Extending a User Interface Prototyping Tool with Automatic MISRA C Code Generation

Gioacchino Mauro
Department of Information Engineering, University of Pisa, Pisa, Italy
giocchi27@gmail.com

Harold Thimbleby
Swansea University — Prifysgol Abertawe, Swansea/Abertawe, UK
harold@thimbleby.net

Andrea Domenici Cinzia Bernardeschi
Department of Information Engineering, University of Pisa, Pisa, Italy
{cinzia.bernardeschi,andrea.domenici}@unipi.it

We are concerned with systems, particularly safety-critical systems, that involve interaction between users and devices, such as the user interface of medical devices. We therefore developed a MISRA C code generator for formal models expressed in the PVSio-web prototyping toolkit. PVSio-web allows developers to rapidly generate realistic interactive prototypes for verifying usability and safety requirements in human-machine interfaces. The visual appearance of the prototypes is based on a picture of a physical device, and the behaviour of the prototype is defined by an executable formal model. Our approach transforms the PVSio-web prototyping tool into a model-based engineering toolkit that, starting from a formally verified user interface design model, will produce MISRA C code that can be compiled and linked into a final product. An initial validation of our tool is presented for the data entry system of an actual medical device.

1 Introduction

Formal methods are important for developing and understanding safe and secure systems. The PVSio-web framework [20, 21, 22, 26] allows developers to use formal methods in a friendly and appealing way as it provides realistic animations and is integrated with a graphical editor for the Emucharts language [23]. (Emucharts is a state machine formalism with guards and actions associated with transitions; it is explained further in Sect. 3.4 below.)

PVSio-web uses the formal modelling language of the Prototype Verification System (PVS) [27], including the PVSio extension [24]. PVS is an industrial-strength theorem proving system that allows formal verification of safety and reliability properties of hardware and software systems [5, 31]. Although PVS itself is very effective, it is not widely used for model-based development and analysis of user interfaces, as the tool has a steep learning curve. PVSio-web softens this learning curve, making the tool more user-friendly and accessible, providing developers with a graphical modelling environment, and a toolbox for developing realistic visual prototypes of user interfaces.

The applications of embedded software in safety-critical applications increase continuously. Taking this into account, together with the requirements to reduce time and overall production costs, automatic code generation plays an essential role. Automatic code generation guarantees a smooth conversion

C. Dubois, P. Masci, D. Méry (Eds.): F-IDE 2016
EPTCS 240, 2017, pp. 53–66, doi:10.4204/EPTCS.240.4

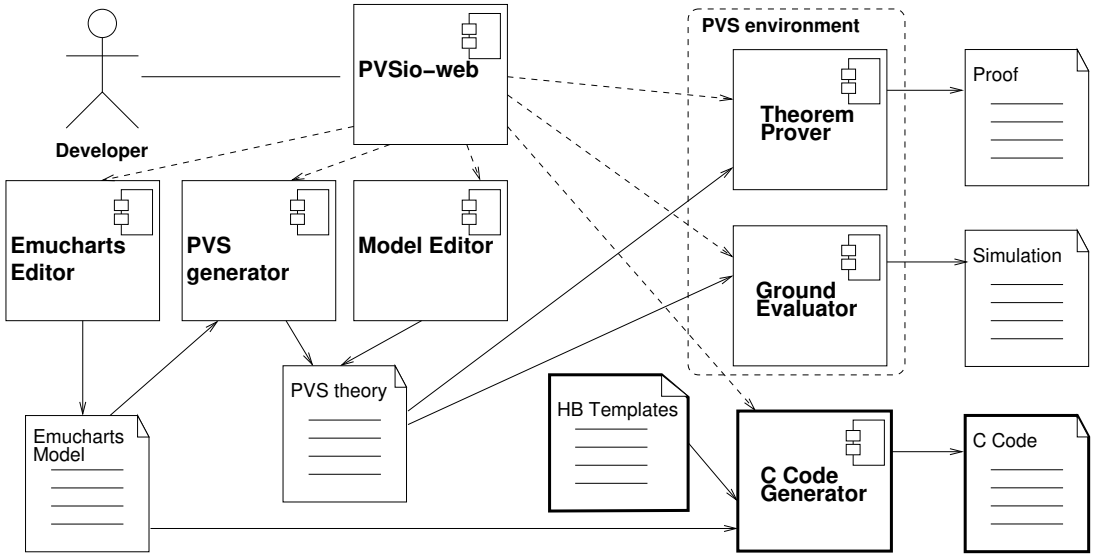

Figure 1: C code generation in the PVSio-web development process.

from model to code and reduces the debugging and testing required for source code, provided that the correctness of code generation and of the high-level model have been verified.

We therefore present an extension to PVSio-web that generates C code. Specifically, our extension generates MISRA C [1], a safety-oriented subset of C developed by the Motor Industry Software Reliability Association (MISRA). MISRA C is commonly used in safety-critical subsystems, such as car braking in automotive systems.

With this new extension, formal PVS specifications generated from Emucharts diagrams are automatically converted to C, significantly shortening project development time. Because of the approach, the semantics of generated C code is equivalent to the formal models, and therefore the code retains the reliability and safety properties formally verified for the PVS model.

In summary, our main contribution is an approach to software development that integrates logic- and state machine-based formal modelling, validation by simulation, and automatic implementation by generating production code to be run on the actual system hardware, all based on an industrial-strength formal methods toolkit.

2 Related work

Model-based approaches are commonly used in the field of human computer interaction, for example [13]. Most approaches are focused on describing user interfaces and their implementations at various levels of abstraction. Developers of user interfaces for interactive systems also have to address heterogeneity and adaptation to the context of use. For example, in [29], a model-based declarative language for the design of interactive applications based on Web services in ubiquitous environments was presented. In contrast to these familiar approaches, the present work proposes a framework enabling a formal verification of user interaction. The framework is meant for safety analysis of safety-critical devices and not with user interface design issues as discussed in [29],

Similarly to our approach, formal models were used in [6] to describe functionality and component interactions, where they were combined with user interface models in order to get the entire model of the

system. Moreover, an Android emulator application was generated, using Java and XML technologies. Presentation models and presentation interaction models were used in [9] to model interactive software systems; these models were shown to be usable with a formal specification of the system functionality. In [7] the same formalisms were used to model user manuals of modal medical devices, proving that the user manual may be not always consistent with actual device behaviour.

In [17], model checking was used to model and prove properties of specifications of interactive systems so that possibly unexpected consequences of interface mode changes can be checked early in the design process. In [19], the complementary role of model checking and theorem proving in the analysis of interactive devices was considered. Recent work [16] explored the paths that a user will take in interacting with medical devices for the analysis of properties of the behaviour of safety-critical devices. A model-checking approach has also been used to analyse hardware behaviour [4].

A discussion of production code generation in model-based development can be found in [11]. Many papers deal with specific code generators, for example TargetLink [3]. Code generators specifically designed for medical systems are described in [2] and [28].

3 PVS, model-driven development and Emucharts

This section provides background information on the PVSio-web framework and its relationship to model-driven development.

3.1 PVS, the Prototype Verification System

The PVS is an interactive theorem prover for a typed higher-order logic language, providing an extensive set of inference rules based on the sequent calculus [30]. Its PVSio extension is a ground evaluator that can compute the results of ground function applications, that is PVS expressions consisting of a function name applied to variable-free arguments. PVS functions are purely declarative definitions of mathematical mappings, without any procedural information on how to compute them, but the PVSio package can derive and execute an algorithm to evaluate a ground function application, turning it into a procedure call. The PVSio package also provides functions with side effects, such as input and output, which do not interfere with the semantics of a theory.

A system is modelled in PVS as a *theory*, a collection of logical statements and definitions about the structural and behavioural aspects of the system. The system's required properties are expressed as theorems to be verified with the PVS theorem prover. If the behavioural aspects are expressed as functions, the system can also be simulated with the PVSio extension. The same logical model can then be used both for verification and simulation.

3.2 Model-driven development

Model-driven development (MDD) is based on creating an executable system model by assembling functional blocks. An executable model makes it possible both to simulate the system and to generate production software to control it. Together with the naturalness of the graphic language of functional blocks, these features make MDD very attractive to developers. However, this approach has two limits: first, functional blocks lend themselves to building design models, but not specification ones; and secondly, formal verification of a block-based model is tedious, and in fact it is uncommon in industrial practice.

A formal approach can be used to create both specification and design models and intrinsically lends itself to rigorous verification of system properties. In particular, logic specification languages,

such as PVS, are supported by automatic or interactive theorem provers used by developers to check if system requirements, expressed as logical formulas, are implied by a system's description expressed in a logic theory. However, formal methods require expertise in languages and methods that are not widely known in the wider developer community. Further, most formal languages abstract from the familiar procedure-oriented computation model of popular programming languages, making it harder to generate executable software.

It is then desirable to have tools and methods providing developers with the features of both approaches. The present work is part of a research effort aimed at this goal. With the PVSio-web framework, a developer can build a model in a graphical state-machine language or a logic language, or both (Sect. 3). The graphical model is translated into the logic language automatically, and the resulting translation is both verifiable and executable using the PVSio ground evaluator, which acts as an interpreter for the PVS language. The PVSio-web framework thus provides features of the formal approach: A formal specification language and a verification tool, and features of MDD, thus providing a full graphical modelling language and a simulation engine. A translator from Emucharts to C makes it possible to generate code from a state machine-based model that can be validated by simulation and verified by theorem proving. The other important feature of MDD — generation of production code capable to be run on the actual system hardware — is a key contribution of this paper.

3.3 PVSio-web

The PVSio-web framework is a set of tools, co-ordinated by a web-based interface, for prototyping and simulation of interactive devices. Its main components are, besides PVS with its PVSio extension: (*i*) the **Prototype Builder**, a graphical tool used to choose a picture of an existing or anticipated device's front panel and to associate PVS functions with active areas of the picture representing device inputs (e.g., buttons or keys) and outputs (e.g., alphanumeric displays or lights); (*ii*) the **Model Editor**, a textual interface to write PVS code; (*iii*) the **Emucharts Editor**, a graphical tool to draw Emucharts state machine diagrams; (*iv*) a **Simulation Environment**; and (*v*) **Code Generators** for PVS and other formal languages (currently Presentation Interaction Models [8], Modal Action Logic [15], and Vienna Development Method [12]) — and for MISRA C, as presented in this paper.

PVSio-web can be used to prototype a new device interface, or to create a reverse-engineered model of an existing one. In either case, a developer creates formal descriptions of the device's responses to user actions, using the model and Emucharts editors, and associates these descriptions with the active areas of the simulated interface, using the prototype builder. In the simulation environment, the developer, or a domain expert or a potential user, interacts with the prototype clicking on the input widgets. These actions are translated to PVS function calls executed by the PVSio interpreter.

3.4 Emucharts

An Emucharts diagram is the representation of an extended state machine in the form of a directed graph composed of labelled *nodes* and *transitions*. Transitions are labelled with triples of the form *trigger*[*guard*]{*action*}, where *trigger* is the name of an event, *guard* is an enabling Boolean expression, and *action* is a set of assignments to typed variables declared in the state machine's *context*. The default guard is the *true* value and the default action is a no-operation. The *state* of the machine is defined by the current node and the current values of the context variables.

The code generator for PVS produces a theory containing functions that define the state machine behaviour on the occurrence of trigger events. Since an Emucharts diagram usually represents a device

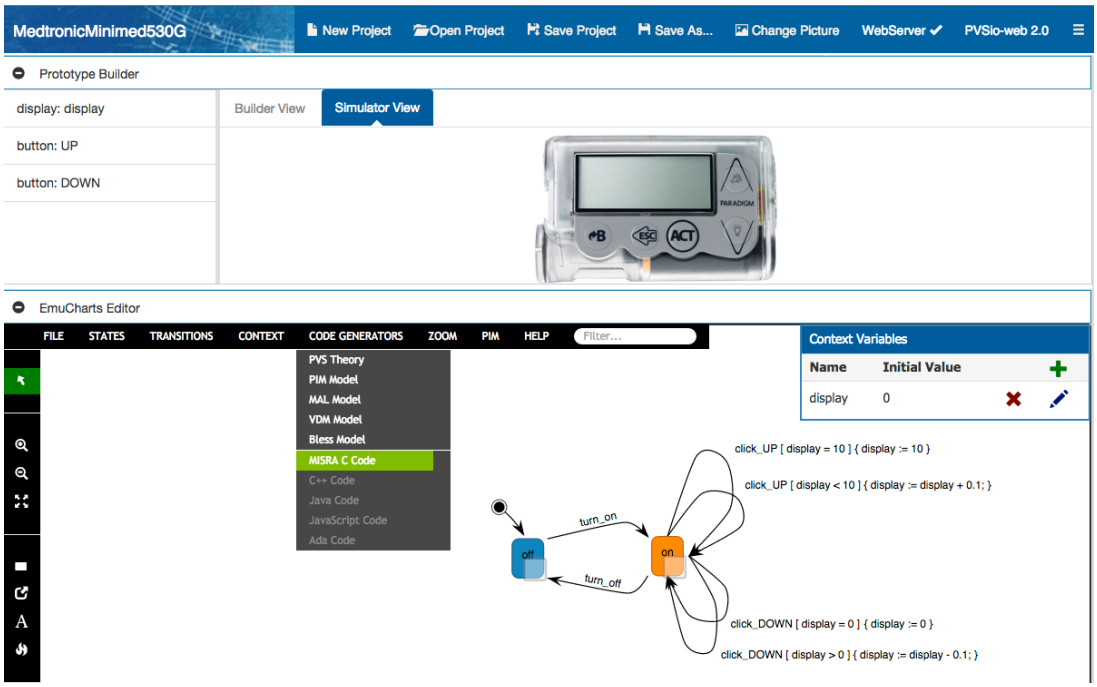

Figure 2: The PVSio-web user interface with the Prototype Builder and Emucharts Editor frames.

response to user actions, such events represent user actions, such as pressing a button on a control panel. During simulation on a PC, a user click on an active area of the device picture causes the simulator to generate a function application expression that is passed to the PVSio ground evaluator.

4 From Emucharts to safe C

The aim of programming code generation in the PVSio-web framework is producing a module that implements the user interface of a device, which can be compiled and linked into the device software without any particular assumptions on its architecture. In this way, the user interface module can be used without forcing design choices on the rest of the software. In our approach, the generated module contains a set of C functions. The main ones are, for each Emucharts trigger: (*i*) a *permission* function, to check if the trigger event is *permitted*, i.e., whether it is associated with any transition from the current state, and (*ii*) a *transition* function that, according to the current state, updates it, provided that the guard condition of an outgoing transition holds. The code includes logically redundant tests (*assert* macros) to improve robustness.

To generate production-quality code fit for safety-critical applications we adopt MISRA guidelines. The MISRA guidelines for the C language, originally conceived for the automotive industry, enforce programming practices to improve maintainability and portability and, above all, to reduce the risk of malfunction due to implementation- or platform-dependent aspects of the C language. For instance, there are rules that bar the use of constructs such as `goto`, and rules requiring that numeric literals be suffixed to indicate their type explicitly. The generated code currently complies with the first version of the 1998 MISRA C guidelines.

⟨ *headerfile* ⟩ ::= ⟨ *preprocessor_directives* ⟩
[⟨ *constant_definitions* ⟩]
⟨ *typedef_definitions* ⟩
⟨ *state_labels_enum* ⟩
⟨ *state_structure* ⟩
⟨ *utility_functions* ⟩
⟨ *init_function* ⟩
⟨ *permission_functions* ⟩
⟨ *transition_functions* ⟩

Table 1: Structure of a header file. Non-terminal symbols are enclosed between angle brackets; square brackets enclose optional symbols.

4.1 Code generation

Our MISRA C code generator was implemented in JavaScript using Handlebars [14], a macro-expansion tool for web applications. A Handlebars template is a piece of text containing "Handlebar expressions," which refer to elements of the surrounding context, typically an HTML document. A Handlebars expression specifies a character string as a function of context elements, which is compiled into a JavaScript function that returns the template text with the substitutions computed by the Handlebars expressions. For example, a template fragment for a C preprocessor `#include` directive is `#include "{{filename}}.h"`, where the Handlebars expression `{{filename}}` contains the `filename` parameter that will be replaced by the actual name of the file to be included.

The code generator produces a header file, an implementation file, a makefile, a simple test driver file, and a documentation manual.

The structure of the header file is defined by the grammar in Table 1. The header file contains, among other items, the declarations (*typedef_definitions* in the grammar) for types with explicit representation of size and sign, e.g., `UC_8`, for *eight-bit unsigned char*, the declaration (*state_labels_enum*) for an enumeration type defining the node labels, and the declaration (*state_structure*) for the *state* structure type representing the state of the Emucharts model. This structure contains one *context* field for each variable defined in the Emucharts context, and two more fields (*curr_node* and *prev_node*) contain the labels of the current and the previous node.

The declarations are followed by the function prototypes of the two utility functions *enter* and *leave*, the *init* function, and, for each trigger, one permission and one transition function. The functions receive a pointer to a structure of type *state* passed by a calling program. The *enter* and *leave* functions, called by the *init* and transition functions, update the *curr_node* and *prev_node* fields, respectively, with the target and source node label of the executed transition. The *leave* function has been introduced to allow future versions to implement checkpointing algorithms. The *init* function initialises the state's context fields with the values of the context variables specified in the Emucharts diagram, and the *curr_node* field with the label of the initial node. As mentioned above, each permission function checks if the current node has a transition labelled by the respective event. Then, the matching transition function chooses among the transitions triggered by that event, according to the respective guards (assumed to be mutually exclusive).

The implementation file contains the function definitions. For example, consider the Emucharts diagram of the data entry system of the Medtronic MiniMed 530G System shown in Figure 3. The diagram has a context variable *display* of type *double* represented on 64 bits, which holds the value shown on the device's display. The node labels and the *state* type are defined as

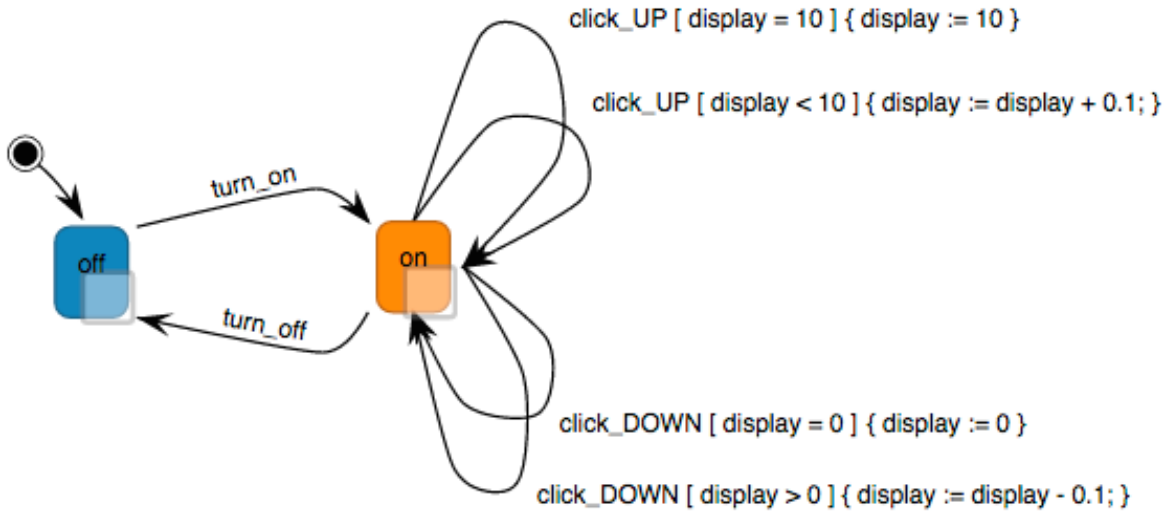

Figure 3: **Emucharts diagram for the Medtronic MiniMed 530G data entry system.**

```
typedef enum { off, on } node_label;
typedef struct {
    D_64 display;
    node_label curr_node;
    node_label prev_node; } state;
```

The code for the permission function associated with the *click_UP* trigger is

```
UC_8 per_click_UP(const state* st) {
    if (st->current_state ==  on) {
        return true;
    }
    return false;
}
```

where the return type UC_8 (eight-bit unsigned character) is used to represent the Boolean type. The transition function is

```
state click_UP(state* st) {
    assert(st->current_state ==  on);
    assert(st->display < 10 || st->display == 10);
    if (st->display < 10 && st->current_state == on) {
        leave(on, st);
        st->display = st->display + 0.1f;
        enter(on, st);
        assert(st->current_state == on);
        return *st;
    }
    if (st->display == 10 && st->current_state == on) {
        leave(on, st);
        st->display = 10.0f;
        enter(on, st);
```

```
        assert(st->current_state == on);
        return *st;
    }
    return *st;
}
```

A proof of the correctness of this translation schema is shown in Appendix A.

5 Case study

The Alaris GP, made by Becton Dickinson and Company, was used as a case study for the MISRA C code generator.

This volumetric infusion pump is a medical device used for controlled automatic delivery of fluid medication or blood transfusion to patients, with an infusion rate range between 1 ml/h and 1200 ml/h. It has a monochrome dot matrix display with three significant digits, and has 14 buttons for operating the device (see Figure 4). The pump has a rather complex user interface, with different modes of operation and ways of entering data, including the possibility of choosing from a list of preloaded treatments. For simplicity, in this paper only the essential part of the data entry interface, concerning numerical input and display, is considered.

Numerical input is done through the chevrons buttons: upward and downward chevrons increase and decrease, respectively, the displayed value. The amount by which the value is increased or decreased depends on whether a single or double chevron is pressed, and on the current displayed value. More precisely, the displayed value is changed as follows: (*i*) If the displayed value is below 100, the value changes by 0.1 units for a single chevron, and steps up or down to the next decade for a double chevron (e.g., from 9.1 to 10.0); (*ii*) if the displayed value is between 100 and 1,000, the value changes by 1 unit for a single chevron, and steps up or down to a value equal to the next hundred plus the decade of the displayed value for a double chevron (e.g., from 310 or 315 to 410); (*iii*) if the displayed value is 1,000 or above, the value changes by 10 units for a single chevron, and steps up or down to a value equal to the next hundred for a double chevron (e.g., from 1,010 or 1,080 to 1,100).

The Emucharts diagram for the numeric data entry is shown in Fig. 5. Triggers *click_alaris_up* and *click_alaris_dn* represent clicks on the upward and downward single-chevron buttons, respectively, and triggers *click_alaris_UP* and *click_alaris_DN* represent clicks on the double-chevron ones. For each event, combinations of guards and actions specify the rules described above.

The PVS code generator translates the diagram into an executable logic theory, and the C code generator produces permission and transition functions for each trigger, as explained previously.

5.1 Mobile applications

The PVSio-web framework uses a standard web interface to integrate its tools: this approach offers a uniform interface that a developer can access with any web browser.

Our framework has been extended by providing the possibility to run simulations on a mobile device. Smartphones and tablets improve usability and help make user interaction similar to actual device operation. For example, mobile devices could be used in a hospital environment to train medical personnel and patients.

An interactive device can be simulated using the C source code produced by the PVSio-web generator, compiled and linked with a mobile device-specific application. For example, the code for the user interface

Figure 4: Front panel of the Alaris GP infusion pump.

of the Alaris infusion pump has been ported to the Android [10] platform using the Android NDK [25] toolset, which can embed C code in a Java project, relying on the Java Native Interface (JNI) [18].

6 Conclusions

We presented the implementation of our MISRA C code generator for the PVSio-web prototyping toolkit. Automatic code generation significantly reduces project development time. Our approach eliminates a human-performed step in the development process: user interface software engineers no longer need to convert the design specifications into executable target code.

Our tool improves the development of safe and dependable user interfaces, as it greatly facilitates using formal methods easily and reliably with real UIs, which we demonstrated with the medical device examples in this paper.

Current and future directions include improving this initial integration with other features of C, still conformant to MISRA C under the most recent 2012 rules. We plan to develop code generators for programming languages such as C++, Java and ADA.

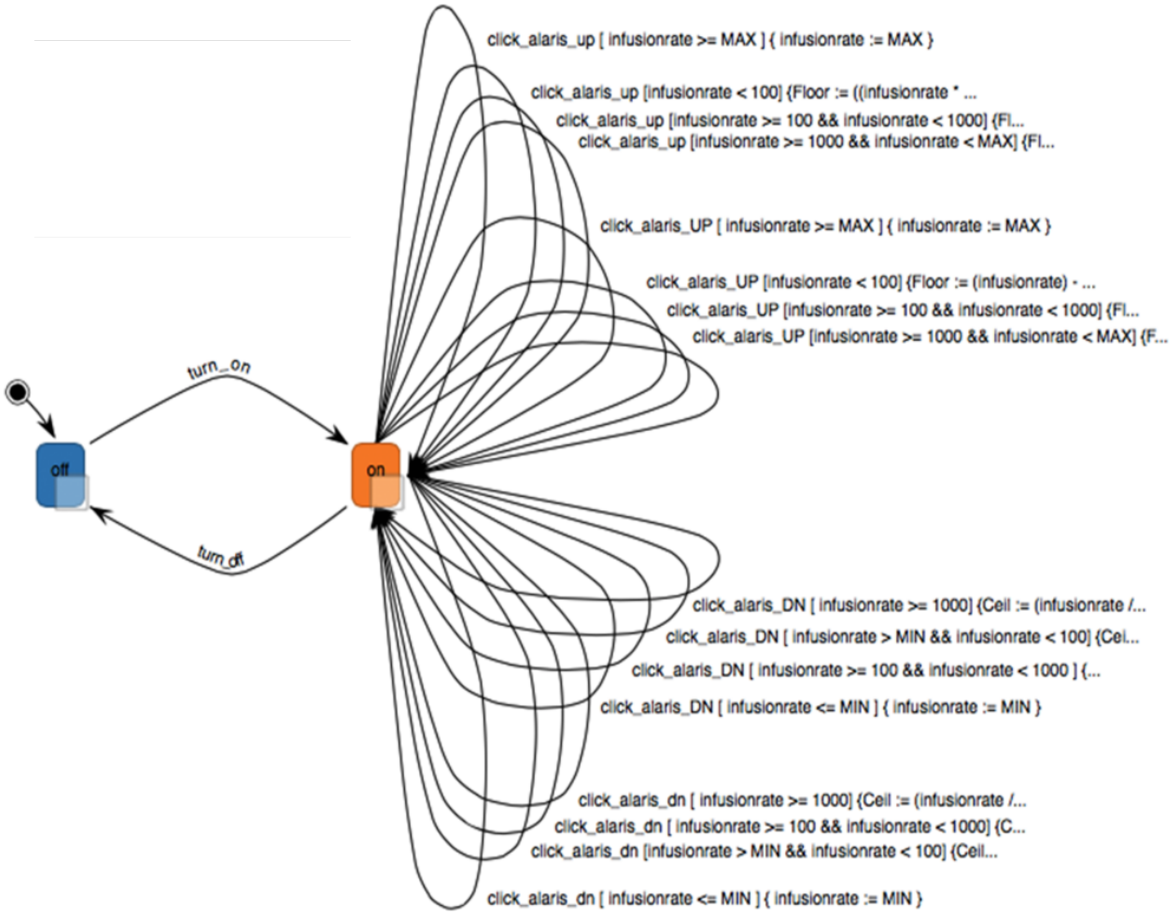

Figure 5: Emucharts diagram for numeric data entry.

Acknowledgements

This work was partially supported by the PRA 2016 project “Analysis of Sensory Data: from Traditional Sensors to Social Sensors” funded by the University of Pisa.

References

[1] Motor Industry Software Reliability Association (1998): *Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association.

[2] Ayan Banerjee & Sandeep K. S. Gupta (2014): *Model Based Code Generation for Medical Cyber Physical Systems*. In: *1st Workshop on Mobile Medical Applications (MMA ’14)*, pp. 22–27, doi:10.1145/2676431.2676646.

[3] M. Beine, R. Otterbach & M. Jungmann (2004): *Development of safety-critical software using automatic code generation*. Technical Report, SAE Technical Papers, doi:10.4271/2004-01-0708.

[4] C. Bernardeschi, L. Cassano, A. Domenici & L. Sterpone (2013): *Unexcitability Analysis of SEUs Affecting the Routing Structure of SRAM-based FPGAs*. In: *Proc. of the 23rd ACM Great Lakes Symposium on VLSI*, GLSVLSI ’13, pp. 7–12, doi:10.1145/2483028.2483050.

[5] Cinzia Bernardeschi, Paolo Masci & Holger Pfeifer (2008): *Early Prototyping of Wireless Sensor Network Algorithms in PVS*, pp. 346–359. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-87698-4_29.

[6] J. Bowen & A. Hinze (2011): *Supporting Mobile Application Development with Model-Driven Emulation*. In: *Formal Methods for Interactive Systems 2011*, *Electr. Comm. EASST* 45, doi:10.14279/tuj.eceasst.45.634.

[7] J. Bowen & S. Reeves (2012): *Modelling User Manuals of Modal Medical Devices and Learning from the Experience*. In: *4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '12)*, pp. 121–130, doi:10.1145/2305484.2305505.

[8] J. Bowen & S. Reeves (2015): *Design Patterns for Models of Interactive Systems*. In: *24th Australasian Software Engineering Conference (ASWEC)*, IEEE, pp. 223–232, doi:10.1109/ASWEC.2015.30.

[9] A. Cerone, P. Curzon, J. Bowen & S. Reeves (2007): *Formal Models for Informal GUI Designs*. *Electronic Notes in Theoretical Computer Science* 183, pp. 57–72, doi:10.1016/j.entcs.2007.01.061.

[10] Guiran Chang, Chunguang Tan, Guanhua Li & Chuan Zhu (2010): *Developing Mobile Applications on the Android Platform*. In: *Mobile Multimedia Processing*, pp. 264–286, doi:10.1007/978-3-642-12349-8_15.

[11] T. Erkkinen & M. Conrad (2007): *Safety-critical software development using automatic production code generation*. Technical Report, SAE Technical Papers, doi:10.4271/2007-01-1493.

[12] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat & M. Verhoef (2005): *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA.

[13] J. D. Foley & P. Noi Sukaviriya (1994): *History, Results, and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation*. In: *Proceedings of Design, Verification and Specification of Interactive Systems (DSVIS'94)*, pp. 3–14, doi:10.1007/978-3-642-87115-3_1.

[14] (2016): *Handlebars Semantic Template*. Available at `http://handlebarsjs.com`.

[15] M. D. Harrison, J. C. Campos & P. Masci (2015): *Reusing models and properties in the analysis of similar interactive devices*. *Innovations in Systems and Software Engineering* 11(2), pp. 95–111, doi:10.1007/s11334-013-0201-3.

[16] MD. Harrison, JC. Campos, R. Rimvydas & P. Curzon (2016): *Modelling information resources and their salience in medical device design*. In: *8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '16)*, doi:10.1145/2933242.2933250.

[17] Campos JC & Harrison MD (2001): *Model checking interactor specifications*. *Automated Software Engineering* 8(3–4), pp. 5275–310, doi:10.1023/A:1011265604021.

[18] (2016): *Java Native Interface*. `http://docs.oracle.com/javase/8/docs/technotes/guides/jni/`.

[19] P. Masci, A. Ayoud, P. Curzon, MD. Harrison, I. Lee & H. Thimbleby (2013): *Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example*. In: *5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, (EICS '13)*, doi:10.1145/2494603.2480302.

[20] P. Masci, P. Mallozzi, F. L. De Angelis, G. Di Marzo Serugendo & P. Curzon (2015): *Using PVSio-web and SAPERE for rapid prototyping of user interfaces in Integrated Clinical Environments*. In: *Verisure2015, Workshop on Verification and Assurance, co-located with CAV2015.*

[21] P. Masci, P. Oladimeji, P. Curzon & H. Thimbleby (2014): *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).*

[22] P. Masci, P. Oladimeji, P. Curzon & H. Thimbleby (2015): *PVSio-web 2.0: Joining PVS to Human-Computer Interaction*. In: *27th International Conference on Computer Aided Verification (CAV2015)*, Springer, doi:10.1007/978-3-319-21690-4_30. Tool and application examples available at http://www.pvsioweb.org.

[23] P. Masci, Yi Zhang, P. Jones, P. Oladimeji, E. D'Urso, C. Bernardeschi, P. Curzon & H. Thimbleby (2014): *Combining PVSio with Stateflow*. In: *6th NASA Formal Methods Symposium (NFM2014)*, doi:10.1007/978-3-319-06200-6_16.

[24] C. Muñoz (2003): *Rapid prototyping in PVS*. Technical Report NIA 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA, USA.

[25] (2016): *NDK*. Available at `http://developer.android.com/ndk`.

[26] P. Oladimeji, P. Masci, P. Curzon & H. Thimbleby (2013): *PVSio-web: a tool for rapid prototyping device user interfaces in PVS*. In: *FMIS2013, 5th International Workshop on Formal Methods for Interactive Systems*, doi:10.14279/tuj.eceasst.69.963.

[27] S. Owre, J. M. Rushby & N. Shankar (1992): *PVS: A Prototype Verification System*. In: *Automated Deduction—CADE-11: 11th International Conference on Automated Deduction*, pp. 748–752, doi:10.1007/3-540-55602-8_217.

[28] M. Pajic, Zhihao Jiang, Insup Lee, O. Sokolsky & R. Mangharam (2014): *Safety-critical Medical Device Development Using the UPP2SF Model Translation Tool*. *ACM Trans. Embed. Comput. Syst.* 13(4s), pp. 127:1–127:26, doi:10.1145/2584651.

[29] F. Paternò, C. Santoro & L. D. Spano (2009): *MARIA: A Universal, Declarative, Multiple Abstraction-level Language for Service-oriented Applications in Ubiquitous Environments*. *ACM Trans. Comput.-Hum. Interact.* 16(4), pp. 19:1–19:30, doi:10.1145/1614390.1614394.

[30] Raymond Merrill Smullyan (1995): *First-order logic*. Dover publications, New York.

[31] Mandayam Srivas, Harald Rueß & David Cyrluk (1997): *Hardware Verification Using PVS*. In Thomas Kropf, editor: *Formal Hardware Verification: Methods and Systems in Comparison*, *Lecture Notes in Computer Science* 1287, Springer-Verlag, pp. 156–205, doi:10.1007/3-540-63475-4_4.

A Correctness of code generation

In order to assess the correctness of the generated code, the Emucharts diagram is taken as the reference model, and a correspondence is established between the evolution of the model and that of the executed code.

A.1 Transition system for an Emucharts diagram

As discussed above (section 4), an Emucharts diagram is a graph of nodes and labelled transitions, extended with a set of typed context variables, each one with an initial value. Its semantics is given by a transition system. Let the following be defined:

- A set $N = \{n_1, \ldots, n_i\}$ of nodes;
- a set $X = \{x_1, \ldots, x_j\}$ of context variables (for simplicity, assumed to be typeless);
- a set $\mathbb{V}$ of values;
- a set $E = \{\epsilon_1, \ldots, \epsilon_k\}$ of events;
- a set $G = \{g_1, \ldots, g_l\}$ of guards, i.e., Boolean expressions involving variables, constants from $\mathbb{V}$, arithmetic and relational operators;
- a denumerable set V of *valuations*, i.e., functions from X to $\mathbb{V}$;
- a set $A = \{a_1, \ldots, a_l\}$ of arcs, i.e., 5-tuples of the form (s, t, e, g, v), where $s, t \in N$ are the arc's source and target node, $e \in E$, $g \in G$, and $v \in V$ is the valuation defined by the action labelling the corresponding transition in the diagram; more precisely, v is the valuation obtained by overriding the previous valuation with the assignments in the action associated with the arc;
- a set $\mathbb{Q}$ of states of the form $\langle n, v \rangle$, with $n \in N$ and $v \in V$;

$$\textbf{arc}\ \frac{\epsilon,(p,q,e,g,v');\ \epsilon = e \wedge n = p \wedge v \models g}{\langle n,v\rangle \rightarrow \langle q,v'\rangle}$$

$$\textbf{idle}\ \frac{\epsilon,(p,q,e,g,v');\ \epsilon \neq e \vee n \neq p \vee v \not\models g}{\langle n,v\rangle \rightarrow \langle n,v\rangle}$$

Figure 6: Emucharts operational semantics.

- a transition relation $\rightarrow \subseteq Q \times Q$, defined by the semantic rules in Figure 6, where the premises contain an event ϵ, an arc label, and a logical condition, and the consequences contain a member of the transition relation that is enabled if the condition holds.

With the above definitions, the associated transition system T is the tuple $(\mathbb{Q}, \rightarrow, q_0)$, where $q_0 = \langle n_0, v_0\rangle$ is the initial state. Since the diagram is deterministic, given a sequence of event occurrences $e_1, \ldots, e_k, \ldots$, the transition system has only one sequential path. If an event cannot affect a state (either it is not permitted or no guard prefixed by the event is satisfied), the system does not change state. The operational semantics are given in Figure 6.

A.2 Transition system for the generated code

The generated functions are used within a more complex system, which is responsible for catching events at the real or simulated user interface and for calling the respective functions according to an appropriate protocol: the *init* function must have been called previously, then, when an event is catched, the permission function of the corresponding trigger is called, and only if it returns *true* can the respective transition function be executed.

Assume that the data entry subsystem of the device is controlled by a program P that responds to input events by calling the respective functions. These function will take the device to the next state.

Also the program P can be modelled as a transition system T_P based on the following sets, each one being isomorphic ($\cong$) to the corresponding set in T, or an extension to that set: (*i*) A set $N_P \cong N$ of node labels, each represented by an enumerator of the *node_label* type in P; (*ii*) a set $X_P = X_c \cup \{x_{\text{curr}}\}$ of variables, where $X_c \cong X$, each variable in X_c represents a context field of the *state* structure in P, and x_{curr} represents the *curr_node* of the *state* structure; (*iii*) a set $\mathbb{V}_P = \mathbb{V}_c \cup N_P$ of values, where $\mathbb{V}_c = \mathbb{V}$; (*iv*) a set $E_P \cong E$ of events, each one associated with one permission function and one transition function in P; (*v*) a set $G_P \cong G$ of guards, each implemented as the condition of an *if* statement in P; (*vi*) a denumerable set $V_P = V_c \cup V_n$ of valuations from X_P to $\mathbb{V}_P$, where $V_c \cong V$ and $V_n\colon \{x_{\text{curr}}\} \rightarrow N_P$; (*vii*) a set $A_P \cong A$ of arcs, where each arc has the form $(v_n(x_{\text{curr}}), v'_n(x_{\text{curr}}), e, g, v'_c)$, and each arc represents an *if* statement in the transition function for event e having guard g as its condition and valuation $v' = v'_n \cup v'_c$ as its controlled statement, with v'_n implemented by the *enter* function and v'_c by the assignments specified in the Emucharts diagram.

With the above definitions, let Q_P be a set of states where each state is a pair $\langle v_n, v_c\rangle$, with $v_n \in V_n$, $v_c \in V_c$. The transition relation $\xrightarrow{P} \subseteq Q_P \times Q_P$ is defined by the semantic rules in figure 7 applied to elements of the above sets, and implemented by the permission functions, which check for each event e if the condition $v_n(x_{\text{curr}}) = p$ holds or not, and by the transition functions, which check if the current values of the variables satisfy the guards, and update node and variables accordingly. The associated transition system T_P is the tuple $(Q_P, \xrightarrow{P}, q_{P0})$, where q_{P0} is the state defined by the initial values of x_{curr} and of the context variables, set by the *init* function. The operational semantics are given in Fig. 7.

$$\textbf{arc}_P \quad \frac{\epsilon, (p, q, e, g, v_c'); \ \epsilon = e \wedge v_n(x_{\text{curr}}) = p \wedge v_c \models g}{\langle v_n, v_c \rangle \xrightarrow{P} \langle q, v' \rangle}$$

$$\textbf{idle}_P \quad \frac{\epsilon, (p, q, e, g, v_c'); \ \epsilon \neq e \vee v_n(x_{\text{curr}}) \neq p \vee v_c \not\models g}{\langle v_n, v_c \rangle \xrightarrow{P} \langle n, v_c \rangle}$$

Figure 7: Generated code operational semantics.

A.3 Equivalence of the transition systems

To prove the correctness of the generated code, we introduce the definition of equivalence between Emucharts states and the program states.

Definition 1. *A member m of one of the sets N, X, $\mathbb{V}$, E, defined in T, is equivalent (∼) to the member m_P paired to m by the isomorphism between the set containing m and the corresponding set in T_P.*

Definition 2. *A state $q = \langle n, v \rangle$, $q \in Q$, is equivalent (∼) to a state $q_P = \langle v_n, v_c \rangle$, $q_P \in Q_P$ iff $n \sim v_n(x_{\text{curr}})$ — so the value of x_{curr} is equivalent to node n, and $\forall_{x \in X} v(x) = v_c(x_P)$ (i.e., matching variables in q and q_P have the same values).*

The proof of correctness for the generated code is by induction on the length of computation. We assume that T and T_P are the transition systems modelling, respectively, an Emucharts diagram and a program that uses the generated code, respecting the previously introduced protocol, and accepts a sequence of input events.

Theorem 1. *Let T and T_P be the transition systems introduced in the above paragraphs, and $e = e_1, e_2 \ldots$ be a sequence of input event sequences. Let $\sigma = q_0, q_1, \ldots$ and $\sigma_P = q_{P0}, q_{P1}, \ldots$ be sequences of states, with $q_i \rightarrow q_{(i+1)}$ and $q_{Pi} \xrightarrow{P} q_{P(i+1)}$.*

We prove that, at each step of the computation, $q_i \sim q_{Pi}$:

Induction base. $q_0 \sim q_{P0}$ by construction.

Induction step. Let $q_j \sim q_{Pj}$ at step j. On the occurrence of an event e, let $q_j \rightarrow q_{(j+1)}$ and $q_{Pj} \xrightarrow{P} q_{P(j+1)}$. We can prove that $q_{(j+1)} \sim q_{P(j+1)}$ by case analysis: (1) e not permitted in q_j; (2) e permitted and guard not satisfied; and (3) e permitted and guard satisfied.

Case 1: e not permitted. If the event is not permitted in the current state, rules **idle** and $\textbf{idle}_P$ apply to T and T_P, respectively, so that $q_{(j+1)} = q_j$ and $q_{P(j+1)} = q_{Pj}$, equivalent by induction hypothesis. Recall that the permission function for e returns *false* in this case, and by hypothesis program P does not call the corresponding transition function.

Case 2: e permitted and guard not satisfied. Also in this case, rules **idle** and $\textbf{idle}_P$ apply to the transition systems. The *if* statements in P check that the guard does not hold, and the respective controlled statements are not executed.

Case 3: e permitted and guard satisfied. In this case, Rules **arc** and $\textbf{arc}_P$ apply to both transition systems, therefore (*i*) T moves from state $q_j = \langle n, v \rangle$ to state $q_{(j+1)} = \langle n', v' \rangle$, or (*ii*) T_P moves from state $q_{Pj} = \langle v_n, v_c \rangle$ to state $q_{P(j+1)} = \langle v_n', v_c' \rangle$. Valuation v_n' maps x_{curr} to a node label equivalent by definition to n', and v_c' maps the context variables in T_P to values equivalent by definition to those assigned by v' to the context variables in T'.

The new states in the two transition systems are therefore equivalent.