web-based integrated development environment for developing VDM-SL models. The tool provides: a textual model editor for viewing and editing models; a model animator for model debugging and testing. The tool supports the exploratory development of formal models, allowing lightweight formal analysis and permissive checking. In contrast to VDMPad, PVSio-web is specifically designed for modelling and analysis of interactive (human-computer) systems. Our tool thus offers functionalities for generating realistic interactive prototypes that can closely resemble the visual aspect and behaviour of a real system. The behaviour of these prototypes is based on formal models executed within the PVSio [9] animation environment of PVS. SCR [10] and B-Motion Studio [11] are also related work in that both tools provide a way to obtain graphical prototypes from formal models. Using SCR, one can formally specify the behaviour of a system, use visual front-ends for demonstrating the system behaviour based on the specifications, and use a group of formal methods tools for the analysis of system properties. With B-Motion Studio, one can create simple graphical visualisations based on Event-B models. SCR and B-Motion Studio are, however, not integrated with Overture. In addition, these tools lack specialised functionalities needed for the analysis of user interfaces (e.g., deployment of prototypes on mobile devices, and logging of user interactions).

Organisation. The remainder of the paper is organised as follows. We first overview the PVSio-web Emucharts Editor in Section 2. The core of the paper, illustrating a VDM-SL model generator for Emucharts, is presented in Section 3. We then give, in Section 4, a small example with a medical device. The example illustrates how Emucharts can be conveniently used to develop a formal model of the data entry system of the device (in subsection 4.2), how PVSio-web supports lightweight formal analysis (in subsection 4.3), and how the developed VDM-SL model generator can be used to produce an executable VDM-SL model that can be further analysed within Overture (in subsection 4.5). Finally, Section 5 provides a number of concluding remarks and indicates the future plans with this work.

2 Emucharts Editor

The PVSio-web Emucharts Editor is a tool for developing models of interactive systems. Models are specified using graphical diagrams called Emucharts, based on Statecharts [12]. Figure 1 shows a snapshot of the Emucharts Editor in use developing a

diagram specifying the behaviour of a medical device. Using the Emucharts Editor, developers can:

- Draw labelled boxes representing *states* of the system. State labels are strings representing the names of the different modes of the modelled system.
- Draw labelled arrows representing *transitions* between states. Transition labels are in the form `t [ cond ] { actions }`. The transition name (`t`) is a symbolic constant identifying the name of the modelled event. The transition condition (`cond`) is a Boolean expression defining the circumstances under which the transition is taken. The transition actions (`actions`) are expressions defining how the system state changes when the transition is taken.
- Define *variables* representing the structure of the system state. State variables are tuples: *(name, type, value, v0)*. Variable names are unique. Variable types can be basic types (e.g., **`bool`**, **`int`**, **`real`**), or user-defined types (e.g., *records*, *lists*). As in modern programming languages, a variable's value can be retrieved by referencing the variable name, and can be updated using assignment expressions (the assignment operator is :=). Each variable has an initial value, *v0*, given as the last element of the tuple.

In the Emucharts Editor, a virtual palette provides the essential elements for drawing the diagram (i.e., boxes and arrows), as well as tools for editing labels of diagram elements, and erasing elements from the diagram. Variables, constants, and functions are declared in a table called *context*, separately from the graphical diagram.

The Emucharts Editor was developed using the Model-View-Controller [13] design pattern, which creates a clear separation between the graphical front-end of the tool, and the logic for generating formal models. The editor, in fact, has two main components. The first is a Visual Editor, which handles both interactions with the user when drawing a diagram, and the look-and-feel of the graphical elements of the diagram. The second element is a Model Generator, which allows developers to translate visual diagrams into formal models.

In this work, we extend the Model Generator, and introduce a new module for producing executable VDM-SL models that can be imported and analysed within Overture.

3 The VDM-SL Model Generator

Our VDM-SL model generator is for Emucharts diagrams representing deterministic event-driven state machines. That is, the state machine has: a finite number of states, each representing a mode of the modelled system; a finite set of transitions, each modelling events that change the system state; and a single initial state, modelling the starting state of the state machine. The state machine can be in only one state at a time, and perform only one transition in each state for each possible input.

The rules for generating VDM-SL models from Emucharts diagrams are as follows, and illustrated in an example in the next section:

- A VDM-SL module is generated for each Emucharts diagram. The name of the module is the name of the Emucharts diagram.

- A VDM-SL state block `EmuchartState` is generated for specifying the state of the VDM-SL model. The record includes a field for each variable declared in the Emucharts diagram. The name and type of each field is the name and type of the variable from which the field has been generated. Two additional record fields, `current_state` and `previous_state`, are also automatically generated: the former represents the current machine state; the latter represents the previous machine state.
- A VDM-SL `mk_EmuchartState` record constructor is available to initialise the state of the VDM-SL model with the initial values of the variables declared in the Emucharts diagram.
- A VDM-SL enumerated type `MachineState` is generated for each Emucharts state. The enumerated type constants are the Emucharts state labels.
- A VDM-SL transition function of type `EmuchartState` $\rightarrow$ `EmuchartState` is generated for each unique transition name in the Emucharts diagram. The function argument models the current state of the VDM-SL model. The function return models the next state of the VDM-SL model after the execution of the transition function.
- The body of each VDM-SL transition function is a sequence of conditional *if-then-else* blocks. The Boolean expression used in each conditional block is the conjunction of two elements: the transition condition specified in the transition label; and a Boolean expression based on the current Emucharts state. The body of each conditional block is a series of modifier expressions (`mu(...)`) that update the current model state according to the transition actions specified in the diagram. The modifier expressions are chained to each other using the *let-in* construct.
- A VDM-SL permission function of type `EmuchartState` $\rightarrow$ `bool` is generated for each function of the VDM-SL model. The body of the permission function is the disjunction of the Boolean expressions used in the top-level *if-then-else* blocks that make up the body of the transition function.
- A VDM-SL operation is generated for each transition function.

4 Example

In this section we use PVSio-web to develop a device prototype that can be formally analysed. The aim is to demonstrate that:

- Emucharts diagrams conveniently hide the technical details of formal languages, and thus make formal verification technologies more accessible to non-experts of formal methods.
- PVSio-web enables rapid generation of a realistic prototype that allows developers to perform an early evaluation of the device, when a physical prototype of the device is not readily available.
- The VDM-SL model generator enables automatic generation of VDM-SL executable models that can be formally analysed within Overture.

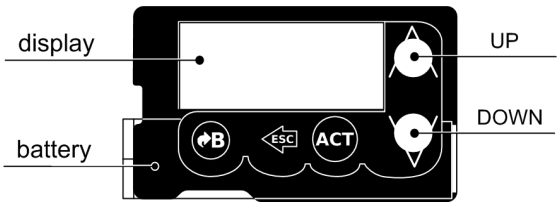

Fig. 2: Blueprint of an insulin pump with UP and DOWN buttons.

4.1 Description of the system

The considered system is an insulin pump used to treat type 1 diabetes. The device allows its user to specify therapy parameters such as the amount of insulin to be injected to keep the blood glucose level under control (bolus dose). The pump is battery-powered, and is turned on by inserting a battery in to the device.

Its data entry system consists of two buttons (UP and DOWN) and a display — a blueprint of the device is in Figure 2. Here, we focus on the behaviour of the device only for data entry of bolus doses. When entering a bolus dose, a click on the UP button increments the display value by 0•1. Similarly, a click on the DOWN button decrements the display value by 0•1. The maximum bolus dose is 10 units.

In the following sub-sections, we develop an Emucharts diagram that models the behaviour of this data entry system. The Emucharts diagram is used within PVSio-web to drive the behaviour of an interactive prototype based on a PVS model (further details about how these prototypes are generated can be found in [1, 2]). The same Emucharts is then used to generate an executable VDM-SL model that can then be further analysed within Overture.

4.2 Emucharts Diagram

An Emucharts diagram modelling the described behaviour of the device is shown in Figure 3. The diagram includes two states, *on* and *off*, modelling whether the device is powered on or off. The *off* state is the initial state. In the diagram, this is represented using a default initial transition that enters the *off* state.

A transition *turn_on* changes the device state from *off* to *on*. This models the action of inserting a battery into the device. Similarly, a transition *turn_off* changes the device state from *on* to *off*. This models the action of removing the battery from the device, or a depleted battery.

Two state transitions *click_UP* model the behaviour of UP button clicks. One transition models button clicks when the display value is less than 10. In this case, the display value is incremented by 0•1. The transition condition is therefore $display < 10$, and the transition action specifies the new value of the display using the assignment expression *display := display + 1*. The other transition is for handling the boundary case at 10. In this case, a click on the UP button resets the display value to 0.

Two other transitions *click_DOWN* model the behaviour of the DOWN button. One transition is for values above 0, and decrements the current display value by 0•1. The

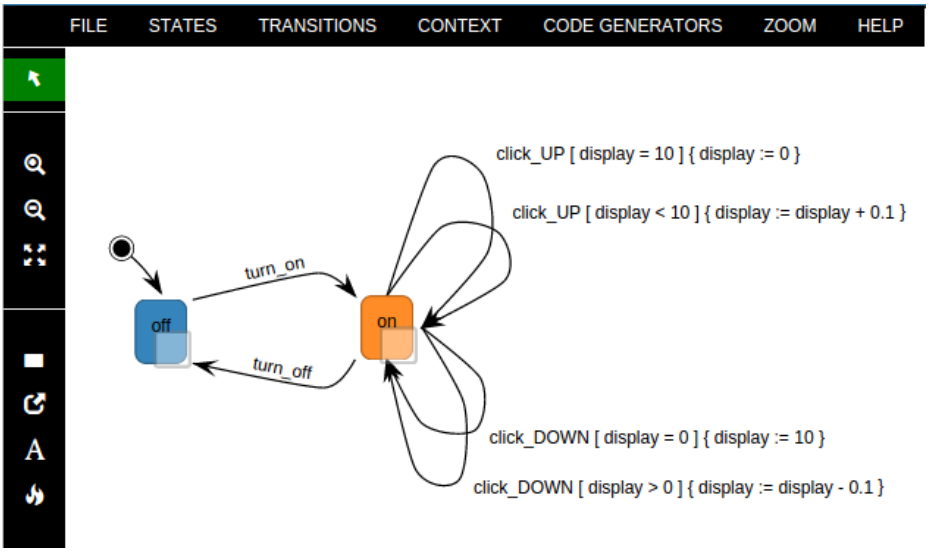

Fig. 3: Snapshot of the Emucharts Editor while drawing a diagram modelling the behaviour of the data entry system described in sub-section 4.1.

other transition is for the boundary case at 0. In this case, a click on the DOWN button changes the display value to 10 (this makes the behaviour of the DOWN button symmetric with respect to the UP button).

Finally, a variable *display* is declared in the Emucharts context for modelling the display value. The type of the variable is *real*, and the initial value is 0.

4.3 Generating and Analysing an Interactive Prototype

The behaviour modelled with the Emucharts diagram is now used as a basis to generate a realistic prototype that resembles the look and feel of the final product. This prototype enables lightweight formal analysis for early evaluation of the device behaviour.

The prototype is generated within PVSio-web using the Prototype Builder front-end. This is done by loading a realistic picture of the device in the tool, and creating interactive areas over the picture (see Figure 1). Three interactive areas are created in this case. The first is for the display, and is associated to the Emucharts variable `display`. Two more capture the user pressing the buttons in the picture of the device. These interactive areas translate the button presses into commands for animating the formal model associated with the Emucharts diagram. This formal model is automatically generated from the diagram, and executed within PVS [5] using its the native PVSio [9] animation environment.

Once the prototype is generated, one can explore the formal model by clicking buttons of the device, seeing the results of the interactions on the device display (see Figure 4). Using the prototype, a lightweight formal analysis can be performed before starting the full formal analysis. For example, one can perform an expert walkthrough [14]

Fig. 4: The insulin pump prototype executed within PVSio-web.

of the device. It is a usability inspection method performed by human-computer interaction specialists for identifying issues with the user interface of a system. By exploring the behaviour of the prototype, for example, the following conceptual issue can be easily identified with few exploratory input key sequences: when the display is 0 and the down button is pressed, the display value rolls over to 10. This behaviour is unsafe, as a single accidental button press while programming the bolus dose could lead to accidental overdoses [4, 15]. As a matter of fact, a real medical device on the market has been recalled because of this design issue [16, 17].

It is worth noting that the prototype has been generated without a full model of the system. Therefore, these prototypes can be generated at the early stages of device development, allowing developers to identify conceptual design issues in advance, and fix them before committing to potentially expensive design decisions.

4.4 Generating a VDM-SL Model

The newly developed VDM-SL model generator is now used to generate a VDM-SL model from the same Emucharts diagram used for the interactive prototype. The generated model can be imported within Overture for further formal analysis (type checking, analysis of proof obligations, generation of test cases, etc.). The steps illustrated in Section 3 are now illustrated for the diagram. The full VDM-SL model generated from the diagram is given in the Appendix.

The VDM-SL model generator creates the type definitions first. An enumerated type *MachineState* is generated that includes two enumerated constants, one for each state represented in the Emucharts diagram.

```
MachineState = <off> | <on>;
```

Listing 1.1: MachineState type

A state block *EmuchartState* is then generated that includes: a field *display* of type *real*, which models the variable defined in the Emucharts context; two fields *current_state* and *previous_state* of type *MachineState*, which store information about the current and previous active state of the state machine.

```
state EmuchartState of
  current_state: MachineState
  previous_state: MachineState
  display: real
```

Listing 1.2: VDM-SL model state

A function *init* is then generated that defines the initial model state. The initial value of the display is 0, the initial value of the current state is $<off>$, as specified in the Emucharts context.

```
init s == s = mk_EmuchartState(<off>, undefined, 0)
```

Listing 1.3: VDM-SL initial state

Functions representing transitions of the state machine are then generated. Transition functions with the same name are automatically merged into the body of a single VDM-SL function. For example, a single function *click_UP* is generated that models the two *click_UP* transitions specified in the Emucharts diagram.

The body of the generated VDM-SL function is, at the top level, a series of *if-then-else* statements. Each conditional statement is generated from a transition function included in the diagram. The Boolean expressions used in the conditional statement are based on both the transition conditions specified in the transition labels, and on the structure of the diagram (in particular, information about which state the transition is leaving from). For example, a Boolean expression generated for the *click_UP* function is *s.current_state* = $<on>$ *and s.display* $<$ *10*, as the arrow representing the transition leaves the *on* state, and the label of the transition includes a condition *display* $<$ *10*.

```
click_UP: EmuchartState -> EmuchartState
click_UP(s) ==
 if (s.current_state = <on>) and (s.display < 10) then ...
 elseif (s.current_state = <on>) and (s.display = 10) then ...
 else undefined
```

Listing 1.4: VDM-SL transition function (overall structure)

The body of each conditional block is then generated. Each block always starts with function *leave_state*. This is an auxiliary function that updates field `previous_state` of the VDM-SL model state with the label of the state that the transition leaves. Each block ends with another auxiliary function, *enter_into*, that updates `current_state`

with the label of the state that the transition enters. The actions specified in the Emucharts diagram are state updates, therefore they are translated using the VDM-SL *mu* operator. Consider transition *click_UP* relative to the case when the display is less than 10. The transition leaves and enters the same state (<on>), and the action specifies that the display value is incremented by 0.1 when the transition is executed.

```
... let new_s = leave_state(<on>, s) in let
        new_s = mu(new_s, display |-> s.display + 0.1 )
    in enter_into(<on>, new_s) ...
```

Listing 1.5: VDM-SL transition function (example state update)

Finally, permission functions are automatically generated by the VDM-SL model generator to restrict the domain of the VDM-SL transition function, and thus enable verification of pre- and post-conditions. For example, the permission function for *click_UP* returns the disjunction of all conditions used in the body of the *click_UP* function. This makes the domain of function *click_UP* explicit, and is used by the VDM interpreter to perform essential sanity checks related to how the function is used in the model.

```
per_click_UP: EmuchartState -> bool
per_click_UP(s) ==
  ((s.current_state = <on>) and (s.display < 10)) or
  ((s.current_state = <on>) and (s.display = 10));
```

Listing 1.6: VDM-SL permission function

4.5 Analysis in Overture

We now carry out an analysis of the generated VDM-SL model using two features of Overture: Proof Obligation Generation and Combinatorial Testing [18].

Overture generates Proof Obligations for a VDM model to ensure the internal consistency of the model. Example checks involve assessing the legal use of types and functions in the model. Besides validating core aspects of the semantics of the VDM-SL model, in our case the analysis is also useful for validating the correct implementation of the VDM-SL model generator. Applying the Proof Obligation Generator to the generated model yields four proof obligations, all of them ensuring legal application of the various state transition functions. For example, for the VDM-SL operation representing transition *turn on*, a proof obligation is generated (see Listing 1.7) to ensure that function `turn_on` is correctly used according to its permission. This proof obligation, as well as the others generated for this example, are trivially true, thus confirming that the generated model is well-formed.

```
pre_turn_on(EmuchartState) => pre_turn_on(EmuchartState)
```

Listing 1.7: Sample proof obligation to ensure correct use of functions.

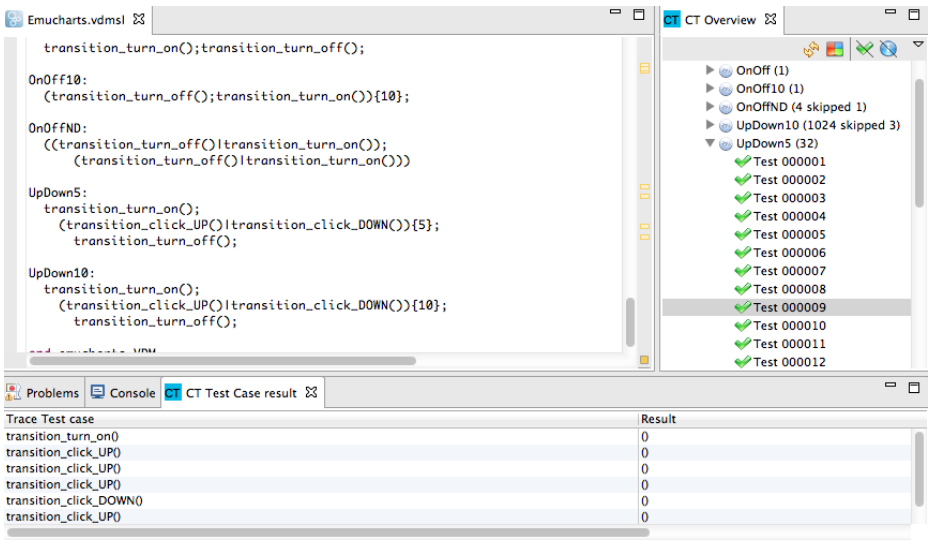

Fig. 5: The Overture Combinatorial Testing feature.

The Overture Combinatorial Testing tool generates test cases for the formal model from traces, allowing one to quickly specify and execute multiple usage scenarios. This can be extremely useful for validating the behaviour of the model against device prototypes or the final product. In Overture, test cases are specified using a trace notation that is akin to regular expressions. In Listing 1.8 we show an example trace for the model developed in the previous sections. It specifies that test cases are generated to explore the following use case: the device is turned on; then, the up and down buttons are randomly pressed 10 times; and then, the device is turned off. This trace expands to 1024 test cases that ensure all combinations of up and down are explored. The results for execution of this combinatorial test trace (and others) are shown in Figure 5. Most tests pass except for 13 which are inconclusive due to violated pre-conditions on the outside test calls such as attempting to turn off a device that is already off (note also that 4 tests are skipped because they share a sequence of calls with an inconclusive test and thus are pointless to execute).

```
traces

UpDown10:
  transition_turn_on();
  (transition_click_UP() | transition_click_DOWN()) { 10 };
  transition_turn_off();
```

Listing 1.8: Sample regular expression.

5 Concluding Remarks

We have illustrated the results achieved to date on integrating PVSio-web with the Overture platform. The integration allows developers to automatically generate VDM-SL models from a state machine description created using the PVSio-web Emucharts Editor. Formal models are thus created without the developer having a deep understanding of the VDM syntax. Also, because the Emucharts Editor incorporates model generators for other formal languages (PVS [5], MAL [19], PIM [20]), the developed tool can be conveniently used to translate VDM-SL state machine models from/to these other formal languages. Future work will extend this initial integration to give Overture and PVSio-web users even more benefits. For example, we plan to further extend the semantics supported by the model generator, e.g., to support diagrams specifying non-deterministic choices, and hierarchical state machines. Another extension relates to the ability of importing VDM models that are manually crafted by developers. This will ease reuse of models and examples already developed by the VDM community. Besides the Emucharts Editor, we plan to integrate two other components of PVSio-web with Overture. One component is the PVSio-web Prototype Builder, which handles the execution of prototypes developed within PVSio-web. The current implementation of this component uses PVSio as execution environment. We will link the Prototype Builder to the Overture interpreter. This will allow Overture users to send commands to the Overture interpreter by interacting with realistic prototypes resembling the real system being modelled, rather than by typing commands in the Overture interpreter console. The other component is the PVSio-web Co-Simulator, which enables integrated simulation of models developed using different modelling and analysis tools. We aim to explore how this component can be integrated with the VDM tool Crescendo [21] for collaborative modelling and simulation. Further work is also needed to determine how best to incorporate PVSio-web within future releases of Overture.

Acknowledgments. This work is part of CHI+MED (EPSRC grant EP/G059063/1).

References

1. Paolo Masci, Patrick Oladimeji, Yi Zhang, Paul Jones, Paul Curzon, and Harold Thimbleby. PVSio-web 2.0: Joining PVS to Human-Computer Interaction. In *27th International Conference on Computer Aided Verification (CAV2015)*. Springer, 2015. Tool and application examples available at `http://www.pvsioweb.org`.
2. Patrick Oladimeji, Paolo Masci, Paul Curzon, and Harold Thimbleby. PVSio-web: A tool for rapid prototyping device user interfaces in PVS. In *5th International Workshop on Formal Methods for Interactive Systems (FMIS2013)*, 2013.
3. Paolo Masci, Yi Zhang, Paul Jones, Paul Curzon, and Harold Thimbleby. Formal Verification of Medical Device User Interfaces Using PVS. In *ETAPS/FASE2014, 17th International Conference on Fundamental Approaches to Software Engineering*. Springer-Verlag, 2014.
4. Paolo Masci, Patrick Oladimeji, Paul Curzon, and Harold Thimbleby. Tool demo: Using PVSio-web to demonstrate software issues in medical user interfaces. In *4th International Symposium on Foundations of Healthcare Information Engineering and Systems (FHIES2014)*, 2014.
5. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.

6. Peter G. Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The overture initiative integrating tools for VDM. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–6, 2010.
7. Tomohiro Oda and Keijiro Araki. Overview of VDMPad: An Interactive Tool for Formal Specification with VDM. In *Proc. of International Conference on Advanced Software Engineering and Information Systems (ICASEIS)*, 2013.
8. Tomohiro Oda, Keijiro Araki, and Peter G. Larsen. VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In *To appear in Proc. of FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2015.
9. Cesar Muñoz. Rapid prototyping in PVS. Technical Report NIA Report No. 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, 2003.
10. Constance Heitmeyer, James Kirby, Bruce Labaw, and Ramesh Bharadwaj. SCR: A toolset for specifying and analyzing software requirements. In *Computer Aided Verification*, pages 526–531. Springer, 1998.
11. Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. Visualising Event-B Models with B-Motion Studio. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, pages 202–204. Springer-Verlag, November 2009.
12. David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
13. Glenn E. Krasner and Stephen T. Pope. A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.
14. Jakob Nielsen. Usability inspection methods. In *Conference companion on Human factors in computing systems*, pages 413–414. ACM, 1994.
15. Abigail Cauchi, Andy Gimblett, Harold Thimbleby, Paul Curzon, and Paolo Masci. Safer 5-key number entry user interfaces using differential formal analysis. In *BCS-HCI '12 Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers, BCS-HCI 2012, 12-14 September 2012, Birmingham, UK*, pages 29–38. British Computer Society, 2012.
16. Medtronic. Important medical device safety information regarding the safe use of the Medtronic insulin pump. http://www.medtronicdiabetes.com/res/img/pdfs/Insulin-Delivery-Through-Main-Menu-Button-Keypad_US-Customer-Letter.pdf, 13 March 2014.
17. US Food and Drug Administration (FDA). Class 2 Recall Medtronic MiniMed Paradigm REALTime and Paradigm REALTime Revel CGM System and MiniMed 530G System. Manufacturer and User Facility Device Experience Database (MAUDE), Recall Event ID 68277, 22 August 2014. http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRES/res.cfm?id=127259.
18. Peter G. Larsen, Kenneth Lausdahl, and Nick Battle. Combinatorial testing for VDM. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 278–285. IEEE, 2010.
19. José C. Campos and Michael D. Harrison. Interaction engineering using the ivy tool. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS09)*, pages 35–44. ACM, 2009.
20. Judy Bowen and Steve Reeves. Modelling safety properties of interactive medical systems. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS13, pages 91–100. ACM, 2013.
21. Peter G. Larsen, Carl Gamble, Kenneth Pierce, Augusto Ribeiro, and Kenneth Lausdahl. Support for Co-modelling and Co-simulation: The Crescendo Tool. In *Collaborative Design for Embedded Systems*, pages 97–114. Springer, 2014.

Appendix: Full VDM-SL Model

```
module emucharts_MedtronicMinimed530G_VDM
exports all
definitions

types
  -- machine states
  MachineState = <off> | <on>;
  -- emuchart state
  state EmuchartState of
    current_state: MachineState
    previous_state: MachineState
    display: real
  -- initial state
  init s == s = mk_EmuchartState(<off>, undefined, 0) end

functions
  -- utility functions
  enter_into: MachineState * EmuchartState -> EmuchartState
  enter_into(ms, s) == mu(s, current_state |-> ms );
  leave_state: MachineState * EmuchartState -> EmuchartState
  leave_state(ms, s) == mu(s, previous_state |-> ms );

  -- transition functions
  per_turn_on: EmuchartState -> bool
  per_turn_on(s) == ((s.current_state = <off>));
  turn_on: EmuchartState -> EmuchartState
  turn_on(s) ==
    if (s.current_state = <off>)
    then let new_s = leave_state(<off>, s)
         in enter_into(<on>, new_s)
    else undefined
  pre per_turn_on(s);

  per_turn_off: EmuchartState -> bool
  per_turn_off(s) == ((s.current_state = <on>));
  turn_off: EmuchartState -> EmuchartState
  turn_off(s) ==
    if (s.current_state = <on>)
    then let new_s = leave_state(<on>, s)
         in enter_into(<off>, new_s)
    else undefined
  pre per_turn_off(s);

  per_click_DOWN: EmuchartState -> bool
  per_click_DOWN(s) == ((s.current_state = <on>) and (s.
      display > 0)) or ((s.current_state = <on>) and (s.
      display = 0));
  click_DOWN: EmuchartState -> EmuchartState
```

```
  click_DOWN(s) ==
    if (s.current_state = <on>) and (s.display > 0)
    then let new_s = leave_state(<on>, s) in let
             new_s = mu(new_s, display |-> s.display - 0.1 )
          in enter_into(<on>, new_s)
    elseif (s.current_state = <on>) and (s.display = 0)
    then let new_s = leave_state(<on>, s) in let
             new_s = mu(new_s, display |-> 10 )
          in enter_into(<on>, new_s)
    else undefined
  pre per_click_DOWN(s);

  per_click_UP: EmuchartState -> bool
  per_click_UP(s) == ((s.current_state = <on>) and (s.display
      < 10)) or ((s.current_state = <on>) and (s.display=10));
  click_UP: EmuchartState -> EmuchartState
  click_UP(s) ==
    if (s.current_state = <on>) and (s.display < 10)
    then let new_s = leave_state(<on>, s) in let
             new_s = mu(new_s, display |-> s.display + 0.1 )
          in enter_into(<on>, new_s)
    elseif (s.current_state = <on>) and (s.display = 10)
    then let new_s = leave_state(<on>, s) in let
             new_s = mu(new_s, display |-> 0 )
          in enter_into(<on>, new_s)
    else undefined
  pre per_click_UP(s);

operations
  transition_turn_on: () ==> ()
  transition_turn_on() == EmuchartState := turn_on(
      EmuchartState)
  pre pre_turn_on(EmuchartState);

  transition_turn_off: () ==> ()
  transition_turn_off() == EmuchartState := turn_off(
      EmuchartState)
  pre pre_turn_off(EmuchartState);

  transition_click_DOWN: () ==> ()
  transition_click_DOWN() == EmuchartState := click_DOWN(
      EmuchartState)
  pre pre_click_DOWN(EmuchartState);

  transition_click_UP: () ==> ()
  transition_click_UP() == EmuchartState := click_UP(
      EmuchartState)
  pre pre_click_UP(EmuchartState);
end emucharts_MedtronicMinimed530G_VDM
```