

Evaluation of Formal IDEs for Human-Machine Interface Design and Analysis: The Case of CIRCUS and PVSio-web

Camille Fayollas¹ Célia Martinie¹ Philippe Palanque¹ Paolo Masci²

Michael D. Harrison^{2,3} José C. Campos² Saulo Rodrigues e Silva²

¹ICS-IRIT, University of Toulouse, Toulouse, France
{fayollas, martinie, palanque}@irit.fr

²HASLab/INESC TEC and Universidade do Minho, Braga, Portugal
{paolo.masci, jose.c.campos, saulo.r.silva}@inesctec.pt

³Newcastle University, Newcastle upon Tyne, United Kingdom
michael.harrison@newcastle.ac.uk

Critical human-machine interfaces are present in many systems including avionics systems and medical devices. Use error is a concern in these systems both in terms of hardware panels and input devices, and the software that drives the interfaces. Guaranteeing safe usability, in terms of buttons, knobs and displays is now a key element in the overall safety of the system. New integrated development environments (IDEs) based on formal methods technologies have been developed by the research community to support the design and analysis of high-confidence human-machine interfaces. To date, little work has focused on the comparison of these particular types of formal IDEs. This paper compares and evaluates two state-of-the-art toolkits: CIRCUS, a model-based development and analysis tool based on Petri net extensions, and PVSio-web, a prototyping toolkit based on the PVS theorem proving system.

1 Introduction

Use error is a major concern in critical interactive systems. Consider, for example, a pilot calibrating flight instruments before take-off. When calibrating the barometer used to measure the aircraft's altitude, a consistency check should be performed automatically by the cockpit software to help guard against use errors, such as mistyping a value or selecting the wrong units.

New IDEs based on formal methods have been developed by the research community to support the design and analysis of high-confidence human-machine interfaces. Each IDE supports different types of analysis, ranging from functional correctness (e.g., absence of deadlocks and coding errors such as division by zero) to compliance with usability and safety requirements (e.g., assessing the response to user tasks, or the visibility of critical device modes). Choosing the right tool is important to ensure efficiency of the analysis and that the analysis addresses the appropriate safety concerns relating to use. To date, little work has been done to compare and evaluate different formal IDEs for human-machine interface design and analysis, and little or no guidance is available for developers to understand which IDE can be used most effectively for which kind of analysis. This paper describes a first step towards addressing this gap.

Contribution. We compare and evaluate two state-of-the-art formal verification technologies for the analysis of human-machine interfaces: CIRCUS [9], a model-based development and analysis tool that uses Petri net extensions; and PVSio-web [19], a prototyping toolkit based on the PVS theorem prover.

The aim of this work is to provide guidance to developers to understand which tool can be used most effectively for which kind of analysis of interactive systems. Both tools have their foundations in existing formal technologies, but are focused towards particular issues relating to the user interface. The capabilities of the two tools are demonstrated in the paper through a common case study based on a critical subsystem in the cockpit of a large civil aircraft. A taxonomy is developed as a result of the comparison that can be used to describe the characteristics of other similar tools.

Organisation. The remainder of the paper is organised as follows. Section 2 illustrates typical features of formal IDEs for the design and analysis of human-machine interfaces, and presents a detailed description of CIRCUS and PVSio-web. Section 3 introduces the common example for comparison of the selected tools, as well as the developed models. Section 4 presents the metrics for comparing the IDEs, and then uses the metrics as a basis for the comparison. Section 5 concludes the paper and presents future directions in which the tools may evolve.

2 The formal modelling and analysis of user interfaces

Formal tools for the modelling and analysis of human-machine interfaces are designed to support multi-disciplinary teams of designers from different engineering disciplines, including human factors engineering (to establish usability requirements, run user studies and interpret compliance), formal methods (to verify compliance of a system design with design requirements), and software engineering (to develop prototypes and software code, e.g., using model-based development methods). Although several tools provide graphical model editors and automated functions for modelling and analysis of interactive elements of a system, different tools are usually complementary, as they support different levels of description, and different types of analysis, ranging from micro-level aspects of human-machine interaction, e.g., aspect and behaviour of user interface widgets, to the analysis of the wider socio-technical system within which the interactive system is used.

In the present work, we compare two state-of-the-art formal tools developed by two different research teams: CIRCUS [9], a toolkit for model-based development of interactive systems; and PVSio-web [19], a toolkit for model-based development of user interface software. Both build on tools that have been developed more generally for model based design and software engineering, extending them with features that are particularly useful when considering the human-machine interface or the wider socio-technical system.

Other formal tools that could be used (and in some cases have been used) for the analysis of human-machine interfaces exist. They offer functionalities that complement those of CIRCUS and PVSio-web. The evaluation of these other tools is not within the scope of this paper, although a brief overview can be found in Section 2.3.

2.1 CIRCUS

CIRCUS, which stands for Computer-aided-design of Interactive, Resilient, Critical and Usable Systems, is an IDE for the formal verification of the system's behaviour as well as the analysis of compatibility between the user's task and the system's behaviour. CIRCUS includes three tools:

- **HAMSTERS** (Human-centred Assessment and Modelling to Support Task Engineering for Resilient Systems) is a tool for editing and simulating task models. The tool can be used to ensure consistency, coherence, and conformity between assumed or prescribed user tasks and the sequence of actions necessary to operate interactive systems [3]. The notation used in the tool

makes it possible to structure users' goals and sub-goals into hierarchical task trees. Qualitative temporal relationships among tasks are described by operators. Various notational elements support modelling of specialised task types, explicit representations of data and knowledge, device descriptions, genotypes and phenotypes of errors, and collaborative tasks.

- **PetShop** (Petri Net workshop) is a tool for creating, editing, simulating and analysing system models using the ICO (Interactive Cooperative Objects) notation [23, 17]. The ICO notation allows developers to specify the behaviour of interactive systems. The notation uses Petri Nets for describing dynamic behaviours, and uses object-oriented concepts (including dynamic instantiation, classification, encapsulation, inheritance and client/server relationships) to describe structural or static aspects of the system.
- **SWAN** (Synergistic Workshop for Articulating Notations) is a tool for the co-execution of PetShop models and HAMSTERS models [3]. The tool allows developers to establish correspondences between system behaviours and tasks, and perform automated system testing by means of co-execution [5].

2.2 PVSio-web

PVSio-web is a toolkit for model-based development of user interface software. The toolkit is based on and extends an established theorem prover, PVS [24], providing a graphical environment for constructing, visualising and analysing formal models of user interface software. PVSio-web has three main components:

- **Prototype Builder and Simulator.** This tool allows developers to create device prototypes based on formal models, and run them within a Web browser. The visual aspect of the prototype uses an interactive picture of the device. Developers create programmable areas over the picture to identify input widgets (e.g., buttons) and output widgets (e.g., displays, LEDs). The tool automatically translates user actions over input widgets (e.g., button presses) into PVS expressions that can be evaluated within PVSio [22], the native PVS component for animating executable PVS models. The Simulator tool executes PVSio in the background, and the effects of the execution are automatically rendered using the output widgets of the prototype to closely resemble the visual appearance of the real system in the corresponding states.
- **Emucharts Editor.** This tool facilitates the creation of formal models using visual diagrams known as Emucharts. These diagrams are based on Statecharts [14]. The tool allows developers to define the following design elements: states, representing the different modes of the system; state variables, representing the characteristics of the system state; and transitions, representing events that change the system state. The tool incorporates a model generator that translates the Emucharts diagram into executable PVS models automatically. The model generator also supports the generation of other different formal modelling languages for interactive systems, including VDM [18], MAL [6], and PIM [4], as well as executable code (MISRA-C).
- **The PVS back-end.** This includes the PVS theorem prover and the PVSio environment for model animation. The back-end is used for formal analysis of usability-related properties of the human-machine interface model, such as consistency of response to user actions and reversibility of user actions.

2.3 Other tools

MathWorks Simulink [20] is a commercial tool for model-based design and analysis of dynamic systems. It provides a graphical model editor based on Statecharts, and functions for rapid generation of realistic prototypes. SCR [13] is a toolset for formal analysis of system requirements and specifications. Using SCR, it is possible to specify the behaviour of a system formally, use visual front-ends to demonstrate the system behaviour based on the specifications, and use a set of formal methods tools for the analysis of system properties. SCADE and IBM's Rational StateMate are two commercial tool sets for model-based development of interactive systems. The tool sets provide, among other features, rapid prototyping, co-simulation, and automated testing. Formal verification is supported by these tools, but is limited to the analysis of coding errors such as division-by-zero. Use-related requirements and tasks can be analysed only using simulation and testing. IVY [7] is a workbench for formal modelling and verification of interactive systems. The tool provides developers with standard property templates that capture usability concerns in human-machine interfaces. The core of the IVY verification engine is the NuSMV model checker. A graphical environment isolates developers from the details of the underlying verification tool, thereby lowering the knowledge barriers for using the tool. The particular tools that are of interest in the design and analysis of interactive systems enable the analysis of user activities, with a focus on what users do in terms of what they perceive about the systems and the actions they perform. Furthermore an important requirement for such tools is that the means of analysis and their results should be accessible to team members without a background in formal techniques, or even software development techniques.

3 Case study and IDE showcase

The case study for comparing the selected tools is based on a subsystem of the Flight Control Unit (FCU) of the Airbus A380. It is an interactive hardware panel with several different buttons, knobs, and displays. The FCU has two main components: the Electronic Flight Information System Control Panel (EFIS CP), for configuring the piloting and navigation displays; and the Auto Flight System Control Panel (AFS CP), for setting the autopilot state and parameters.

In future cockpits, the interactive hardware elements of the FCU panel might be replaced by an interactive graphical application rendered on displays. This graphical software (hereafter, referred to as FCU Software) will provide the same functionalities as the corresponding hardware elements. This graphical software will be displayed on one of the screens in the cockpit. Pilots will interact with the FCU Software via the Keyboard and Cursor Control Unit (KCCU) that integrates keyboard and track-ball (see Figure 1).

The present paper illustrates how CIRCUS and PVSio-web can be used to create models and prototypes of the FCU Software. Developers can explore design options and analyse requirements for these future generation FCUs using these formal IDEs for model-based development of human-machine interfaces. To keep the example simple, we focus further and analyse the EFIS CP. This component includes most of the fundamental interactive elements of the FCU.

3.1 Description of the system and its use

A close up view of the EFIS CP is shown in the rightmost picture of Figure 1. The left panel of the EFIS CP window is dedicated to the configuration of the barometer settings (upper part) and of the Primary Flight Display (lower part). The right panel is dedicated to the configuration of the Navigation Display. The top part provides buttons for displaying navigation information on the cockpit displays.

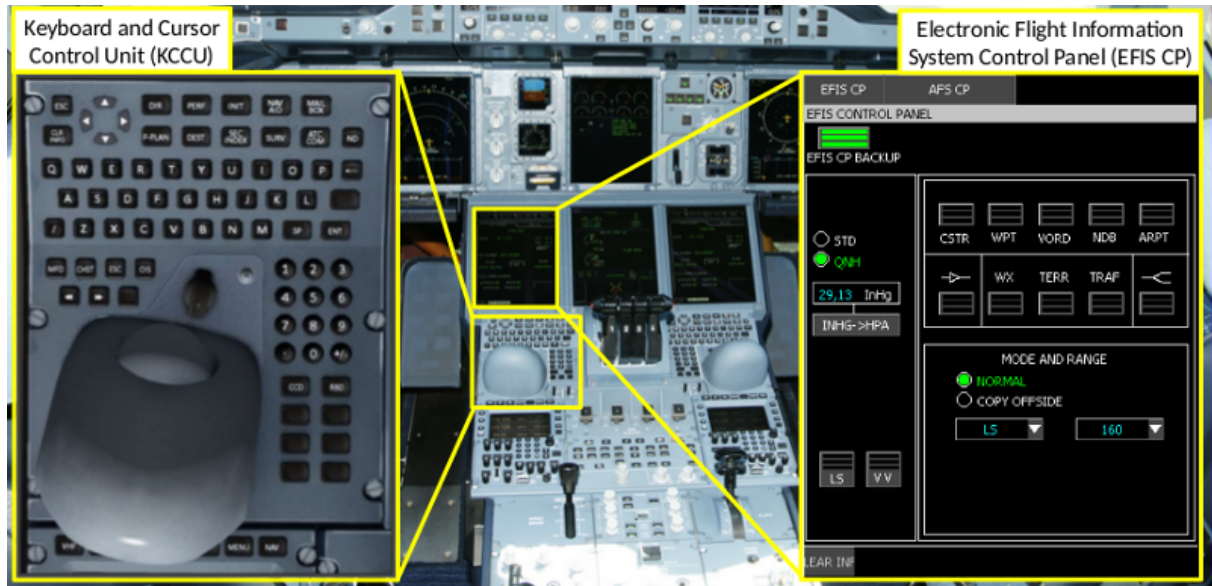


Figure 1: Keyboard and Cursor Control Unit and Flight Control Unit Software.

The ComboBoxes at the bottom allow the pilot to choose display modes and scale.

In this work, we focus more particularly on the configuration of the barometric settings (upper part of the left panel). This panel is composed of several widgets: two CheckButtons enable pilots to select either Standard (STD) or Regional Pressure Settings (QNH) mode. When in QNH mode, a number entry widget (EditBoxNumeric) enables pilots to set the barometric reference. Finally, a button (PushButton) enables pilots to switch the barometer units between inches of mercury (inHg) and hectopascal (hPa). When switching from one unit to the other, a unit conversion is triggered, and the barometer settings value on the display is updated accordingly. When the barometer unit is inHg, the valid range of values is [22, 32.48]. When the unit is hPa, the valid range is [745, 1100]. If the entered value exceeds the valid value range limits, the software automatically adjusts the value to the minimum (when over-shooting the minimum valid value) or the maximum (when overshooting the maximum valid value).

When starting the descent (before landing), pilots may be asked to configure the barometric pressure to the one reported by the airport. The barometric pressure is used by the altimeter as an atmospheric pressure reference in order to process correctly the plane altitude. To change the barometric pressure, pilots select QNH mode, then select the pressure unit (which depends on the airport), and then edit the pressure value in the EditBoxNumeric.

3.2 Modelling and analysis using CIRCUS

A prototype user interface was developed in CIRCUS that captures the functionalities of the FCU Software. The workflow for the modelling and analysis of this interactive system includes six main steps that are briefly detailed below.

Step 1: Task analysis and modelling. This step describes user activities. It identifies goals, tasks, and activities that are intended to be performed by the operator. For example, the task “Perform descent preparation”, is decomposed into several HAMSTERS models represented as subroutines, components and instances of components [10]. Due to space constraints, we only describe an extract of the model in Figure 2.

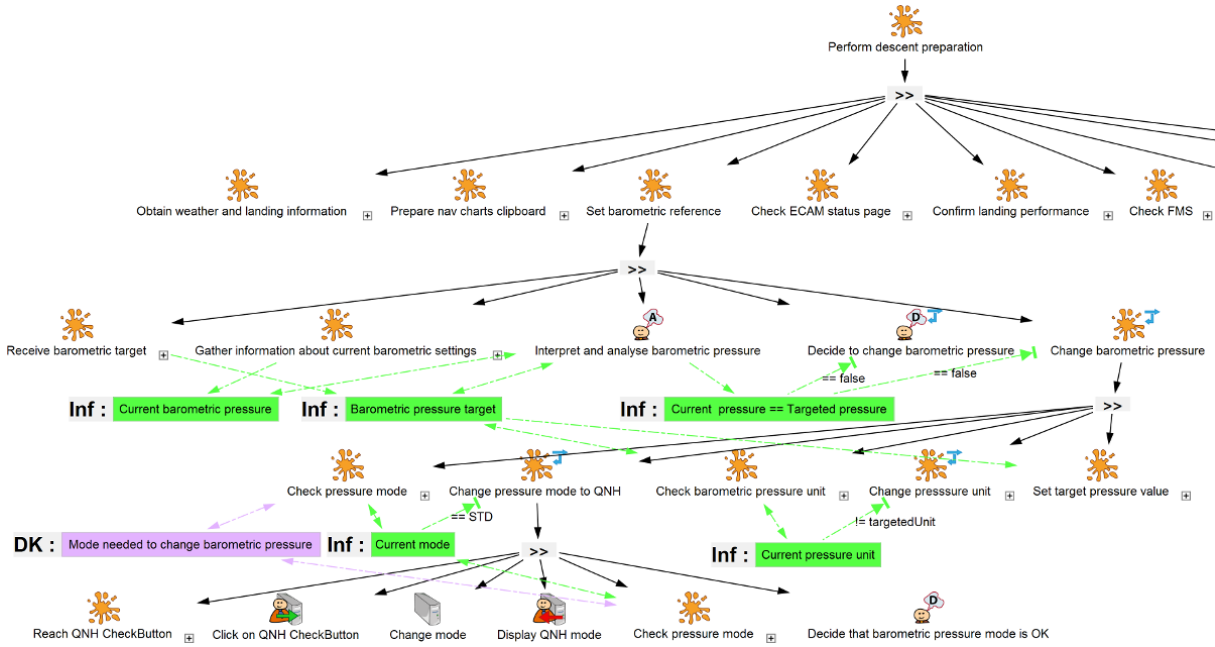


Figure 2: Extract of the task model for the task “Perform descent preparation”.

A simplified version of the task model is described as the abstract task “Perform descent preparation” in the first row of Figure 2. The second row refines this task into several abstract sub-tasks (e.g., “Obtain weather and landing information”). Each one of these abstract tasks corresponds to a step of the operational procedure that is intended to be performed by the flight crew when preparing the descent phase [1].

In the present paper, we focus on the “Set barometric reference” abstract task, refined in the third row. The task is decomposed as follows: the pilot receives the new barometric target (“Receive barometric target” abstract task) and remembers the corresponding information (“Barometric pressure target”). The pilot then needs to gather information about the current barometer settings (“Gather information about current barometric settings”), thus remembering a new piece of information (“Current barometric pressure”). The pilot then needs to compare the two values that have been received in the previous two steps (“Interpret and analyse barometric pressure” cognitive analysis task) creating the “Current pressure == Targeted pressure” information. If the targeted pressure is different from its current value the pilot decides to change the pressure (“Decide to change barometric pressure” cognitive decision task) and change it (“Change barometric pressure” abstract task).

The fourth row of Figure 2 refines the “Change barometric pressure” abstract task as follows: the pilot must first check the pressure mode (“Check pressure mode” abstract task), switch to QNH mode if the current mode is STD (“Change pressure mode to QNH” abstract task), check the unit and change it if needed (“Check barometric pressure unit” and “Change pressure unit” abstract tasks) and finally set the new pressure value (“Set target pressure value” abstract task).

Finally, the fifth row refines the “Change pressure mode to QNH” abstract task. This task is performed by the pilot if the current mode is STD. In this task, the pilot first reaches the QNH checkButton (“Reach QNH CheckButton” abstract task). Then s/he clicks on it (“Click on QNH CheckButton” interactive input task). The system then changes the mode and displays the new state (“Change mode” system task and “Display QNH mode” interactive output task). The pilot can then check that the new pressure

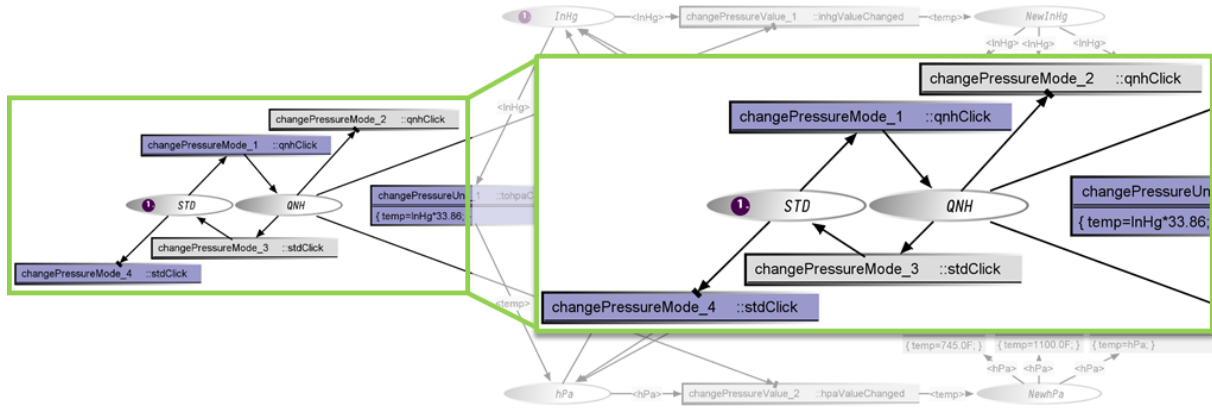


Figure 3: ICO model of the barometer settings behaviour.

mode is the good one (“Check pressure mode” abstract task and “Decide that barometric pressure mode is OK” cognitive decision task). It is important to note that this task model is detailed both in terms of user task refinement (e.g., cognitive task analysis) to allow the analysis of workload and performance (see Step 3 below); and in terms of interactive task refinement (see, for instance, the refinement of the “Change pressure mode to QNH” abstract task which includes the “Click on QNH Checkbutton” interactive input task) to allow the compatibility assessment between the task model and the behavioural model of the system (see Step 6 below).

Step 2: Workload and performance analysis. As presented in Figure 2, the HAMSTERS notation and tool enable human task refinement. It makes it possible to differentiate between cognitive, motor, and perception tasks as well as representing the knowledge and information needed by the user to perform a task. The refinement allows the qualitative analysis of user tasks, workload and performance. For example, the number of cognitive tasks and information that pilots need to remember may be effective indicators for assessing user workload [9]. This kind of analysis can be used to reason about automation design [16].

Step 3: User interface look and feel prototyping. This step aims at developing the user interface look and feel. A result of this step is described in the screen-shot of the EFIS CP presented in Figure 1. The widgets are organised in a style that is compatible with the library defined by the ARINC 661 standard [2].

Step 4: User interface formal modelling. The behaviour of the user interface is specified using ICO models. The behaviour of the barometer settings part of the EFIS CP user interface is represented by the ICO model presented in Figure 3. The left part of this model (that has been enlarged) is dedicated to the pressure mode. As described in Section 3.1, the pressure mode can be in two different mutually exclusive states: STD and QNH. The user can switch from one mode to the other one by clicking either STD or QNH CheckButton (clicking on a CheckButton while already in the corresponding mode is also possible but will have no impact on the pressure mode). This behaviour is defined by the enlarged part of the ICO model presented in Figure 3. The state of the pressure mode is represented by the presence of a token within “QNH” or “STD” places (in Figure 3, place “STD” holds a token meaning that the current pressure mode is STD). Transitions “changePressureMode_1” and “changePressureMode_2” (resp. “changePressureMode_3” and “changePressureMode_4”) correspond to the availability of the “qnhClick” (resp. “stdClick”) event: when one of these two transitions is enabled, the “qnhClick” event is available (thus enabling the QNH CheckButton). The “changePressureMode_1” transition therefore makes it possible to switch from STD pressure mode to QNH pressure mode as a result of clicking on

the QNH CheckButton. The “changePressureMode_2” transition allows the user to click on the QNH CheckButton while in QNH pressure mode without any impact on the pressure mode. The right part of the ICO model presented in Figure 3 (behind the enlarged part of the model) allows pressure to be changed. While we only present here a part of the ICO model describing the behaviour of the EFIS user interface, it is important to note that the ICO notation has also been used to describe the behaviour of the widgets and the window manager. The ICO models can be validated using the simulation feature.

Step 5: Formal analysis. The PetShop tool provides the means to analyse ICO models by the underlying Petri net model [26] using static analysis techniques as supported by the Petri net theory [25]. The ICO approach is based on high level Petri nets. As a result the analysis approach builds on and extends these static analysis techniques. Analysis results must be carefully taken into account by the analyst as the underlying Petri net model can be quite different from the ICO model. Such analysis has been included in CIRCUS and can be interleaved with the editing and simulation of the model, thus helping to correct it in a style that is similar to that provided by spell checkers in modern text editors [8]). It is thus possible to check well-formedness properties of the ICO model, such as absence of deadlocks, as well as user interface properties, either internal properties (e.g., reinitiability) or external properties (e.g., availability of widgets). Note that it is not possible to express these user interface properties explicitly — the analyst needs to express these properties as structural and behavioural Petri net properties that can be then analysed automatically in PetShop.

The analysis of the enlarged part of the ICO model presented in Figure 3 allows developers to check that, whatever action is taken, the pair of places “STD” and “QNH” will always hold one (and only one) token, exhibiting the mutual exclusion of the two states. Transitions connected to these places correspond to the availability of two events “qnhClick” and “stdClick”, and therefore it can be demonstrated that these events will remain available whatever action is triggered. Lastly, there are two transitions in the model that correspond to the event “qnhClick” (transitions “changePressureMode_1” and “changePressureMode_2”). This could potentially lead to non-determinism in the model. However, as “changePressureMode_1” has place “STD” as input place and “changePressureMode_2” has “QNH” place as input place, non-determinism is avoided due to the mutual exclusive marking of these places.

Step 6: Compatibility assessment between task models and user interface models. This step aims at guaranteeing that the task model and the formal model of the user interface behaviour are complete and consistent together (thus helping to guarantee that procedures followed by the operators are correctly supported by the system). A correspondence editing panel is used to establish the matching between interactive input tasks (from the task model) with system inputs (from the system model) and between system outputs (from the system model) with interactive output tasks (from the task model). The co-execution part of the SWAN tool provides support for validation as it makes it possible to find inconsistencies between the two models, e.g., sequences of user actions allowed by the system model and forbidden by the task model, or sequences of user actions that should be available but are not because of inadequate system design. The SWAN tool also provides support for automated scenario-based testing of an interactive application [5]

3.3 Modelling and analysis using PVSio-web

The focus of the PVSio-web analysis is the interaction logic of the EFIS data entry software. Here, we describe the modelling and analysis workflow supported by the tool, and highlight the main characteristics of the developed models (the full description is included as an example application in the PVSio-web tool distribution [19]).

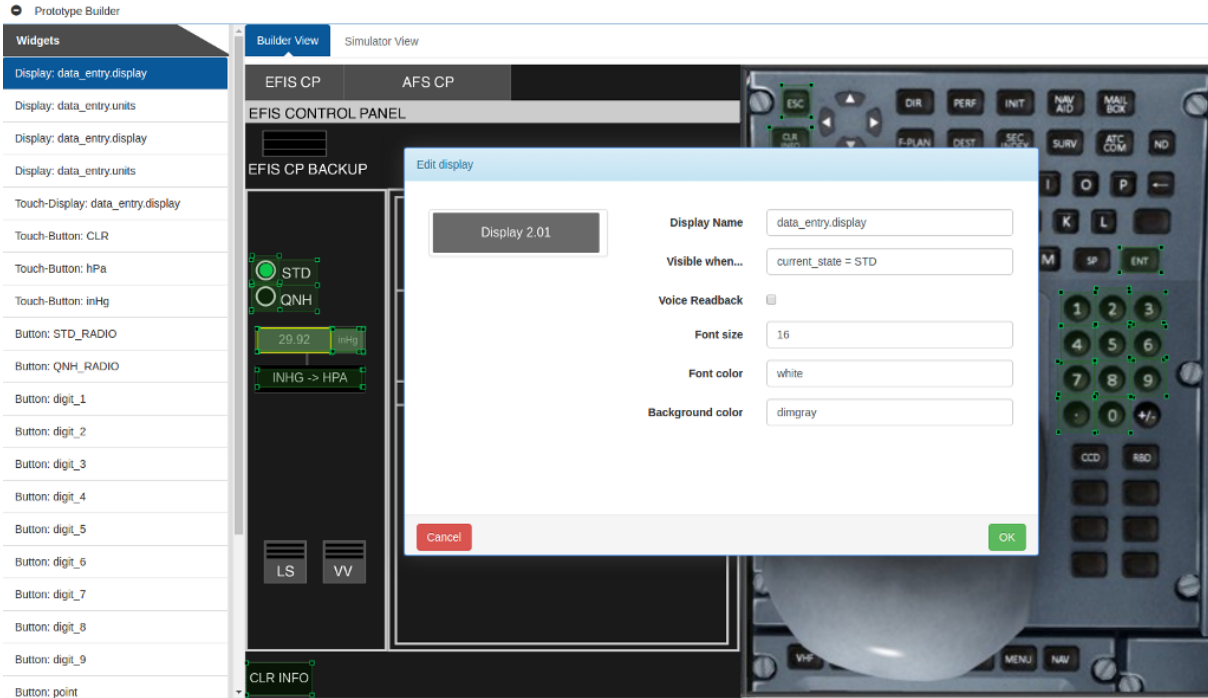


Figure 4: FCU Software prototype developed in the PVSio-web Prototype Builder. Shaded areas over the picture identify interactive system elements.

Step 1: Define the visual appearance of the prototype. The visual aspect of the prototype is based on a picture of the EFIS panel. The PVSio-web Prototype Builder is used to create the visual appearance of the prototype and is defined using the Prototype Builder. A picture of the EFIS panel and KCCU are loaded in the tool, and interactive areas are created over relevant buttons and display elements (see Figure 4). Fifteen input areas were created over the picture of the Keyboard and Cursor Control Unit, to capture user actions over the number pad keys, as well as over other data entry keys (ENT, CLR, ESC, and the units conversion button). Four display elements were created for rendering relevant status variables of the PVS model: a touchscreen display element handles user input on the `EditTextNumeric` for entering the barometer pressure value; two display elements render the STD and QNH CheckButtons; an additional display element renders the pressure units.

Step 2: Define the behaviour of the prototype. The prototype is driven by a PVS model that includes an accurate description of the following features of the system: the modal behaviour of the data entry system; the numeric algorithm for units conversion; the logic for interactive data entry; and the data types used for computation (double, integer, Boolean). Modelling patterns were used to guide the development of the models (some of these patterns are described in [11]). The model was developed using, in combination, the PVSio-web Emucharts Editor and the Emacs-based model editor of PVS. The Emucharts editor allowed us to create a statechart-based diagram that can be automatically translated into PVS models. The Emacs-based model editor was used to build a library function linked to the Emucharts diagram to improve modelling efficiency. The developed Emucharts (shown in Figure 5) includes the following elements: 3 states (STD, QNH, and EDIT PRESSURE) representing three different modes of operation; 25 transitions, representing the effect of user actions on the Keyboard and Cursor Control Unit when adjusting the barometer settings, and internal timers handling timeouts due to inactivity during interactive

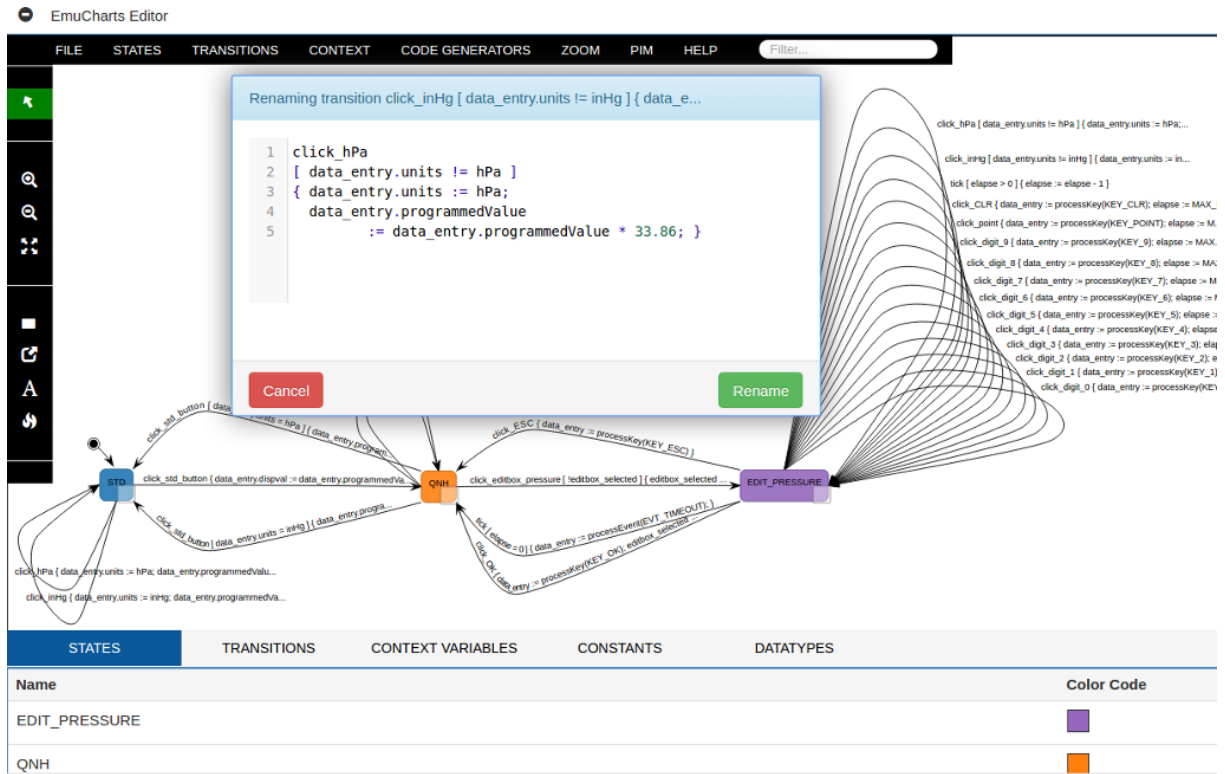


Figure 5: Emucharts model of the FCU Software created in the Emucharts Editor.

data entry; and 9 status variables, representing the state of the system (units, display value, programmed value, etc.). The library function, `ProcessKey`, is used within the Emucharts diagrams to define the effect on state variables of transitions associated with key presses.

Step 3: Model validation. This analysis ensures internal consistency of the model, as well as checking accuracy with respect to the real system. Internal consistency is assessed by discharging proof obligations (called type-check-conditions) automatically generated by the PVS theorem prover. These proof obligations check coverage of conditions, disjointness of conditions, and correct use of data types. For the developed Emucharts model, PVS generated 22 proof obligations, all of which were discharged automatically by the PVS theorem prover. Accuracy of the model is assessed by using the prototype to engage with Airbus cockpit experts. Experts can press buttons of the prototype, and watch the effect of interactions on the prototype displays. By this means it is possible to check that the prototype behaviour resembles that of the real system.

Step 4: Formal analysis. The prototype and the PVS theorem prover are used in combination to analyse the model. The prototype is used to perform a lightweight formal analysis suitable to establish a common understanding within a multidisciplinary team of the correct interpretation of safety requirements and usability properties. This analysis consists in the execution of sample input key sequences demonstrating scenarios where a given requirement is either satisfied or fails. This initial lightweight analysis based on test cases is extended to a full formal analysis using the PVS theorem prover, to check that requirements and properties of the model are satisfied for all input key sequences in all reachable model states. To perform this full analysis, PVS theorems need to be defined that capture the requirements and properties. They are expressed using structural induction and a set of templates described in [12]. An example

property that can be analysed is *consistency of device response to user actions*. The consistency property is motivated by the fact that users quickly develop a mental model that embodies their expectations of how to interact with a user interface. Because of this, the overall structure of a user interface should be consistent in its layout, screen structure, navigation, terminology, and control elements. Example consistency properties are: a designated set of function buttons always change the mode; a further set of keys, for example concerned with number entry, will always change the barometric variable relevant to the mode but do not change the mode; an *enter* key always changes the relevant parameter when in the relevant mode; an *escape* key ensures that the value set in the mode is discarded and the barometric value reverts to the value it had when it entered the mode.

In PVS, the consistency template is formulated as a property of a group of actions $A_c \subseteq \wp(S \rightarrow S)$, or it may be the same action under different modes, requiring that all actions in the group have similar effects on specific state variables selected using a filter. The relation *consistent* : $C \times C \rightarrow T$ connects a filtered state, before an action occurs (captured by *filter_pre* : $S \times MS \rightarrow C$), with a filtered state after the action (captured by *filter_post* : $S \times MS \rightarrow C$). The description of the filters and the *consistent* relation specify the consistency across states and across actions. Here *MS* is defined to be a set of modes. Two modes are relevant here. A set of modes not defined includes the mode that allows the entry of the barometer value. Within the barometer entry mode are two modes that relate to the different units that can be used to enter the barometric values, defined as: inHg and hPa. A general notion of consistency assumes that the property is restricted to a set of states focused by a guard: *guard* : $S \times MS \rightarrow T$. This guard may itself be limited by a mode. The general consistency template can therefore be expressed as:

Consistency

$$\forall a \in A_c \subseteq \wp(S \rightarrow S), s \in S, m \in MS : \\ \text{guard}(s, m) \wedge \text{consistent}(\text{filter_pre}(s, m), \text{filter_post}(a(s), m))$$

Two examples are now used to illustrate the use of the consistency template. The first is that a set of actions never change the barometric entry mode. The *pre_filter* and *post_filter* both extract the barometric entry mode, and are of the form *filter_baro*(st: state): UnitsType = Units(st). This property relates directly to modes and therefore the mode parameter can be omitted in the filter definition. The set of actions determined by *state_transitions_actions* which encompasses the set of transitions as determined by the enabled actions in the barometric mode. In summary, the *consistent* relation in this case is equality and the theorem that instantiates the consistency template is:

```
modeinvariant: THEOREM FORALL (pre, post: state):
state_transitions_actions(pre, post) => (filter_baro(pre) = filter_baro(post))
```

On the other hand the action *click_hPa* always changes the entry mode. So here *consistent* is inequality.

```
alwayschgmode: THEOREM
  FORALL (pre, post: state):
    (post = click_hPa(pre) AND guard_baro(pre))
    => filter_baro(pre) /= filter_baro(post)
```

4 Tool comparison

In this section, we first present the criteria that were identified to compare the characteristics and functionalities of the two tools. These criteria form a basis for the comparison of these two tools.

4.1 Comparison criteria

We identified 22 criteria suitable to compare the characteristics and functionalities of the two tools. These criteria are general, and can be used as a reference to define a taxonomy suitable to classify and compare other similar formal IDEs for user interface modelling and analysis. These criteria are divided in four categories:

- **General aspects of the tools**

1. Scope/purpose of the tool within the development process, e.g., requirements analysis, prototyping, verification.
2. Tool features, e.g., modelling of user tasks and goals, analysis of usability properties, simulation of user tasks.
3. Tool extensibility, e.g., to model systems from different application domains, or to perform a different type of analysis
4. Prerequisites and background knowledge, e.g., distributed systems, object oriented languages, Petri Nets, task modelling, PVS.
5. IDE instance and principle, e.g., Eclipse plugin, Web, Netbeans API
6. IDE availability, e.g., snapshot, demo, downloadable, open source.

- **Modelling features**

7. Notation names, e.g., ICO, HAMSTERS, Emucharts, PVS.
8. Notation instance, e.g., Petri Net, state machines, higher-order logic.
9. Notation paradigm, e.g., event-based, state-based, declarative.
10. Structuring models, e.g., object-oriented, functional, component-based.
11. Model editing features, e.g., textual, visual, autocompletion support.
12. Suggestions for model improvements, e.g., strengthening of pre-conditions.

- **Prototyping features**

13. Support for prototype building, e.g., visual editor, library of widgets.
14. Execution environment of the prototype, e.g., Java virtual machine, Javascript execution environment.
15. User interface testing, e.g., automatic generation of input test cases.
16. Human-machine interaction techniques, e.g., Pre-WIMP (input dissociated from output), WIMP, post-WIMP, tangible, multimodal.
17. Code generation, e.g., C, C++, Java.

- **Analysis of human-machine interaction**

18. Verification type, e.g., functional verification, performance analysis, hierarchical task analysis;
19. Verification technology, e.g., theorem proving, static analysis.
20. Scalability of the analysis, e.g., illustrative examples, industrial size.
21. Support for the analysis of the wider socio-technical system.
22. Related development process, e.g., user centered design, waterfall development process, agile development.

4.2 CIRCUS and PVSio-web comparison

In this section, we discuss, following the four categories of criteria identified above, the comparison of CIRCUS and PVSio-web. A detailed assessment of all the criteria presented above is presented in tabular form in the Appendix.

General aspects of the tools. From a high-level perspective, the scope of CIRCUS and PVSio-web is the formal development of user interfaces. Both tools support modelling and analysis of the interaction logic of the user interface software. However, the two tools offer different modelling and analysis technologies that are tailored to support two different (and complementary) styles of assessment of user interfaces. CIRCUS supports explicit modelling of user tasks and goals, allowing developers to simulate user tasks and check their compatibility with the interactive behaviour of the system. PVSio-web, on the other hand, supports explicit modelling of general usability and safety properties, facilitating the assessment of compliance of a user interface design with design guidelines and best design practices (e.g., according to standards or regulatory frameworks). Whilst a certain level of background knowledge is needed to use the tools effectively, basic knowledge about Petri nets and task models (for CIRCUS) and state machines and state charts (for PVSio-web) is already sufficient to get started with illustrative examples. This is extremely useful to reduce the typical knowledge barriers faced by novice users. The two IDEs are developed using standard technologies supported by multiple platforms (Netbeans Visual API for CIRCUS, Web technologies for PVSio-web), and can be executed on any standard desktop/laptop computer.

Modelling features. Both tools provide powerful graphical IDEs designed to assist developers in the creation of formal models. CIRCUS uses specialised graphical notations and diagrams: the ICO notation is used for building system models; the HAMSTERS notation is used for describing user tasks. ICOs are based on object-oriented extensions to Petri nets, and support both event-based and state-based modelling. HAMSTERS is a procedural notation. The complexity of models is handled using information hiding (as in object-oriented programming languages), and component-based model structures. This facilitates the creation of complex models, as well as the implementation of editing features that are important for developers, such as auto-completion of models and support for parametric models. The use of specific notations, however, limits the ability of developers to import external models created with other tools, or export CIRCUS models to other tools. PVSio-web, on the other hand, uses modelling patterns to support the modelling process. Developers can use either a graphical notation (Emucharts diagrams, or a textual notation (PVS higher-order logic), or a combination of both, to specify the system model. This has many benefits: software developers that are familiar with Statecharts can build models using a language that is familiar for them, and gradually learn PVS modelling by examples, checking how the Emucharts model translates into PVS; Emucharts models can be translated into popular formal modelling languages different than PVS (e.g., VDM); expert PVS users can still develop entire models using PVS higher-order logic only, and software developers can import these PVS models as libraries, thus facilitating model re-use. The main drawback is that the current implementation of Emucharts lacks mechanisms for model optimisation (e.g., a battery of similar PVS functions is generated instead of a single function with a parameter), and technical skills are necessary to understand model improvements suggested by the tool (through the PVS type-checker).

Prototyping features. Both IDEs provide a visual editor for rapid generation of prototypes supporting a range of interaction styles, including: graphical user interfaces with windows, icons, menus, and pointer (WIMP); user interfaces with physical buttons (pre-WIMP); touchscreen-based user interfaces (post-WIMP); and multi-modal user interfaces (e.g., providing both visual and auditory feedback). Both tools promote the use of the Model-View-Controller (MVC [15]) paradigm, with a clear separation between

the visual appearance of the prototype and the logic behaviour. Whilst prototypes developed with the two IDEs share these similarities, prototype building and implementation is substantially different in the two IDEs. CIRCUS prototypes are developed in Java (for their visual appearance) and in ICO models (for their behaviour). Developers can define their own widgets library. For example, for the case study presented in Section 3, we created a library of widgets whose visual aspect and behaviour is compatible with that described in the ARINC 661 standard. PVSio-web prototypes are developed in JavaScript, and their behaviour is defined by a PVS executable model. Rapid prototyping is enabled by a lightweight building process where the visual aspect of the prototype is defined by a picture of the real device, virtually reducing to zero the time and effort necessary to define the visual appearance of the prototype. Initial support for code generation is also available for MISRA-C, for behavioural models developed using Emucharts [21]. A specialised tool (Prototype Builder) is provided with the IDE, to facilitate the identification of interactive areas over the picture, and to link these areas to the PVS model. The current implementation of the Prototype Builder supports only the definition of push buttons and digital display elements, and developers need to edit a JavaScript template manually to introduce more sophisticated widgets (e.g., knobs, graphical displays, etc.). Integration of these more sophisticated widgets in the Prototype Builder is currently under development.

Analysis of human-machine interaction. Multiple verification technologies are used in the two IDEs to enable the efficient analysis of human-machine interaction. Both tools build on established formal methods technologies, and enable lightweight formal analysis based on simulation and testing. CIRCUS implements static analysis techniques from Petri nets theory to perform automatic analysis of well-formedness properties of the model (absence of deadlocks, token conservation), and of basic aspects of the interactive system design (e.g., reinitiability of the user interface and availability of widgets). Simulation is used for functional analysis and quantitative assessment of the system. Either direct interaction with the prototype and automated execution of task models can be used during simulations. Properties verified by this means include: compliance with task models; statistics about the total number of user tasks, and estimation of the cognitive workload of the user based on the types of human-machine interactions necessary to operate the system. PVSio-web uses the standard PVS theorem proving system to analyse well-formedness properties of the model (coverage of conditions, disjointness of conditions, and correct use of data types). Usability and safety requirements can be verified using both lightweight formal verification and full formal verification. Lightweight verification is based on interactive simulations with the prototypes. User interactions can be recorded and used later as a basis for automated testing in a way similar to the way task models are used in CIRCUS. Full formal verification is carried out in the PVS theorem prover, and is partially supported by property templates capturing common usability and safety requirements described in the ANSI/AAMI/IEC HF75 usability standard. Although the full formal analysis is in general not fully automatic, the combined use of property templates and modelling patterns usually leads to proof attempts where minimal human intervention is necessary to guide the theorem prover (typically, for case-splitting and instantiation of symbolic identifiers). Proof tactics for full automatic verification of a standard battery of property templates are currently under development. Dedicated front-ends presenting verification results in a form accessible to human factors specialists are also being investigated.

5 Conclusion and perspectives

In this paper, we presented a first step towards providing guidance to developers to understand which formal tool can be used most effectively for which kind of analysis of interactive systems. This is

achieved through the identification of 22 criteria enabling the characterisation of IDEs for interactive systems formal prototyping and development. These criteria have been used to compare two state-of-the-art formal tools developed by two different research teams: CIRCUS, a toolkit for model-based development of interactive systems; and PVSio-web, a toolkit for model-based development of user interface software based on the PVS theorem proving system. In order to assess all the criteria, we modelled and analysed a case study from the avionics domain using these two tools. The result of this comparison led to the conclusion that the two studied tools are complementary rather than competitive tools. Whilst they have roughly the same scope (formal development of user interfaces), these two tools enable different kinds of modelling and analysis. For instance, CIRCUS supports explicit modelling of user tasks and goals, allowing developers to simulate user tasks and check their compatibility with the interactive behaviour of the system while PVSio-web supports explicit modelling of general usability and safety properties, facilitating the assessment of compliance of a user interface design with design guidelines and best design practices. These two analysis styles are complementary, and both provide important insights about how to develop high-confidence user interfaces. Based on this understanding, we are now developing means to integrate the two IDEs, to enable new powerful analysis features, such as automated scenario-based testing of user interfaces [5]. The envisioned integration is introduced at two levels: at the modelling level, developing PVSio-web extensions for importing/translating HAMSTERS task models into PVS models and properties; and at the simulation level, building CIRCUS extensions for co-execution of task models and PVSio-web prototypes. Additional extensions under development for the two toolkits include: modelling patterns for describing human-machine function allocation; proof tactics and complementary use of different verification technologies for improved automation of usability and safety properties; innovative front-ends for inspecting formal proofs supporting safety and usability claims of user interfaces; and widgets libraries for different application domains.

Acknowledgment. This work is partially supported by: Project NORTE-01-0145-FEDER-000016, financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF); Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) PhD scholarship.

References

- [1] SAS Airbus (2016): *Airbus A380 Flight Crew Operating Manual*. <http://www.airbus.com/>.
- [2] Airlines Electronic Engineering Committee (2002): *ARINC 661 specification: Cockpit Display System Interfaces To User Systems*. Aeronautical Radio Inc.
- [3] Eric Barboni, Jean-François Ladry, David Navarre, Philippe Palanque & Marco Winckler (2010): *Beyond Modelling: An Integrated Environment Supporting Co-execution of Tasks and Systems Models*. In: *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '10, ACM, pp. 165–174, doi:10.1145/1822018.1822043.
- [4] Judy Bowen & Steve Reeves (2013): *Modelling Safety Properties of Interactive Medical Systems*. In: *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, ACM, pp. 91–100, doi:10.1145/2494603.2480314.
- [5] José C. Campos, Camille Fayollas, Célia Martinie, David Navarre, Philippe Palanque & Miguel Pinto (2016): *Systematic Automation of Scenario-based Testing of User Interfaces*. In: *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '16, ACM, New York, NY, USA, pp. 138–148, doi:10.1145/2933242.2948735.

- [6] José C. Campos & Michael D. Harrison (2001): *Model Checking Interactor Specifications*. *Automated Software Engineering*, 8(3-4), pp. 275–310, doi:10.1023/A:1011265604021.
- [7] José C. Campos & Michael D. Harrison (2009): *Interaction Engineering Using the IVY Tool*. In: *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '09, ACM, pp. 35–44, doi:10.1145/1570433.1570442.
- [8] Camille Fayollas, Célia Martinie, Philippe Palanque, Eric Barboni, Racim Fahssi & Arnaud Hamon (In Press, 2016): *Exploiting Action Theory as a Framework for Analysis and Design of Formal Methods Approaches: Application to the CIRCUS Integrated Development Environment*. In: *Formal Methods in Human Computer Interaction*, Springer.
- [9] Camille Fayollas, Célia Martinie, Philippe Palanque, Yannick Deleris, Jean-Charles Fabre & David Navarre (2014): *An Approach for Assessing the Impact of Dependability on Usability: Application to Interactive Cockpits*. In: *Proceedings of the 2014 Tenth European Dependable Computing Conference*, EDCC '14, IEEE Computer Society, pp. 198–209, doi:10.1109/EDCC.2014.17.
- [10] Peter Forbrig, Célia Martinie, Philippe Palanque, Marco Winckler & Racim Fahssi (2014): *Rapid Task-Models Development Using Sub-models, Sub-routines and Generic Components*. In: *Human-Centered Software Engineering: 5th IFIP WG 13.2 International Conference, HCSE 2014, Paderborn, Germany, September 16-18, 2014. Proceedings*, Springer Berlin Heidelberg, pp. 144–163, doi:10.1007/978-3-662-44811-3_9.
- [11] Michael D. Harrison, José C. Campos & Paolo Masci (2016): *Patterns and templates for automated verification of user interface software design in PVS*. Technical Report, Newcastle University. Available at <http://www.ncl.ac.uk/computing/research/publication/225438>.
- [12] Michael D. Harrison, Paolo Masci, José C. Campos & Paul Curzon (In Press, 2016): *The specification and analysis of use properties of a nuclear control system*. In: *Formal Methods in Human Computer Interaction*, Springer.
- [13] Constance Heitmeyer, James Kirby, Bruce Labaw & Ramesh Bharadwaj (1998): *SCR: A toolset for specifying and analyzing software requirements*. In Alan J. Hu & Moshe Y. Vardi, editors: *Computer Aided Verification: 10th International Conference, CAV'98*, 1427, Springer Berlin Heidelberg, pp. 526–531, doi:10.1007/BFb0028775.
- [14] Ian Horrocks (1999): *Constructing the User Interface with Statecharts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [15] Glenn E. Krasner & Stephen T. Pope (1988): *A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80*. *Journal of Object Oriented Programming* 1(3), pp. 26–49. Available at <http://dl.acm.org/citation.cfm?id=50757.50759>.
- [16] Célia Martinie, Philippe Palanque, Eric Barboni & Martina Ragosta (2011): *Task-model based assessment of automation levels: application to space ground segments*. In: *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, IEEE, pp. 3267–3273, doi:10.1109/ICSMC.2011.6084173.
- [17] Célia Martinie, Philippe Palanque & Marco Winckler (2011): *Structuring and Composition Mechanisms to Address Scalability Issues in Task Models*. In: *Human-Computer Interaction – INTERACT 2011: 13th IFIP TC 13 International Conference, 2011, Proceedings, Part III*, Springer Berlin Heidelberg, pp. 589–609, doi:10.1007/978-3-642-23765-2_40.
- [18] Paolo Masci, Peter G. Larsen & Paul Curzon (2015): *Integrating the PVSio-web modelling and prototyping environment with Overture*. In: *13th Overture Workshop, satellite event of FM2015*, Grace Technical Reports, Grace-TR 2015-06, pp. 33–47. Available at <http://grace-center.jp/wp-content/uploads/2012/05/13thOverture-Proceedings.pdf>.
- [19] Paolo Masci, Patrick Oladimeji, Yi Zhang, Paul Jones, Paul Curzon & Harold Thimbleby (2015): *PVSio-web 2.0: Joining PVS to HCI*. In Daniel Kroening & S. Corina Păsăreanu, editors: *Computer Aided Verification: 27th International Conference, CAV 2015, Proceedings, Part I*, Springer International Publishing, pp. 470–478, doi:10.1007/978-3-319-21690-4_30. Tool available at <http://www.pvsioweb.org>.
- [20] MathWorks: *Mathworks Simulink*. <http://www.mathworks.com/products/simulink>.

- [21] Gioacchino Mauro, Harold Thimbleby, Andrea Domenici & Cinzia Bernardeschi (2016): *Extending a user interface prototyping tool with automatic MISRA C code generation*. In: *3rd Workshop on Formal Integrated Development Environment (F-IDE)*, satellite workshop of *Formal Methods 2016*, Electronic Proceedings in Theoretical Computer Science (EPTCS).
- [22] César A Muñoz & Ricky Butler (2003): *Rapid prototyping in PVS*. Available at <http://ntrs.nasa.gov/search.jsp?R=20040046914>. NASA/CR-2003-212418, NIA Report No.2003-03.
- [23] David Navarre, Philippe Palanque, Jean-Francois Ladry & Eric Barboni (2009): *ICOs: A Model-based User Interface Description Technique Dedicated to Interactive Systems Addressing Usability, Reliability and Scalability*. *ACM Transactions on Computer-Human Interaction (TOCHI)* 16(4), pp. 18:1–18:56, doi:10.1145/1614390.1614393.
- [24] Sam Owre, John M. Rushby & Natarajan Shankar (1992): *PVS: A Prototype Verification System*. In: *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction, CADE-11*, Springer Berlin Heidelberg, pp. 748–752, doi:10.1007/3-540-55602-8_217.
- [25] James Lyle Peterson (1981): *Petri Net Theory and the Modeling of Systems*. Prentice Hall.
- [26] José-Luis Silva, Camille Fayollas, Arnaud Hamon, Célia Martinie, Eric Barboni et al. (2014): *Analysis of WIMP and Post WIMP Interactive Systems based on Formal Specification*. *Electronic Communications of the EASST* 69, doi:10.14279/tuj.eceasst.69.967.

Appendix

Formal IDE	CIRCUS	PVSio-web
1. Scope/purpose	Interactive system prototyping, development, and analysis.	User interface software prototyping and analysis.
2. Tool features	User task and goals description, interaction logic (dialog) and interaction techniques modelling, interactive system prototyping, support for verification of properties, assessment of compatibility between user tasks and interactive system prototype.	Interaction logic modelling, rapid prototyping of user interface software, verification of safety requirements and usability properties, code generation and documentation.
3. Tool extensibility	Each tool within CIRCUS offers an API supporting connection to other computing systems. For instance, connecting PetShop execution engine to cockpit software simulators or connecting Petri net analysis tools to PetShop analysis module.	PVSio-web has a plug-in based architecture that enables the rapid introduction of new modelling, prototyping, and analysis tools; support for new widgets types and widgets libraries can be introduced in Prototype Builder; Emucharts Editor can be extended with new model generators and code generators.
4. Background knowledge	Object-Oriented Petri Nets (for Petshop), Java programming, distributed systems, hierarchical task modelling.	State machines, PVS higher order logic and PVS theorem proving (only required for full formal verification).
5. IDE principles	Netbeans Visual API	Web
6. IDE availability	Available upon request for collaborations only.	Open source, downloadable at http://www.pvsioweb.org
7. Notation names	ICO, HAMSTERS.	Emucharts, PVS.
8. Notation instance	Petri Net, task models.	Statecharts, higher-order logic.
9. Notation paradigm	Event-based, state-based, procedural.	Event-based, state-based, functional.
10. Structuring models	Object-oriented, component-based.	Module-based.
11. Model editing features	Graphical editing of task models, ICO models and their correspondences, auto-completion features of models, visual representation of properties on models, simulation of models at editing time.	Graphical and textual editing of models, automatic generation of PVS models.
12. Suggestions for model improvement	Suggestions for model correction by real time analysis of models and continuous visualization of analysis results.	Strengthening of pre- and post- conditions of transition functions (based on proof obligations generated by PVS).
<i>continues on next page...</i>		

Formal IDE	CIRCUS	PVSio-web
13. Prototype building	Use of graphical user interface editor of NetBeans for standard interactions (e.g. WIMP), possible to create interactive components and assemble them for non standard interactions (e.g. multitouch).	Visual editing, based on a picture of the real system.
14. Prototype execution	Java Virtual Machine.	Javascript execution environment, Lisp.
15. User interface testing	Automatic execution of test sequences based on a task model	Automated execution of input test sequences recorded during interactions with the prototype.
16. Human-machine interaction techniques	Pre-WIMP, WIMP, post-WIMP, multimodal, multi-touch. Run-time re-configuration of interaction techniques.	Pre-WIMP, WIMP, post-WIMP, multimodal.
17. Code generation	Run-time execution of ICO models (to support prototyping and co-execution of task and system models).	Run-time execution of PVS executable models through the PVS ground evaluator (to support rapid prototyping), and automatic generation of production code compliant to MISRA-C (only for formal models developed using Emucharts diagrams).
18. Verification types	Well-formedness of the model: absence of deadlocks, token conservation. Functional analysis: reinitiability; availability of widgets; compliance with task models. Quantitative analysis: statistics about the total number of user tasks; estimation of the cognitive workload of the user based on the types of human-machine interactions necessary to operate the system. Simulation-based analysis through model animation.	Functional analysis, including: coverage of conditions, disjointness of conditions, correct use of data types, compliance with design requirements. Simulation-based analysis through model animation.
19. Technology	Static analysis of Petri Nets; interactive simulation of task and system models. Proofs and properties verification left to the analyst.	Theorem proving; interactive simulations.
20. Scalability	Applied to very large scale (industrial) applications (more than 200 models).	User interface prototype of stand-alone devices.
21. Analysis of the wider socio-technical system	Modelling of integrated views of the three elements of socio-technical systems (organization, human and interactive systems); however, FRAM-based description of organization and variability of performance has only be addressed at model level and not a tool level.	Modelling patterns based on distributed cognition theory have been explored in PVS but are not currently integrated in the IDE.
22. Related development process	User centered design (task-based design), iterative development, model-based engineering.	User centered design, agile development, model-based engineering.