Open source tools for usability analysis of medical devices

Prof. Cinzia Bernardeschi

Department of Information Engineering
University of Pisa

cinzia.bernardeschi@unipi.it
1. Use errors in medical devices

2. PVSio-web framework
   1. The Prototype builder
   2. The Simulator
   3. How to build a prototype: a simple example
   4. Available demos: Bbraun infusion pump, Alaris GP, Radical7 monitoring system, Stellant contrast media injector

3. A case study on infusion pumps

4. Building a prototype of a contrast media injector

5. Prototypes of more complex systems
   - ICE
   - Heart-pacemaker

6. Conclusions
1. Use errors in medical devices
Most medical devices used in hospital and home care are interactive, they are controlled by software that governs key aspects of the user interface and performs key safety functions.

**Medical monitors**
- monitor vital parameters, blood pressure, pulse oximetry and respiratory rate

**Infusion pumps**
- deliver fluids, such as nutrients and medications, into a patient’s body in controlled amounts

**Contrast media injectors**
- inject contrast and saline into the bloodstream of patients (used, for example, in CT)
Use errors in medical devices

Use errors with infusion devices, as well as with other medical devices, is a known source of incidents in healthcare

- Design flaws can induce use errors in the data entry system of a device (e.g., silently discarding decimal point key presses for certain range of values)

- Critical control-flows of functions and different operating modalities can induce use errors in the programming of a device (e.g., interleaving of operations of different operating modes)

Main points:
(i) understanding of the design challenges with user interface software for medical systems
(ii) tools and techniques for design and analysis of software incorporated in interactive medical systems
Usability of medical devices

Human factors play a fundamental role in reducing use errors in medical devices: the user must be considered in the device design and usability tests are fundamental for limiting the use errors.

User interface issues can potentially lead to use errors, e.g., complicated menu structures have a huge potential for accidentally making an error.

If medical devices are to be used safely, it is important that “user interface software” is designed to make the device easy to use and mistakes made by users are corrected.

“This recommended practice covers general human factors engineering (HFE) principles, specific HFE principles geared towards certain user-interface attributes, and special applications of HFE (e.g., connectors, controls, visual displays, automation, software–user interfaces, hand tools, workstations, mobile medical devices, home health care devices).”


“This part of IEC 62366 specifies a process for a manufacturer to analyse, specify, develop and evaluate the USABILITY of a MEDICAL DEVICE as it relates to SAFETY. This usability engineering (HUMAN FACTORS ENGINEERING) process permits the manufacturer to assess and mitigate RISKS associated with correct use and use errors, i.e., normal use.”
Due to more sophisticated software, recalls have increased since 2006

Due to more sophisticated software, recalls have increased since 2006.

<table>
<thead>
<tr>
<th>Year</th>
<th>Total Recalls</th>
<th>Software-Related Recalls</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>604</td>
<td>84</td>
<td>13.9%</td>
</tr>
<tr>
<td>2006</td>
<td>663</td>
<td>119</td>
<td>17.9%</td>
</tr>
<tr>
<td>2007</td>
<td>638</td>
<td>119</td>
<td>18.7%</td>
</tr>
<tr>
<td>2008</td>
<td>847</td>
<td>192</td>
<td>22.7%</td>
</tr>
<tr>
<td>2009</td>
<td>782</td>
<td>146</td>
<td>18.7%</td>
</tr>
<tr>
<td>2010</td>
<td>981</td>
<td>147</td>
<td>15.0%</td>
</tr>
<tr>
<td>2011</td>
<td>1,277</td>
<td>315</td>
<td>24.7%</td>
</tr>
</tbody>
</table>

Percentage of Recalls Related to Software

2. PVSio-web framework
PVSio-web [1, 2] is an open-source environment enabling developers and users of medical devices to assess and validate them with respect to human-machine interaction.

This makes it possible to discover subtle interfacing errors that may have grave consequences, such as under- or over-dosing [3].

PVSio-web is implemented in JavaScript by a software platform composed of several scripts, invoked and coordinated through a web interface.


PVSio-web

The tool and application examples available at
http://www.pvsio-web.org
PVSio-web consists of two tools:

Prototype Builder       Simulation Environment.


To install PVSio-web, first you need to install PVS and NodeJS, and then clone the PVSio-web github repository.

- PVS is open source, under the GNU General Public License (GPL), http://pvs.csl.sri.com/download.shtml.

- NodeJS can be downloaded at http://nodejs.org


A developer uses the Prototype Builder tool to create a graphical representation of the device's front panel and link its controls and displays to functions describing how the device responds to user actions on the controls and how it shows information on the displays.
A developer uses the Model Editor to create the system model

```plaintext
alarisGH_ArenaCC: THEORY
BEGIN

-- constants & type definitions

max: nonneg_real = 1200
min: nonneg_real = 0.01
alaris_real: TYPE = (x: nonneg_real | x <= max)

state: TYPE =

[# left_display: alaris_real,
timer: alaris_timer,
step: alaris_step #]

|-- alaris' chevron (UP,up,dn,EN) -----------------------------------------------

alaris_up(delta: alaris_step, val: alaris_real): alaris_real =

  IF val < 10
  THEN trim( floor((val*100) / 100 )

  ELSIF val >= 10 AND val < 100
  THEN trim( floor((val*10) / 10 )

  ELSIF val >= 100 AND val < 1000
  THEN trim( floor((val) )

  ELSE
     trim( (floor(val/10) + delta) * 10 ) ENDIF

alaris_UP(delta: alaris_step, val: alaris_real): alaris_real =

  IF val < 10
  THEN trim( floor((val*10) / 10 )

  ELSIF val >= 10 AND val < 100
  THEN trim( floor(val) + delta )

  ELSIF val >= 100 AND val < 1000
  THEN trim( (floor(val/10) + delta) * 10 )

  ELSE
     trim( (floor(val/100) + delta) * 100 ) ENDIF
```

2nd UBORA Design School, Pisa - September 3-7, 2018
A Simulation Environment enables a developer or a prospective user to simulate the device and interact with the simulation through the image of its panel.

Screenshot of Simulator View section when the user clicks on a virtual device button (the up arrow key or the double up arrow key) several times and reach 0.27 value and the each of this action is shown on virtual device display.
Behind PVSio-web

A PVSio-web prototype follows a Model-View-Controller (MVD) design pattern, which promotes a clear separation between the behavior of the prototype and its visual appearance.
Basic elements for the visual appearance of the prototype:

- a realistic picture of the device

- a Javascript module that
  - contains the interactive widgets,
  - captures user interactions with the prototype and
  - renders the device state

- The interactive widgets can be
  - input widget (e.g., button, sliders, …)
  - output widget (e.g., digital displays)

- a HTML file, for loading and executing the visual appearance of the prototype in a web browser
The widget library (written in Javascript) can be used for

1. capturing user actions performed on selected regions of the prototype
2. animating selected regions of the prototype in response to user actions
3. Translating user actions into commands that drive the evaluation of the device model simulated on the PVSio-web back-end

Example: DISPLAY widget, using Basic Display (BasicDisplayEVO)

instantiate the widget using the constructor method which takes three arguments

- A unique identifier for the widget
- The position and size of the widget (top, left, high, width)
- Optional attribute for customizing the visual aspects and functionality of the widget

- use the *render* method to make the widget visible and render, for example, the string “Hello world”
To execute the prototype

1. start the PVSio-web back-end

2. open a web browser at the page
   http://www.pvsioweb.org/tutorials/HelloWorld
Loading and executing the prototype

HTML code for loading the javascript file of the prototype in the web browser

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title></title>
  <meta name="description" content="">
  <meta name="viewport" content="width=device-width">
  <link href="../../client/lib/bootstrap/css/bootstrap.min.css" rel="stylesheet" media="screen">
  <link href="../../client/css/style.css" rel="stylesheet" type="text/css"/>
  <link href="../../client/css/animate.css" rel="stylesheet" type="text/css"/>
  <link href="../../client/lib/jquery-toggles/css/toggles.css" rel="stylesheet">
  <link href="../../client/lib/jquery-toggles/css/themes/toggles-modern.css" rel="stylesheet">
  <link href="../../client/lib/bootstrap/css/bootstrap-slider.css" rel="stylesheet" type="text/css"/>
</head>
<body class="noselect" style="background:#dedfdd;">
  <div id="content" class="content">
    <img src="device.png">
    <div id="screen">
    </div>
  </div>
  <script defer="true" src="../../client/lib/layout.js"></script>
  <script defer="true" src="../../client/lib/promise-0.1.1.js"></script>
  <script defer="true" src="../../client/lib/keys.js"></script>
  <script defer="true" src="../../client/lib/jquery.js"></script>
  <script defer="true" src="../../client/lib/underscore/underscore.js"></script>
  <script defer="true" src="../../client/lib/d3/d3.js"></script>
  <script defer="true" src="../../client/lib/backbone.js"></script>
  <script defer="true" src="../../client/lib/handlebars-v1.3.0.js"></script>
  <script defer="true" src="../../client/lib/bootstrap/js/bootstrap.min.js"></script>
  <script defer="true" src="../../client/lib/jquery-toggles/toggles.min.js"></script>
  <script defer="true" src="../../client/lib/bootstrap/js/bootstrap-slider.js"></script>
  <script defer="true" src="../../client/require.js" data-main="index.js"></script>
</body>
</html>
```
The model of the behaviour of the system must be written in the PVS language. It can be entered textually with the Model Editor, but it can also be generated from the Emucharts Editor.

Emucharts is a graphical language to define finite-state automata extended with a set of variables, and where each transition is annotated with an event trigger, an optional guard, and an optional action. The event trigger models an atomic input to the automaton, the guard is a logical condition (true by default) on the variables, and the action is a set of assignments to the variables.

An Emucharts automaton is translated into an executable PVS theory by the PVSio-web Code Generator.
A simple example is the following: a two-state diagram where turning on the device changes its state from OFF to ON and only in the ON state it is possible to click buttons UP or DOWN (both of clicks are bounded by the minimum and maximum thresholds 0 and 100 values).

val: integer; initialised at 0; min = 0; max = 100
ON, OFF: button
UP, DOWN: button
User action: click button ON, click button OFF,
click button UP, click button DOWN
Download and install PVSio-web on your computer

- Start PVSio-web back-end
  
  ./start.sh

- Create a New Project
  - insert the name
  - load picture: tutorials/Echo/device.png

Prototype Builder:
- Create Numeric Display. Name: d
- Create a Button. Name: b
How to build a prototype: a simple example

ModelEditor:
- write the model or import a file with the model

Simulator View
- simulate the model

counter: THEORY
BEGIN
state: TYPE = [#
  d: integer #]

init (x: real): state = (# d := 0 #)

click_b (st: state): state =
st WITH = [# d := d(st) + 1 #]

tick(st: state): state = st
END counter

after 1 click
To sum it up

1) Take a picture of the user interface of the real system that could be used as a basis to create the visual appearance of the interactive simulation, the front-end of the prototype.

2) Build an executable model (an executable specification) of the behavior of the system in the PVS language, the back-end of the prototype.

3) Use the PVSio-web library to create interactive widgets over the picture of the system:
   • Input widgets translate user actions over buttons into expressions of the executable PVS model to be evaluated to compute the system response
   • Output widgets mirror state attributes of the PVS model and resemble the look & feel of the real system in the corresponding state.

4) Use the simulator to validate the user interface of medical devices by interactively animating a formal specification of the user interaction with the device.
The following demos are available in the PVSio-web framework
To execute a demo, open a web browser at the specified page

**BBraun perfusor**
http://www.pvsioweb.org/demos/BBraun

**Radical7 Patient monitor**
http://www.pvsioweb.org/demos/Radical7

**Stellant** contrast media injector
http://www.pvsioweb.org/demos/stellantV2

**AlarisGP** infusion pump
http://www.pvsioweb.org/demos/AlarisGP
3. A case study on infusion pumps
Infusion pumps

Infusion pumps are medical device that deliver drugs and nutrients into a patient body at controlled rates and volumes

Rate and volumes are usually entered by nurses using buttons and keys on the device interface

Design errors that can lead to situations in which wrong rates or volumes can be accidentally inserted in the device have been identified.

Work developed within the CHI+MED research project (http://www.chi-med.ac.uk/), and in collaboration with the Center for Devices and Radiological Health of the US Food and Drug Administration (CDRH/FDA).

CHI+MED (Computer-Human Interaction for Medical Devices, EP/G059063/1) was an EPSRC-funded project to improve the safety of interactive (programmable) medical devices.

EPSRC - Engineering and Physical Sciences Research Council (UK’s agency for funding research in engineering and the physical sciences)
An accurate specification of the device behaviour has been obtained by reverse-engineering the real device.

An example of user interface with navigation keys: 4 arrow keys and a display

In the manual: to enter volume or rates, just use the arrow keys (up key/down key, to increment/decrement the digit; left/right key to select different digits)

Digits are not independent.

Issues when we enter high value: unexpected behaviour of the device.
Another example of user interface with navigation keys: 4 arrow keys and a display

Independent digits.

We are allowed to go from the rightmost position to the leftmost position with one click.
Another user interface with navigation keys. Independent UP and DOWN keys, and the impression is that each key operates on a different digit. But … digits are not independent. Moreover when you overshoot the maximum value, the display shift numbers, and the last integer digit is smaller than the other integer digits.
The device ignores key presses: 20.01 is displayed 20.0 (for numbers above 10, only one fractional digit; for numbers above 100, no fractional digit) The device ignores the decimal point: 214.2 returns High value (2142) But 100.1 is displayed as 1001, and the infusion can be started
Viewing angle in seven segment display

**Patient monitors**
- Alcon Everest
- Datascope Accutorr Plus

**Alaris PC infusion pump**
Lessons learned

Identification of design issues

- user input erroneously discarded
- inappropriate feedback
- unexpected device modes
- some device silently discard after a timeout
- other devices silently confirm after a timeout

A recorded video of the demonstration is available on YouTube "Medical Device Training - Design Issues in Medical User Interfaces"
https://www.youtube.com/watch?v=T0QmUe0bwL8
4. Building a prototype of a contrast media injector

joint work with
Paolo Masci, Department de Informatica, Universidade do Minho, Braga, Portugal
Davide Caramella, Ruggero Dell'Osso, Department of Diagnostic and Interventional Radiology, University of Pisa, Italy

Bernardeschi C, Masci P, Caramella D, Dell'Osso R.
Contrast media injectors inject contrast and saline into the bloodstream of patients. Iodinated media can be nephrotoxic, being a known cause of possible acute renal failure in hospitalised patients.

To minimize the risk of adverse health problems, it is therefore important to set up the injectors so that it delivers the minimal amount of contrast media necessary for the diagnostic task.

Sophisticated interfaces and injection setting are available.

The simulation
- was created to support training of clinicians, and
- helped to identify and raise awareness among clinicians of critical workflows that could induce accidental use errors that may have safety implications.
Stellant CT contrast media injector

- Dual-syringe injector that performs injection of contrast and saline into the bloodstream of patients

- The workstation allows clinicians to set up and manage personalized injection protocols for different patients, based on parameters such as patient weight, past scan procedures, current scan settings, diagnostic tasks, etc.

Commonly used in CT in hospitals

The injection system includes a workstation and an injector

Interaction with the workstation is carried out through a touchscreen display
The front panel of the injector uses a number of displays and LEDs to provide feedback to the clinician about the state of the device.

Various buttons on the front panel of the injector allow clinicians to operate the device.

The body of the device includes an injector head (where syringes are inserted), plungers for controlling the volume of liquid in the syringes (plungers are automatically advanced and retracted when syringes are inserted and removed), and tubes/needles (used to connect syringes and the patient).

The workstation seamlessly communicates with the injector, allowing clinicians to monitor progress of the injection directly from the workstation screen.
The injector: displays and lights

- Two seven-segments displays (VolumeA and VolumeB) report, depending on the device mode, either the volume of liquid in the syringe, or the position of the plunger.

  Each display has three significant digits, and can render only integer numbers.

- One LED light placed next to a lock symbol indicates whether the injection protocol has been set and locked from the workstation.

- Two large LED lights indicate whether an injection is running.
The injector: filling syringes

**Automatic filling**
An Autoload button, can be used to move the plungers and load the volume of saline and contrast configured by the clinician on the workstation (plus (+) and minus(-) can be used to increase/decrease the volume before filling syringes).

Two buttons FillA and FillB activate the autoload sequence for the syringes.

**Manual filling**
A Manual load button enables manual adjustment of the plunger position using the chevron keys available on the front panel of the injector (speed - where the chevron keys are pressed)
The injector: control injection

A Prime button can be used to remove air-in-line

A Check-Air button is for checking air-in-line

An Arm button is used to make the system ready for an injection.

An Abort button terminates an injection procedure and disarms the injection.

A Start/Hold button allows to start the injection (when the injection is not started), and to pause the injection (this function is active when an injection is running).
The contrast media injector prototype

an interactive simulation of the complete injection system using the PVSio-web prototyping framework

screenshot of the simulation

Live version of the simulation:  http://www.pvsio-web.org/demos/stellantV2
In the example, output widgets are used to represent a system state where

- the syringes are plugged into the injector and spiked to a bag with saline and contrast liquids

- the injector has completed the process of loading the saline and contrast liquids in the syringes (the two seven segments displays on the front panel of the injector indicate the volume of liquid loaded in the two syringes)
The specification of the system is reverse engineered using in combination

- the user manual
- direct interaction with the real device
- the results of a field study we conducted that focused on how expert
  users routinely operate the device

This allowed the entire team to look closely and in a systematic manner into
various design aspects of the system.

This greatly helped engineers understand how clinicians use the system, and
greatly helped the entire team to discuss and demonstrate various corner
cases that could potentially have safety consequences in specific contexts.
Executable PVS specification

Two main steps:

Specify the system state as a PVS record type with relevant state attributes

```
state: TYPE = [#
  mode: Mode,
  vol_saline: Volume,
  vol_contrast: Volume,
  lock_LED: LED,
  ...
#]
```

Specify the behavior of the system as a set of transition functions that range over system states.

```
click_btn_manual(st: state): state =
st WITH [
  mode := MANUAL,
  vol_saline := plunger_saline(st),
  vol_contrast := plunger_contrast(st),
  vol_saline_confirmed := FALSE,
  vol_contrast_confirmed := FALSE,
  btn_manual_timeout := BTN_MANUAL_TIMEOUT
]
```
Executable PVS specification of the injection system

The size of the PVS specification is approx. 800 lines.

The PVS record type for the injector system includes 57 state attributes
- 38 attributes for the injector state
- 14 attributes for the workstation state
- 5 attributes for the state of the syringes

The PVS specification includes 33 transition functions
- 26 for modelling the behavior of the injector
- 7 for modelling the workstation

The main focus of the simulation was the injector: this is the reason behind the small number of transition functions used for modelling the workstation
The PVS specification includes the following PVSio-web widgets:

- 33 buttons
- 9 displays
- 5 LEDs
- 2 syringes

Each button widget is linked to a transition function in the PVS model: this is done through the APIs of the PVSio-web widget, which include a parameter for specifying the name of the transition function to be evaluated when a given user action is performed on the widget.

Each display and LED widget is seamlessly associated with a state attribute defined in the PVS specification. The creation of these widgets follows a pattern that is similar to that for button widgets.
Example of button widget

Manual load button widget

```javascript
var sys = {};
sys.btn_manual = new Button("btn_manual", {
    top: 792, left: 210, width: 38, height: 38
}, {
    callback: render
});
```

Button is the widget constructor

The first argument of the constructor is a string defining the widget identifier. The created widget is stored in a field btn_manual of a variable sys. Transition function in the PVS model to be linked to the widget is constructed by concatenating the user action that activates the widget with the widget identifier (e.g., click_btn_manual when the user clicks on the button).

The second argument defines the coordinates and size of the widget.

The third argument provides information about which callback function is to be invoked for refreshing the visual appearance of the prototype.
Example of display widget

LED widget

```javascript
sys.lock_LED = new LED("lock_LED", {
  top:916, left:221, width:13, height:13
}, {
  color: "green"
});
```

LED is the widget constructor

The first argument is the widget identifier

The second argument defines position and size of the widget

The third argument specifies the LED color
Refresh of the visual aspects of widgets

The visual aspect of all widgets is periodically refreshed every time the PVS specification is evaluated.

The evaluation of the specification occurs either when the user interacts with an input widget (e.g., presses a button), or periodically (if the device has internal timers that are ticking). A JavaScript function `render` contains the code for refreshing the widgets.

```javascript
function render(err, event) {
    var res = stateParser.parse(event.data);
    if (res) {
        sys.btn_manual.render(res);
        sys.lock_LED.render(res);
        ...
    }
}
```

In its basic form, the `render` function simply parses the PVS state and invokes the render method of the widgets.
Software model of the device and the console

http://www.pvsioweb.org/demos/stellantV2
Risk of incorrect injection settings

The Volume displays normally report a value corresponding to the volume of liquid loaded in the syringes.

However, in certain operating modes for the injector, the display values have a different meaning.

Information on the front panel of the injector is not always sufficient to discriminate these different cases.

Syringes are not plugged, the display values indicate the maximum value of volume that can be loaded in the syringes;

Syringes are connected but empty, the display values indicate the current position of the syringe plungers;

Button AutoFill on the front panel is pressed, the display values indicate the target volume of liquid that will be loaded in the syringes. This value is reset to 0 as soon as buttons FillA or FillB are pressed.
Risk of undetected air-in-line

For patient safety, it is important to check that air bubbles are purged from syringes and tubing before the injection.

For this reason, the arming phase of the injector is disabled if the CheckAir button available on the front panel of the injector has not been pressed.

However, this button is only a placeholder, i.e., a functionality provided by the device to remind the clinician to verify the absence of air (the injector does not have any sensor for detecting air-in-line).

Pushing the CheckAir button does not trigger any actual check from the device – the clinician needs to look into syringes and tubes and make sure there are no air bubbles. If air bubbles are present, the clinician can use the Prime button to remove the air.
Risk of misprogramming the injector

The injector provides two main modalities for filling syringes, and two modalities for priming: automatic and manual. These two modalities can be *interleaved*.

**Manual mode**: clinicians use the chevron keys to load the volume prescribed by the protocol and prime the syringes.

**Automatic mode**, a single button press on FillA and FillB on the front panel of the injector allows clinicians to load liquid in each syringe, and then a single button press on the Prime button primes the syringes.

The volume of liquid loaded using automatic mode is larger than the volume prescribed by the protocol: +h mL for the contrast, and +k mL for the saline. The automatic prime function pushes exactly h mL of contrast and k mL of saline out of the syringes.

*If clinicians interleave the two modalities* and accidentally omit to check the volume on the injector display, there is a risk of injecting a volume of saline and contrast that is slightly different than the intended values.
Risk of misreading values

This issue concerns the phase in which the injector needs to be connected to the patient to start the injection.

In this phase, the injector needs to be rotated of 180 degrees. The rotation moves air up in the syringes – this is a safety precaution for preventing air being injected in the veins of the patient.

However, the rotation causes the displays provided on the front panel of the injector to be upside-down. This is particularly unfortunate, because the device uses seven-segments displays and certain numbers can be accidentally mis-read when the display is upside-down (e.g., 51 can be misread as 12, a well-known problem with using seven-segments displays in medical devices).
The interactive simulation stimulated a constructive discussion within a multidisciplinary team of engineers and clinicians, about possible design improvements to the device that could prevent the identified critical workflows.

Clinicians played a fundamental role in the identification of critical scenarios, as well as in the description of how the medical device is routinely used in the real-world.

A possible use of the simulation tool is to enhance the proficiency of the clinical users of the injector, helping them to avoid possible traps, thus increasing patient safety.

The simulation facilitated the multidisciplinary work necessary to obtain results that have strong impact and immediate utility to different stakeholders.
5. Prototypes of more complex systems

The case of Integrating Clinical Environments

Functional Elements of the Integrated Clinical Environment
ASTM standard F2761-2009

Clinician

Integrated Clinical Environment (ICE)

ICE Supervisor

External Interface

Network Controller

Data Logger

ICE Interface

Medical Device

ICE Interface

Other Equipment

Patient

2nd UBORA Design School, Pisa - September 3-7, 2018
PVSio-web has been extended to introduce support of automatic generation of user interface prototypes equipped with a standard FMI co-simulation interface.

Heart-pacemaker prototype

Cinzia Bernardeschi, Andrea Domenici, Paolo Masci:

Architecture of the heart model (Chen et al., 2014)

(Chen et al., 2014) T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre, Quantitative verification of implantable cardiac pacemakers over hybrid heart models, Information and Computation, vol. 236, n. 0, 2014.
The pacemaker model as a network of Timed Automata

Modelling patterns have been defined to represent TAs in PVS executable specifications.

LRI: THEORY BEGIN
Mode: TYPE = {LRI, ASed}
state: TYPE = [#
    time: real,
    loc: Mode #]

init_LRI: state = (# time := 0, loc := LRI #)

en_APout(st: state): boolean = % enabling
    loc(st) = LRI AND time(st) >= TLRI-TAVI

APout(st: (en_APout)): state = (# time := 0, loc := LRI #)
................
en_tau(st: state): bool = false % time-checking predicate
tau(st: state): state = st % timing function
END LRI
The prototype

The PVSio-web co-simulation environment integrates the two models:

- The Heart model simulation is started in the Simulink environment
- the PVSio environment is loaded with the PVS theory describing the Pacemaker
- PVSio-web requests the heart model to return heartbeat signals
- PVSio-web sends PVSio a function call with heartbeat signals as arguments
- PVSio computes the pacemaker model response and returns it to PVSio-web
- PVSio-web sends the pacemaker response to the heart model
Simulation

Debugger Monitor

PVSio WebSocket

Simulink WebSocket

```c
)#), vrp:=(# loc:=Idle,
time:=29 #) #), Vget:=0, VP:=0,
vpon:=FALSE, vptime:=0, wclk:=29 #)

<-RECEIVED
[ 10:23:48 GMT+0200 (CEST) ]
(# Aget:=0, AP:=0, apon:=FALSE,
apt ime:=0, dev:=(# avi:=(# clk:=29,
loc:=Idle, time:=30 #),
lri:=(# loc:=LRI, time:=30 #),
pvard:=(# loc:=Idle, time:=30 #),
uri:=(# clk:=30 #),
vrp:=(# loc:=Idle,
time:=30 #) #), Vget:=0, VP:=0,
vpon:=FALSE, vptime:=0, wclk:=30 #)
```

```c
(# Aget:= 0.00, Vget:= 0.00 #)

<-RECEIVED
(# Aget:= 0.00, Vget:= 0.00 #)

<-RECEIVED
(# Aget:= 0.00, Vget:= 0.00 #)

<-RECEIVED
(# Aget:= 0.00, Vget:= 0.00 #)

<-RECEIVED
(# Aget:= 0.00, Vget:= 0.00 #)

<-RECEIVED
(# Aget:= 0.00, Vget:= 0.00 #)
```
The *Prototype Verification System* is an interactive theorem prover developed at SRI International by S. Owre, N. Shankar, J. Rushby, and others.

PVS has a rich **specification language** to define theories.

PVS has many powerful **inference rules** to prove theorems **interactively**.

- A user submits a theorem, then chooses inference rules until the proof ends successfully, or gets stuck.
- Often a single step is sufficient.

PVS has a **simulation extension** (*PVSio*). *PVSio* generates for each function in the declarative PVS language a procedure to compute its value (Lisp). *PVSio* can also compute functions with side effects, such as producing outputs.
Formal verification

Logic specifications model a system by stating its properties in a formal language.

Logic specifications are used for the formal verification of systems, using **automatic theorem proving**.

- System definition (THEORY)
- System properties (THEOREMS)
- Theorem prover
- PROVED!
- NOT PROVED
Formal verification

Formal verification is an important complement to simulation.

E.g., suppose we want to verify that the previously shown ICP system satisfies this property:

*It is always the case that module LRI is in state LRI and its clock is reset when transition AP is executed.*

\[
\text{lri}_\text{ap}: \text{LEMMA} \\
\quad \text{FORALL } (s0, s1: \text{State}): \\
\quad \quad \text{per}_{\text{APout}}(\text{lri}(s0)) \text{ AND } s1 = \text{APout}(s0) \text{ IMPLIES} \\
\quad \quad \quad \text{mode}(\text{lri}(s1)) = \text{LRI} \text{ AND } \text{time}(\text{lri}(s1)) = 0
\]

A single application of the *grind* rule (multiple simplifications) is sufficient.

Rule? (grind)

... 

Q.E.D.

Run time = 0.17 secs.

Formal proof of the deterministic behavior of the pacemaker, in any heart condition.
6. Conclusions

- PVSio-web is an open-source graphical environment for facilitating the design and evaluation of interactive (human-computer) systems.

- PVSio-web has been successfully used for:
  - analyzing commercial, safety-critical medical devices (reverse engineering from the product) to fix existing issues, and identify new potential issues in advance.
  - medical device design, by both formal methods experts and non-technical end users.
  - creating training material for device developers and device users, and
  - raising awareness of criticalities during training sessions.
**Conclusions**

*PVSio-web has been used* for demonstration of user interface issues with medical devices in use at UCLH and in other UK hospitals in the **CHI+MED** (Computer-Human Interaction for Medical Devices, *EP/G059063/1*) project, funded by the **Engineering and Physical Sciences Research**

*PVSio-web has been used* for modelling contrast media injectors at Department of Diagnostic and Interventional Radiology of the University of Pisa, Italy, in the framework of the "Centro interdipartimentale di ricerca in Promozione della Salute e Information Technology" of Pisa (ProSIT).

**US Food and Drug Administration (FDA)** & The Medicines and Healthcare products Regulatory Agency (*MHRA*) *are using these results and trialling these methods on premarket reviews*