Extending a user interface prototyping tool with automatic MISSRA C code generation

G. Mauro$^1$  H. Thimbleby$^2$  A. Domenici$^1$  C. Bernardeschi$^1$

$^1$Department of Information Engineering
University of Pisa, Italy

$^2$Swansea University — Prifysgol Abertawe, Swansea/Abertawe, UK

3rd Workshop on Formal Integrated Development Environment (Formal-IDE)
satellite workshop of FM2016
Limassol, Cyprus, November 8, 2016
Keywords

- User interface prototyping tool
  - malfunctions often arise from ill-designed interfaces
  - realistic simulations help understand and validate user interactions
- Formal specification
  - formal specifications reduce the risk of design errors
- **MISRA C code generation**
  - automatic code generation from validated models improves dependability
Model-driven development and formal models

MDD is essentially:

- Creating an executable model
- Validating the model by simulation
- Implementing the model by automatic code generation

The model is usually expressed in a block-based graphical language.

The language is formal in that it has a well-defined semantics, but . . .

. . . \textit{it has not been conceived} with formal verification in mind.

By \textbf{formal model} we mean a simulatable, executable and formally verifiable model.

Formal models enrich MDD by adding formal verification to simulation.
Background: The Prototype Verification System

Proving:
The PVS is an interactive theorem prover environment based on:

- A **typed higher-order logic** language
- a **sequent calculus** deduction system.

A PVS **theory** is a collection of definitions and statements, including axioms.

A PVS model is a collection of theories describing a system.

A system’s requirement is expressed as a **theorem** to be proved wrt the theory.

Animating:
The PVSio extension is a **ground evaluator** that translates PVS function definitions into LISP code.

A PVS function definition may contain applications of **extra-logical** functions, providing, e.g., input and output.

A PVS model can then be animated, i.e., **simulated**.
PVSio-web and Emucharts

PVSio-web is a framework for prototyping and simulation of interactive device interfaces.

- Originally conceived for medical devices
- A formal model of the interface can be written in PVS, or
- entered in the graphical Emucharts language and translated to PVS
- PVS functions are associated with active areas of device pictures.

Users

- access the PVSio-web framework with any web browser, and
- interact with the simulated device clicking on buttons in the device picture.
The PVSio-web user interface
Code generation in the PVSio-web development process

Developer

Emucharts Editor

PVS generator

Model Editor

PVS environment

Theorem Prover

Ground Evaluator

PVS theory

HB Templates

C Code Generator

C Code

Simulation

Proof

Emucharts Model
The C language was chosen as a target language due to its widespread use in embedded systems.

The MISRA (Motor Industry Software Reliability Association) guidelines were adopted to enhance dependability.

The C code generator produces the **source code** for a **module** implementing the interface subsystem of the device.

The generated module can be linked to software to **control** the actual device or **simulate** it.
Emucharts is a formal **state machine** language with **guards**, **context variables**, and **actions** on context variables.

PVSio-web stores Emucharts diagrams in **Json files**. **Handlebars** templates extract information from Json files and insert them into C code text.

For each transition trigger (**input event**) $\tau$, two functions are generated:

- a **permission** function checks if any transition from the current state is triggered by $\tau$, and if so
- a **transition** function checks whether the guard (if any) is satisfied and executes the transition accordingly.
Translation mechanics

The C code generator is a JavaScript module that parses the JSON representation of a diagram and produces code (and documentation) from Handlebars templates:

```javascript
/** * Enumeration of state labels. */
typedef enum {
    {{#each states~}}{{name}}{{#unless @last}},{{/unless}}
} MachineState;{{/if}}
...
{{#if transitions}}/* definition of permission function */
{{#each transitions}}{{#if id}}UC_8 per\_{{id}}(const state *st) {{
    if(st->current_state == {{source.name}}){
        return true;
    }
    return false;
}}{[/if]{[/each]}}{[/if]}
```
Example: Emucharts diagram for MiniMed 530G data entry

```
# Example code

click_UP [ display = 10 ] { display := 10 }

click_UP [ display < 10 ] { display := display + 0.1; }

click_DOWN [ display = 0 ] { display := 0 }

click_DOWN [ display > 0 ] { display := display - 0.1; }
```

- **Trigger**
- **Guard**
- **Action**
- **Context Variable**
typedef enum { off, on } node_label; %
typedef struct {
    D_64 display; %
    node_label curr_node; %
    node_label prev_node; } state; %

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

D_64 (double float, 64-bit) and UC_8 (unsigned char, 8-bit) are MISRA C type aliases.
state click_UP(state* st) { % transition function
    assert(st->current_state == on);
    assert(st->display < 10 || st->display == 10);
    if (st->display < 10 && st->current_state == on) {
        st->display = st->display + 0.1f;
        enter(on, st);
        assert(st->current_state == on);
        return *st;
    }
    if (st->display == 10 && st->current_state == on) {
        st->display = 10.0f;
        enter(on, st);
        assert(st->current_state == on);
        return *st;
    }
    return *st;
}
%-- machine states
MachineState: TYPE = { off, on }

%-- emuchart state
State: TYPE = [
    current_state: MachineState,
    previous_state: MachineState,
    display: real
]

... per_click_UP(st: State): bool =
    ((current_state(st) = on) AND (display(st) < 10))
    OR ((current_state(st) = on) AND (display(st) = 10))
click_UP(st: (per_click_UP)): State =
  COND
  (current_state(st) = on) AND (display(st) < 10)
    -> LET new_st = leave_state(on)(st),
        new_st = new_st WITH [ display := display(st) + 0.1 ]
        IN enter_into(on)(new_st),
  (current_state(st) = on) AND (display(st) = 10)
    -> LET new_st = leave_state(on)(st),
        new_st = new_st WITH [ display := 10 ]
        IN enter_into(on)(new_st)
ENDCOND
An application: the Alaris GP infusion pump

A volumetric infusion pump with a wide range of infusion rates (1 ml/h to 1200 ml/h).

A floating point display with three significant digits.

Numerical input through single- and double-chevron buttons.
Numerical input for infusion rate

- If the displayed value $d$ is $< 100$,
  - a single-chevron button adds $\pm 0.1$
  - a double-chevron button changes the value to the next higher or lower decade (e.g., from 9.1 to 10.0)

- If $100 \leq d < 1000$,
  - a single-chevron button adds $\pm 1$
  - a double-chevron button changes the value to the next higher or lower hundred plus the decade of $d$ (e.g., from 314 to 410)

- If $d \geq 1000$,
  - a single-chevron button adds $\pm 10$
  - a double-chevron button changes the value to the next higher or lower hundred (e.g., from 1010 to 1100)

- Buttons have no effect when the maximum or minimum allowed rate values are displayed.
Emucharts diagram for numerical data entry
A mobile application

The generated C code can run on a mobile device.

Interactive simulations on a mobile device are more realistic and can be used for training purposes.

The user interface code generated for the Alaris GP has been ported to the Android platform.
Conclusions

Code generation enables the PVSio-web framework to close the MDD cycle, from formal specification to industry-standard code:

- Developers (possibly unfamiliar with the PVS language) build a device interface model with the graphical, state-machine based Emucharts language.
- The model can be both verified by theorem proving and validated by simulation.
- The verified/validated model is implemented automatically by C code generation.
Thank you

Ευχαριστώ