This is the authors' version of a paper accepted for publication at the ACM Transactions on Computer-Human Interaction.
The published version can be found in ACM Transactions on Computer-Human Interaction, 27(5), August. DOI:10.1145/3404199

Supporting the analysis of safety critical user interfaces: an exploration of three formal tools

JOSÉ CREISSAC CAMPOS, HASLab / INESC TEC and Universidade do Minho, Portugal, Portugal
CAMILLE FAYOLLAS, University of Toulouse, France, France
MICHAEL D. HARRISON, School of Computing, Newcastle University, UK, England
CÉLIA MARTINIE, ICS-IRIT, University of Toulouse, France, France
PAOLO MASCI, National Institute of Aerospace, USA, United States of America
PHILIPPE PALANQUE, ICS-IRIT, University of Toulouse, France, France

Use error due to user interface design defects is a major concern in many safety critical domains, for example avionics and healthcare. Early detection of latent user interface problems can be facilitated by user centered design methods that integrate formal verification technologies. This paper considers the role that formal verification technologies can play in the context of user centered design by considering three existing tools: CIRCUS, PVSio-web, and IVY. These tools have been developed to support the model based analysis of critical user interfaces. They have their foundations in existing formal verification technologies, but each of them is focused towards particular issues relating to user interface design. The paper explores the different phases of the user centered design process and the extent to which each of these tools supports these phases. Criteria are developed for assessing their role at each stage of the design process. The results of the evaluation provide guidance to developers to help choose the most appropriate tool based on their analysis needs while at the same time setting challenges for future developments.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**.

Additional Key Words and Phrases: Formal modeling of interactive systems, user centered design, safety critical systems

ACM Reference Format:
José Creissac Campos, Camille Fayollas, Michael D. Harrison, Célia Martinie, Paolo Masci, and Philippe Palanque. 2020. Supporting the analysis of safety critical user interfaces: an exploration of three formal tools. 1, 1 (May 2020), 48 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

User interface software issues are an important source of system and device failure in application domains such as healthcare, avionics, and traffic control. It is therefore surprising that relatively little work has been done

This work is part supported by the European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia (project POCI-01-0145-FEDER-016826).
Authors' addresses: M.D. Harrison, School of Computing, Newcastle University, Urban Sciences Building, Newcastle upon Tyne, UK; P. Masci, National Institute of Aerospace, Hampton, VA, USA; J.C. Campos, HASLab/INESC TEC and Universidade do Minho, Braga, Portugal; C. Fayollas, C. Martinie and P. Palanque, ICS-IRIT, University of Toulouse, France.
Authors' addresses: José Creissac Campos, HASLab / INESC TEC and Universidade do Minho, Portugal, Braga, Portugal; Camille Fayollas, University of Toulouse, France, ICS-IRIT, Toulouse, France; Michael D. Harrison, School of Computing, Newcastle University, UK, Newcastle upon Tyne, England; Célia Martinie, ICS-IRIT, University of Toulouse, France, Toulouse, France; Paolo Masci, National Institute of Aerospace, USA, Hampton, VA, United States of America; Philippe Palanque, ICS-IRIT, University of Toulouse, France, Toulouse, France.

XXXX-XXXX/2020/5-ART $15.00
https://doi.org/0000001.0000001

to provide tool support for the analysis of use related software requirements. Tools for modeling and analysis of critical user interfaces require additional characteristics to support a multidisciplinary team of designers from different engineering disciplines, including, for example, domain experts (to establish design requirements and interpret compliance), human factors engineers (to run user studies), formal methods analysts (to verify compliance of a system design with design requirements), and software engineers (to generate prototypes and develop software from verified user interface models).

Significant work has been done to create tools that combine user centered design and formal verification technologies. However, in practice these tools support different levels of description and different types of analysis, ranging from aspects of human-machine interaction at the micro-level, e.g., layout and behavior of user interface widgets, to the analysis of user tasks and the wider socio-technical system within which the interactive system is used. To address these different, use related, safety concerns correctly, it is important that developers choose the most appropriate tool, based both on the analysis needs and in a form that makes sense to the teams that typically perform these processes. It is important to consider how the process and results are presented to these teams. Clear guidelines and detailed benchmarks that are necessary to choose the right t ool are, as yet, unavailable. The aim of this paper is to make a first step to address this gap.

This paper explores the role of formal analysis tools in the model based design and implementation of interactive systems, and in particular, their role in supporting use related safety of interactive systems. It does this by comparing three model based analysis tools (CIRCUS [34], PVSio-web [61], and IVY [19]) and their ability to support the different stages. Each provides a different focus on the design process. For example, while CIRCUS supports task models explicitly, IVY and PVSio-web focus particularly on the device model, and PVSio-web further provides a prototyping environment which allows a richer exploration of the design decisions. These tools are distinctive because they have their foundations in formal verification technologies and are chosen because the developers of each tool claim safety analysis as an important justification for their development. Furthermore these three tools each cover some aspects of the modeling, prototyping, verification, and validation of critical user interfaces, focusing on user interface design issues. As will be discussed in Section 9 they are not unique in this respect but they are representative of the current state of similar formal analysis tools. The paper introduces criteria that aim to clarify understanding of the appropriateness of each tool while at the same time providing a foundation for assessing which tools are best for which design problem. The aim is that this evaluation should provide developers with illustrations of how expert users of the tools are able to analyze a range of usability and safety properties of the system. It is hoped that these illustrations may be used by developers as guidance, to enable choice of the most appropriate tool for their analysis needs. Each tool (with the exception of parts of the PVSio-web tool) has been used exclusively by the community who developed it. However the tools have been used to provide analyses of systems that have many users. The intention of the evaluation is to encourage a broader community to consider using them within their own organizations.

The paper builds on an earlier paper [35] which provides a less detailed comparison of CIRCUS and PVSio-web.

Contribution. The novel contributions of the paper are:

- a detailed exploration of the role of formal tools during the stages of model based analysis of critical user interfaces;
- an extended set of criteria that can be used for evaluating a model based analysis tool for critical user interfaces;
- an exploration and comparison of the three model based analysis tools through the medium of a case study.

Organization. The paper is organized as follows.

- Section 2 provides an overview of the three tools that form the basis for the comparative evaluation.
- Section 3 describes the evaluation method and introduces the groups of criteria that are used as the basis for the comparison.

- Section 4 introduces the design problem which is based on a subsystem of the Flight Control Unit of the Airbus 380.
- Sections 5, 6 and 7 contain the evaluations of the three tools applied to the design problem.
- Section 8 summarizes the results of the evaluation.
- Section 9 discusses other related tools that provide some of the elements of the tools evaluated in the paper.
- Section 10 concludes the paper.

2 THE SELECTED ANALYSIS TOOLS

The tools were selected according to their ability to support the engineering of safety critical user interface designs:

- **User centered design.** The tool should support the three phases of a user centered design process: requirements definition; detailed user interface design; design evaluation.
- **Formal verification.** The tool should support the use of formal (mathematical) verification technologies, as recommended by standards in critical application domains such as avionics and healthcare.
- **Maturity level.** The tool should have a stable release, and should have been applied successfully to realistic examples.
- **Tool updates.** The tools should be under active development, and new releases of the tool should continue to be available.
- **Tool availability.** The tool should be available free of charge for academic use.

2.1 CIRCUS

CIRCUS[1] (Computer-aided-design of Interactive, Resilient, Critical and Usable Systems) is an integrated development environment that embeds two types of models: system models (covering functional core, dialog and interaction techniques) and hierarchical task models (describing user actions, user knowledge, strategies, information and devices used to reach a goal). CIRCUS is available only through research collaboration. The tool (or some of its components) have been deployed in organizations (e.g., CNES) and companies (e.g., Airbus and Thales).

The CIRCUS environment aims to support multi-disciplinary teams of software engineers, system designers and human factors experts. It is embedded within the NetBeans[2] platform and its architecture is presented in Figure 1. CIRCUS consists of the following components:

- **HAMSTERS** enables the editing and simulation of 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 [6]. The notation used in the tool makes it possible to structure users' goals and sub-goals into hierarchical task trees. Mathematical operators such as >> (meaning sequence) are available for expressing temporal relationships among tasks as well as for the modeling of: specialized task types; explicit representations of data and knowledge; device descriptions; genotypes and phenotypes of errors and collaborative tasks.
- **PetShop** is a tool for creating, editing, simulating and analyzing system models using the ICO (Interactive Cooperative Objects) notation [56, 68]. The ICO notation allows developers to specify the behavior and the appearance of interactive systems. The notation uses Petri nets for describing dynamic behaviors, and uses object-oriented concepts (including dynamic instantiation, classification, encapsulation, inheritance and client/server relationships) to describe structural or static aspects of the system. In the ICO UIDL, an object is an entity featuring four components: a cooperative object which describes the behavior of the object, a

[1]Documentation can be found at https://www.irit.fr/recherches/ICS/documentation/ and contact address is palanque@irit.fr
[2]Information about NetBeans can be found at https://netbeans.org/

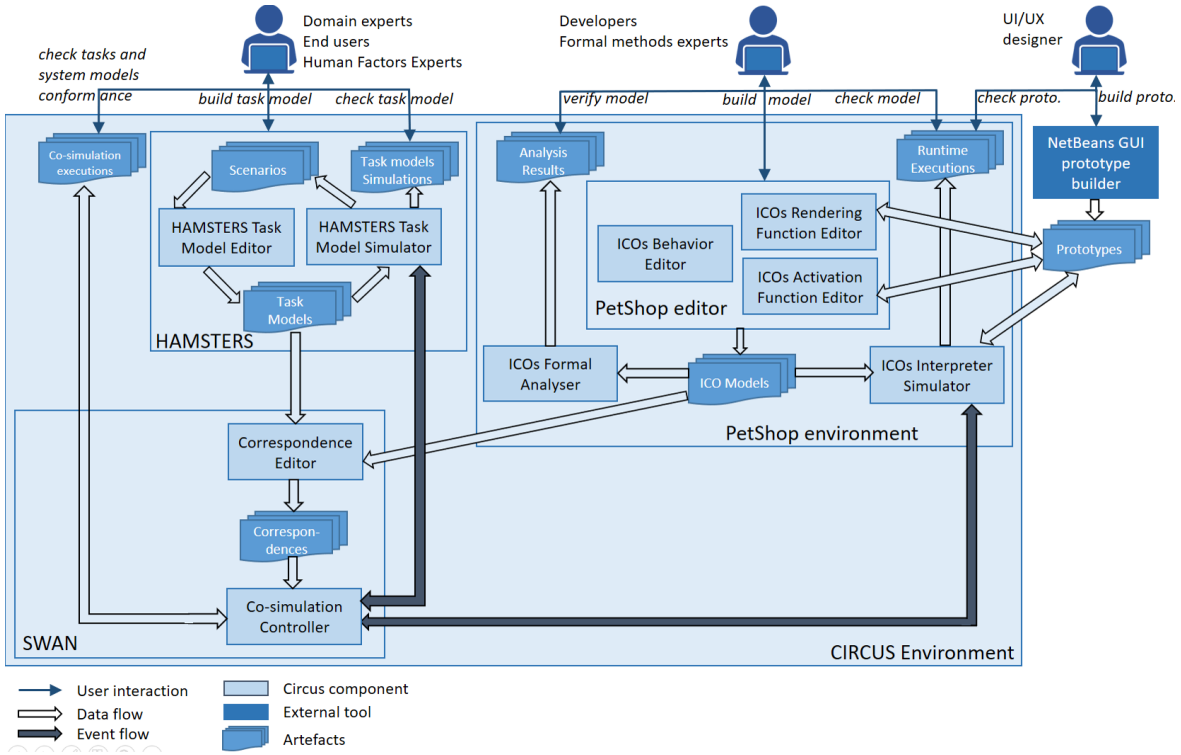

Fig. 1. The architecture of the CIRCUS environment

presentation part, and two functions (the activation function and the rendering function) which make the link between the cooperative object and the presentation part. Prototyping is therefore done by using Java widget libraries in the presentation part, or by invoking Java code dedicated to presentation [67].

- **SWAN** is a tool for the co-execution of PetShop models and HAMSTERS models [6]. The tool allows developers to establish correspondences between system behaviors and tasks, and perform automated system testing by means of co-execution [16].

2.2 PVSio-web

PVSio-web is an open source toolkit for model based development of user interfaces. The toolkit can be downloaded from github[3] and www.pvsioweb.org.

The toolkit aims to support a multi-disciplinary team of user interface engineers, domain experts, and software analysts. This support is realized by integrating specialized components designed for different target users. In its current version (2.2), the toolkit includes seven main components (see Figure 2):

- **Prototype Builder.** This component allows developers to create the visual aspect and the logic of operation of a device prototype. The visual aspect of the prototype is an interactive picture of the device realized using

[3]https://github.com/thehogfather/pvsio-web

Web technologies. The logic of operation, the model of interactive behavior of the system, is developed in the language of the Prototype Verification System (PVS) [70].

- **Simulator.** This component renders the visual appearance of a prototype within a Web browser. The logic of operation of the prototype is executed in PVSio [65], the native component of the PVS system for animating PVS models. User actions over input widgets (e.g., button presses) are translated by the Simulator into PVS expressions that can be evaluated in PVSio. The result of the PVSio evaluation is rendered on the Web browser using the output widgets of the prototype, so that the visual appearance of the prototype closely resembles that of the real system in the corresponding states.
- **Storyboard Editor.** This editor facilitates the development of preliminary mockup prototypes based on story-boards. Developers can load mockup pictures of different screens, define input widgets on the pictures, and link user actions attached to input widgets with the transitions between screens.
- **Emucharts Editor.** This component facilitates the creation of PVS models using visual diagrams known as Emucharts. These diagrams are based on the Statecharts notation [48].
- **Model Editor.** This component allows developers to edit PVS models. The editor provides syntax highlighting, auto-completion, search, and compile. A file browser incorporated in the model editor allows developers to select, rename, delete and create files and directories.
- **Co-simulation Engine.** This component enables integrated simulation of PVS models and other models developed with different simulation frameworks, e.g., Simulink. Two co-simulation engines are currently implemented: an internal engine based on a WebSocket protocol, and an external engine [60] based on a communication and co-ordination middleware [89]. A preliminary version of a third co-simulation engine based on the FMI standard[4] is described in [75].
- **Property proving assistant.** This component includes the PVSio [65] environment and the PVS [70] theorem proving assistant, which are developed and maintained by SRI International and NASA Langley. PVSio is used during simulations to evaluate PVS expressions generated by the PVSio-web Simulator. The theorem prover is used for formal analysis of use related safety properties of the prototypes. Initial support for automatic instantiation and verification of use related property templates [46] is available in the latest release of the toolkit.

2.3 IVY Workbench

IVY is a tool for model based analysis of interactive systems designs. It is free for academic use, and can be downloaded[5].

The tool consists of a set of plug-ins that provide a front end to the NuSMV model checker [23]. The toolkit supports a notation, Modal Action Logic (MAL), that enables the specification of interactive systems and provides a set of property templates designed to aid the development of appropriate properties for the analysis of the model. The results, which include traces provided by the model checking analysis when a property fails to be true, are visualized. The tool offers a selection of visualization formats. The aim in developing IVY was to provide representations and analysis tools that were more easily usable by user interface developers, and in which the results could be communicated effectively within an interdisciplinary team of software engineers and formal methods experts. The IVY toolkit architecture is organized into an extensible set of interoperable components. In its current version, the toolkit includes four components (see Figure 3):

- **MAL editor.** This component provides a standard text editor with some support for visualizing the structure of MAL models. A simple structural pattern is used for aiding the development of the MAL specification. The modeling language is based on Structured MAL [82]. MAL (Modal Action Logic) is a (deontic) modal

[4]http://fmi-standard.org
[5]http://ivy.di.uminho.pt/

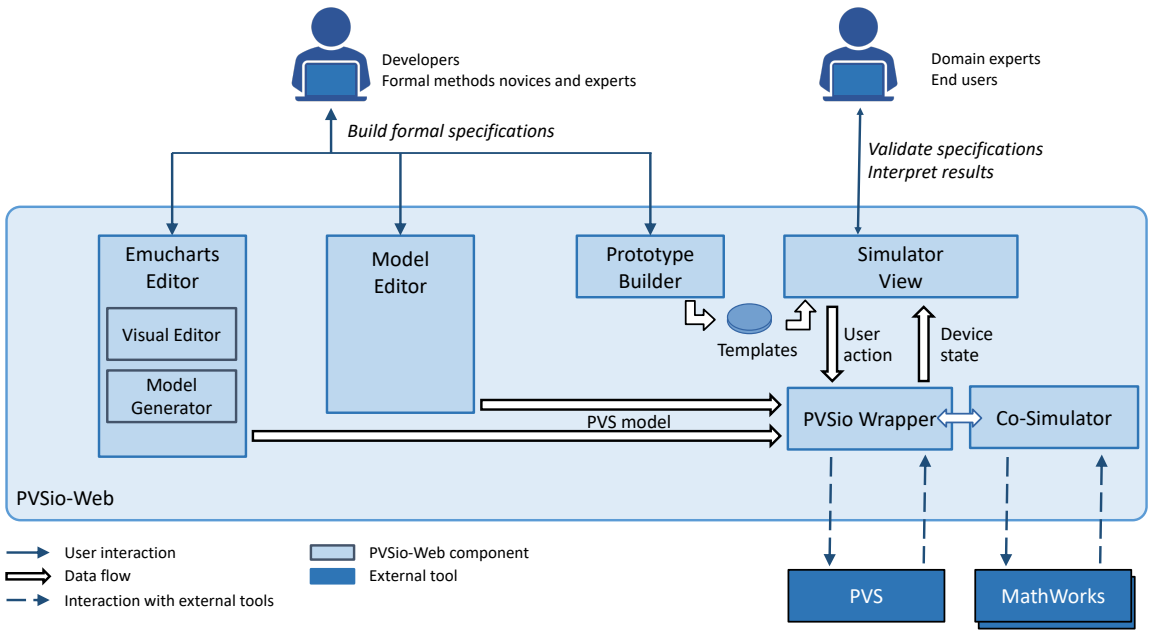

Fig. 2. The structure of PVSioweb

logic that incorporates a notion of *action*. Structured MAL adds mechanisms for structuring the specification which are used to express the notion of interactor [29, 76]. Interactors are modules that have a state (defined by attributes) which is (partially) made available to the user through some presentation medium, and a set of actions (some available to users, some internal) that act on that state. MAL describes a logic of actions and is used to write production rules that describe the effect of actions on the state of the device. This style of specification was used because there is some evidence that it is found easy to use by software engineers [63] and preferred to the notation used by the NuSMV model checker. The language also enables the expression of when an action is allowed, using permissions. Non-determinism is possible when more than one action is allowed in the same state of the described model and/or when an action does not fully determine the next state of the system

- **Property editor.** This component supports the formulation of properties of the model. Properties are expressed in the CTL notation (as used in NuSMV). Assistance is provided in formulating the properties using templates or patterns. The analyst is able to instantiate a property template with the attribute or action names defined in the MAL model.
- **Trace visualizer.** If a property fails to be true of the model a counter-example is produced. This provides a witness indicating one case where the property has failed. The IVY tool enables a variety of formats for describing counter-examples including a matrix notation showing the values of all the attributes of the state at each step of the execution of actions, and a variant of a UML activity diagram.

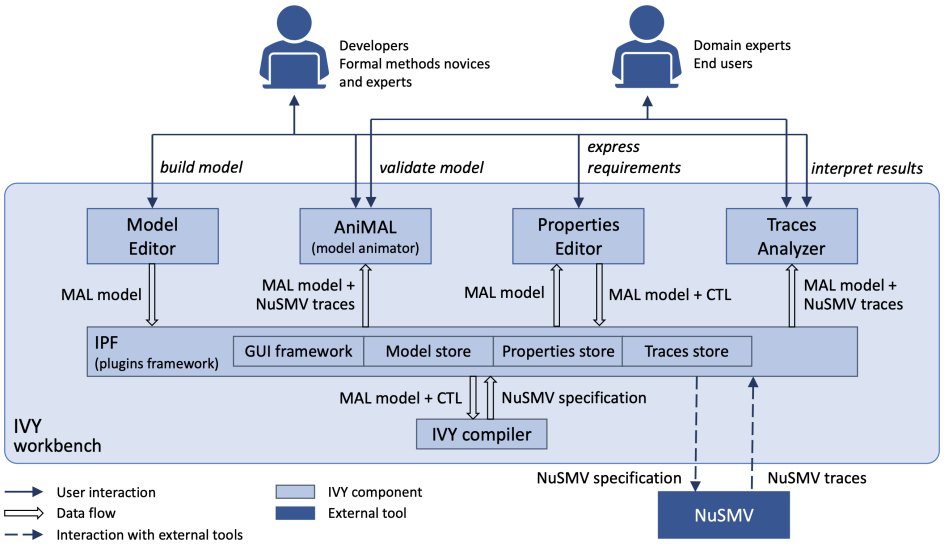

Fig. 3. The structure of IVY

- **Trace simulator.** The simulator enables the production of traces dynamically. It is possible to explore alternative paths, and therefore an analysis of "what if" scenarios.

3 EVALUATION METHOD

The method adopted for the evaluation of the tools is based on a generic development process described by Billman et at [10]. They discuss two difficulties: (1) when change is easy then information to guide change is scarce; (2) when information is available then change is difficult. Billman et al. propose a generic development process that extends and develops Boehm's spiral model [12]. This generic process consists of three phases:

(1) a work and task analysis loop designed to establish criteria for assessing the adequacy or efficiency of the technology to support the work for which it is designed;
(2) an early prototyping loop which provides rapid feedback about the effectiveness of a solution and how much the technology is necessary for the work;
(3) a full scale prototyping loop which takes fuller functionality to assess the proposed solution in a richer information environment.

The criteria that have been developed here have a grouping that is orthogonal to these phases. Criteria first concern the general characteristics of the tool in terms of how much of the development process it assesses, who can use the tool or the information it provides and whether the tool can be extended. Criteria then focus on three concerns that span the phases. These are the means of producing models, the mechanisms for prototyping and the means of analysis. Evaluating the tools therefore involves the following steps.

- Criteria are identified that are important for modeling and analysis of critical user interfaces.
- A design problem is introduced: a fictional enhancement to an existing safety-critical user interface.
- The means by which each tool can be applied to the example is discussed assessing the performance of the tool against each group of the criteria.

To ensure best and most effective use of the tools, the application of each tool to the example was carried out by a member of the team that had developed the tool. The criteria and the design problem are introduced in the following sub-sections. The application of the tools to the example is described in Sections 5, 6, and 7. The criteria will be applied using the design problem described in Section 4 to illustrate their application.

3.1 The criteria

Based on experience using and developing tools for model based analysis of critical user interfaces, 22 criteria were identified to highlight the characteristics of the tools. The criteria are summarized in four groups designed to represent how well the candidate tool performs against the various aspects of the development and analysis of safety critical interactive systems. The criteria are qualitative and are designed to be of general application.

Group 1: General aspects of the tool. The first group are summary criteria. General features of interactive systems design and analysis are a focus. Criteria assess how well the tool supports the stages of a typical development process, whether it provides support for multi-disciplinary use and whether the tool itself supports an explicit development process. Criteria also relate to whether the tools can be extended or tailored to particular requirements and what technology is used that aids these adaptations.

(1) **Scope/purpose of the tool within the development process.** At what stage in a user centered design process can the tool be used, e.g., when exploring conceptual user interface design, when exploring design requirements, when analyzing a system design against its design requirements.
(2) **Support for multidisciplinary teams.** What are the target users of the tool (e.g., domain experts, software engineers, human factors specialists) and which functionalities of the tools can support multi-disciplinary work among target users.
(3) **Related development process.** Which development process could be supported by the tool, e.g., user centered design, waterfall development process, agile development.
(4) **Tool features**. Which aspects of the system can be modeled and analyzed, e.g., modeling of user tasks and goals, analysis of usability properties, simulation of user tasks.
(5) **Tool extensibility.** Which aspects of the tool can be customized for specific needs that are not provided by the tool, e.g., to model systems from different application domains, or to perform a different type of analysis.
(6) **Prerequisites.** What background knowledge is required to use the tool, e.g., distributed systems, object oriented languages, Petri Nets, task modeling, model checking, theorem proving.
(7) **IDE instance and principle.** What technology is used for the development of the tool, e.g., Eclipse API, Netbeans API, Web technologies.
(8) **IDE availability.** Which type of distribution is available, and under which conditions of use, e.g., snapshot, demo, downloadable, open source.

Group 2: Modeling criteria. The second group of criteria focus on the type of modeling supported by the tool. Criteria focus on the modeling notation as well as the theory paradigm supported by the notation. An important criterion relates to mechanisms for structuring specifications. Criteria also consider how editing of a specification is supported as well as any checking of the specification under development.

(9) **Modeling paradigm.** The modeling approach supported by the tool, e.g., event-based, state-based, data-flow based, declarative.
(10) **Modeling language.** The notation supported by the tool to represent relevant aspects of the system under analysis, e.g., Petri Net, state machines, higher-order logic.
(11) **Structuring models.** Mechanism for model organization, e.g., object-oriented, functional, component-based.

(12) **Model editing features.** Support provided by the tool for developing models, e.g., textual, visual, auto-completion support.
(13) **Suggestions for model improvements.** Support for checking compliance with best modeling practice and design patterns, e.g., strengthening of pre-conditions.

Group 3: Prototyping criteria. The third group of criteria focus on the means of developing prototypes and styles of interaction supported by prototypes. Criteria focus on support for prototype building as well as the execution environment for a developed prototype. Criteria also focus on the interaction techniques that can be supported and whether code can be generated from the prototype.

(14) **Support for prototype building.** Environment provided for defining the visual appearance of the user interface, and to link user interface widgets to functionalities defined in a model, e.g., visual editor, library of widgets.
(15) **Execution environment of the prototype.** Technology used to deploy the prototype, e.g., Java virtual machine, Javascript execution environment.
(16) **Human-machine interaction techniques.** What user interactions can be captured/defined by the tool, e.g., Pre-WIMP (input dissociated from output), WIMP, post-WIMP, tangible, multitouch, multimodal.
(17) **Code generation.** Capability provided by the tool for translating the prototype into code for a physical prototype or the final product, e.g., C, C++, Java.

Group 4: Analysis and verification criteria. The fourth and final group is concerned with analysis of a model or corresponding prototype. First criteria are concerned with the type of verification supported and tools provided to support it. Also of concern are the support for usability testing and whether there is support for analysis of the broader sociotechnical system. Finally a concern is with issues of scalability.

(18) **Verification type.** Types of model based analysis supported by the tool, e.g., functional verification, performance analysis, hierarchical task analysis.
(19) **Verification technology.** Approach used in the tool to perform the analysis, e.g., theorem proving, static analysis, model checking. This can give developers insights about the level of completeness of the analysis and the level of automation of the verification tasks.
(20) **Scalability of the analysis.** What complexity and size of models can be analyzed with the tool, e.g., illustrative examples, industrial scale.
(21) **User interface testing.** Whether test suites for the final product can be generated from the model, e.g., automatic generation of input test cases.
(22) **Support for the analysis of the wider socio-technical system.** Whether the tool supports extending the analysis of a single device to include aspects of the context within which the device is used, e.g., teamwork, human-human collaboration, organization.

4 A SAFETY-CRITICAL CASE STUDY

The example adopted for the evaluation of the tools is based on a subsystem of the Flight Control Unit (FCU) of the Airbus A380. The FCU is an interactive hardware panel with several different buttons, knobs, and displays (see Figure 4) which allows flight crew to manipulate auto-pilot parameters (e.g. altitude, speed and heading) as well as setting information to be displayed using the Navigation Displays (ND). One ND is available for each pilot as presented at the bottom of Figure 4. Further details of the example are available[6].

A future cockpit design is considered. The interactive hardware elements of the FCU panel are to be replaced by an interactive graphical application rendered on one of the cockpit displays (see Figure 5). This graphical software (hereafter, referred to as FCU Software) would provide the same functionalities as the corresponding

[6]https://sites.google.com/view/fcusoftware

Fig. 4. The cockpit layout relevant to the proposed design

hardware elements. An envisaged design of this kind provides a comprehensive set of modeling and analysis challenges. It is proposed that the FCU Software would offer two components relevant to the present exploration: an Electronic Flight Information System Control Panel (EFIS CP), for configuring the piloting and navigation displays (ND); and an Auto Flight System Control Panel (AFS CP), for setting the autopilot state and parameters.

To keep the design problem simple the EFIS Control Panel only is considered. The right side of the EFIS Control Panel (see right-hand side of Figure 5) is dedicated to the configuration of the Navigation Display (ND) and the top part provides buttons for adding or removing information on the ND. For instance, WPT button adds the way-points information on the ND. The DropDownComboBoxes at the bottom allow the pilot to choose display modes on the ND (e.g. LS for Landing System or VV for Velocity Vector) and range of the weather radar (e.g. 160).

The left side of the EFIS CP is dedicated to the configuration of the barometer settings. Pilots interact with the FCU Software via the Keyboard and Cursor Control Unit, that integrates a keyboard and track-ball (see the leftmost picture in Figure 5). 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 aircraft 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. The EFIS CP panel is composed of several widgets: two CheckButtons enable pilots to select the pressure mode which value is either Standard (STD) or Regional Pressure Settings (QNH). When in QNH mode, a number entry widget (EditBoxNumeric following ARINC 661 terminology [5]) enables pilots to set the barometric reference value. When in STD mode, it is not possible to enter a value in that EditBoxNumeric. Finally, a button (PushButton labelled INHG→HPA on the Figure) 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

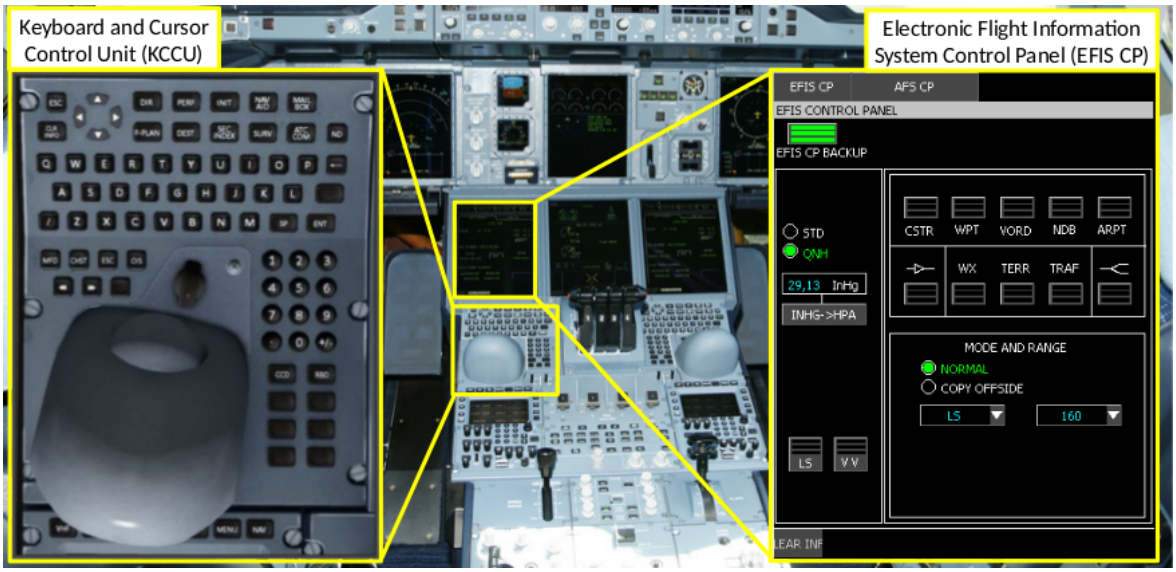

Fig. 5. Keyboard and Cursor Control Unit and Flight Control Unit Software.

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).

Some requirements of the proposed design

The following use related requirements will provide an introduction to the analysis capabilities of the tools. They are intended to show simple examples of the requirements that might be relevant for this particular design. Other examples, relating to FDA use centered requirements for medical devices can be found, for example, in [46, 59]. The following four requirements relate to the ease of access to features of the interface.

R1: reinitialisability It should always be possible to return to the initial state of the FCU through a single action. This requirement addresses the problem that the user might become disorientated and locked into some feature of the interface. It also guarantees that the use of the system does not alter its functioning.

R2: availability of buttons The STD and QNH modes should always be accessible through a single action. This requirement ensures accessibility of the two main modes of the design and ensures that the mode can be changed through a simple action.

R3: mutual exclusion The STD and QNH modes should be mutually exclusive. The only means of change from one mode to another should be the designated actions described in R2. This ensures that the design does not allow the user to stray into another mode without using the STD and QNH buttons.

R4: reversibility Actions should be available that are reversible in the sense that an action's effect on relevant state variables may be undone by a following action. This ensures that an action can be recovered from by performing a simple procedure.

5 EVALUATION OF CIRCUS

5.1 General aspects of the tool

CIRCUS allows human factor experts to gather information during the task analysis phase as well as to check the behavior through the simulation of task models using the HAMSTERS tool. The tool also enables, using the ICO notation, modeling of the entire interactive system from drivers of the input devices [1], interaction techniques [39] to interactive applications [69]. User interface prototyping capabilities are provided by the NetBeans IDE and therefore limited to high-fidelity prototyping of WIMP interactions. More sophisticated interfaces can be programmed in JavaFX, for example, interactive cockpit multi-touch interfaces described in [38]. Early prototyping is outside the scope of CIRCUS. Interactive system models and task models can be inter-connected to check their compatibility and to simulate them [72]. A Java-based API allows developers to extend CIRCUS with additional features, e.g., to connect the user interface simulator to cockpit software simulators [4], or to extend the analysis modules with external Petri net analysis tools [84]. The workflow supported by the tool includes six steps:

Step 1: Task analysis and modeling. This step identifies goals, tasks, and activities that are intended to be performed by the operator, modeled using the HAMSTERS tool. This also enables description of use error related scenarios, providing a basis for assessing the costs of recovering from errors [30].

Step 2: Workload and performance analysis. HAMSTERS enables differentiation between cognitive, motor, and perception tasks and represents the knowledge and information needed by the user to perform a task. This enables qualitative analysis to assess 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 [34].

Step 3: User interface look and feel prototyping. The CIRCUS environment is built on top of the NetBeans platform and therefore it is possible to produce rapid, high-fidelity prototypes of the look and feel of a design using Rapid Application Development (RAD) tools.

Step 4: User interface formal modeling. Formal modeling is achieved using the ICO notation. The formal models can embed qualitative and quantitative time and, use the underlying Petri nets to describe concurrent behavior and dynamic instantiation [39].

Step 5: Formal analysis. ICO models can be analyzed using techniques for high-level Petri nets (e.g. symbolic graph calculation [49]) or, alternatively, techniques related to the underlying Petri net model. These analysis can be presented to the analyst using CIRCUS, interleaved with the editing and simulation of the model to support the correction of detected modeling faults [33]).

Step 6: Compatibility assessment between task models and user interface models. The task model and the formal model of the user interface behavior can be analyzed in terms of their mutual completeness and consistency.

Summary criteria

scope/purpose: interactive system prototyping, development, and analysis;
support for multi-disciplinarity: human factor expert performing task analysis and building task models in HAMSTERS, software engineers;
related development process: user centered design (task-based design only), iterative development, model based engineering;
tool features: user task and goals description, interaction logic (dialog) and interaction techniques modeling, interactive system prototyping, support for verification of generic properties, assessment of compatibility between user tasks and interactive system prototype;
tool extensibility: user task and goals description (with extensibility features of the HAMSTERS notation itself (see [57]), interaction logic (dialog) and interaction techniques modeling, interactive system prototyping,

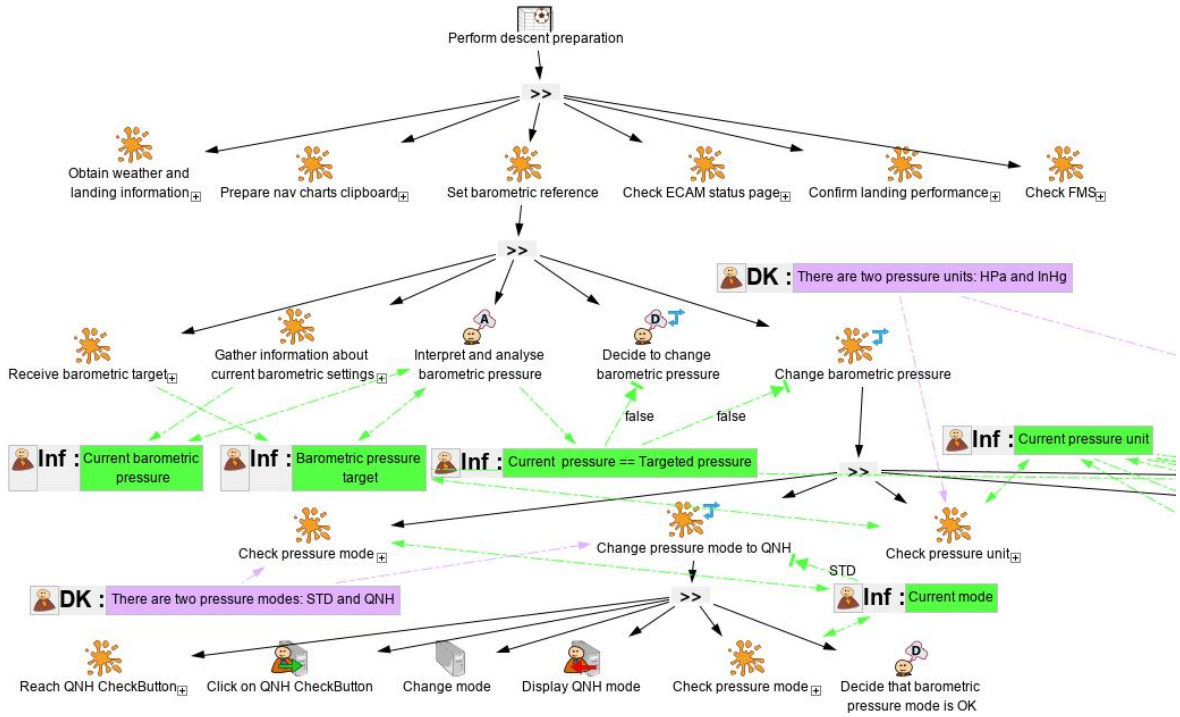

Fig. 6. Extract of the task model for the user goal "Perform descent preparation".

support for verification of properties, assessment of compatibility between user tasks and interactive system prototype;

Prerequisites: object-oriented analysis and programming; high-level Petri nets (for PetShop), Java programming, distributed systems, hierarchical task modeling;

IDE instance and principle: Netbeans Visual API;

IDE availability: documentation can be found at https://www.irit.fr/recherches/ICS/documentation/ and contact address is palanque@irit.fr.

5.2 Modeling features of CIRCUS

The CIRCUS environment provides two main modeling features: the HAMSTERS tool enables the modeling of user tasks; the PetShop tool enables the modeling of the interactive system behavior. These two features are detailed below using the example.

5.2.1 Task modeling. HAMSTERS has many of the characteristics of other task analysis notations (see, for example, [52, 64]). It is distinctive in that it includes informative annotations to goals and subgoals. Hence in Figure 6 (this is an extract of the full model[7]) the root node of the tree "Perform descent preparation" is annotated as a goal. Sub-goals are similarly annotated as such. Subgoals that involve processing by the operator are also

[7]https://sites.google.com/view/fcusoftware

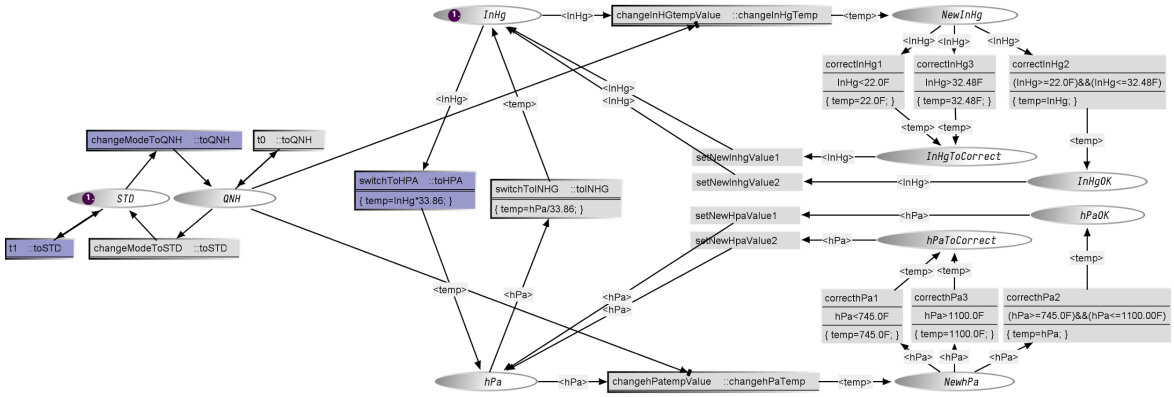

Fig. 7. ICO model of the barometer settings behavior.

annotated distinctively (see, for example, “Interpret and analyse barometric pressure”). Information used in completing a sub-goal is provided in a box marked “Inf”. Iterative sub-goals within a sequence of subgoals (marked by “>>”) are identified (see “change barometric pressure” for example). Decision subgoals are marked with a “D” and action subgoals with an “A”. Further boxes marked with “DK” identify a design issue. In both the cases that can be seen in Figure 6 these identify issues associated with the modes that may be required in any design to achieve the task. The task representation, as displayed in this fragment, captures part of the procedure described in the Flight Crew Operating Manual (FCOM) [2].

This task modeling activity is typically performed by Human Factors experts. In the context of aviation, certification specifications require that such task descriptions are performed and that the aircraft manufacturer demonstrates that the flight desk supports the performance of these tasks (see Certification Specification 25 - section 13.02[8]).

5.2.2 System modeling. The behavior of the FCU’s user interface is described in an ICO model. The barometer settings feature of the EFIS user interface is presented in Figure 7. The left part of this model (enlarged in Figure 8) is dedicated to the pressure mode. This mode can be in two mutually exclusive states: STD and QNH. The user can switch from one mode to the other by clicking either the STD or QNH CheckButton (clicking on a CheckButton while already in the corresponding mode is also possible but does not change the pressure mode). The state of the pressure mode is represented by the presence of a token within “QNH” or “STD” places (in Figure 7, place “STD” holds a token meaning that the current pressure mode is STD). Transitions “changePressureMode_1” and “changePressureMode_2” correspond to the availability of event “qnhClick”. When one of these two transitions is enabled (highlighted in blue in this example), 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 the QNH CheckButton. Transition “changePressureMode_2” allows the user to click the QNH CheckButton when in QNH pressure mode without any impact on the pressure mode. Similarly, transitions “changePressureMode_3” and “changePressureMode_4” correspond to the availability of event “stdClick”.

Summary modeling criteria

Modeling language: ICO, HAMSTERS;

[8]CS-25-Amendment 17 - Certification Specifications and Acceptable Means of Compliance for Large Aeroplanes. EASA, 2015

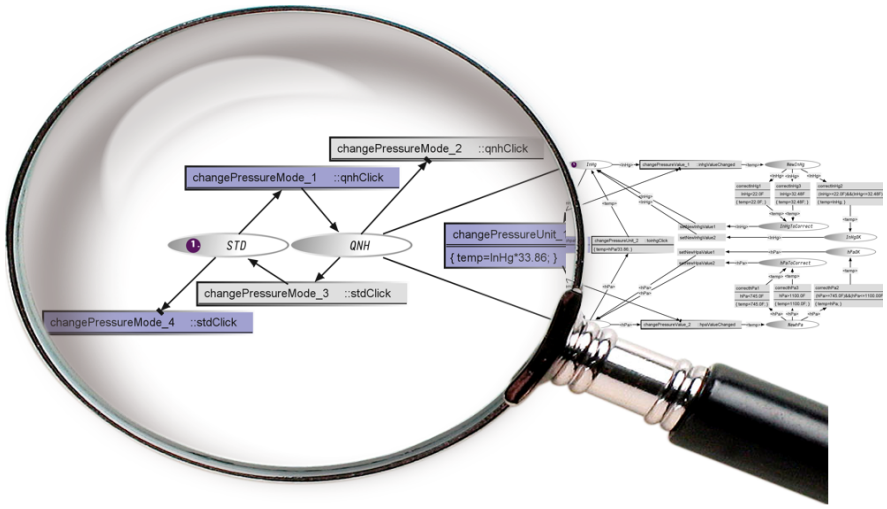

Fig. 8. ICO model of the barometer settings behavior. Right-hand side is a magnification of the left part of the model presented in Figure 7.

Modeling paradigm: event-based, state-based, procedural;
Structuring models: object-oriented, component-based;
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;
Suggestions for model improvements: suggestions for model correction by real time analysis of models and continuous visualization of analysis results);

5.3 Prototyping features of CIRCUS

The CIRCUS environment embeds the NetBeans Swing GUI Builder. This enables the prototyping of Java Swing interfaces. Swing components must be connected explicitly to ICO models. Beyond Java Swing, the CIRCUS environment also supports the use of the JavaFX libraries. The use of this feature has already been proposed as a means of prototyping applications featuring multi-touch interactions [37]. In the case of the EFIS example user interface a library of JavaBeans components has been created to provide assurance that widgets are compatible with the ARINC 661 standard [3]. The resulting interface is displayed in the screen-shot of the EFIS presented on the right-hand side of Figure 5.

When the interface prototype has been connected to the ICO models, the CIRCUS environment enables interaction with the prototype using interactive simulation. When the user interacts with the prototype, user actions on the input devices trigger events on the user interface elements that fire, in turn, transitions in the ICO model as presented in [66].

The CIRCUS environment offers a logging feature so that so that all the evolutions of the ICO model (e. g. token added to a place, firing a transition ...) following a user action on the user interface [71] are recorded. This feature makes user testing possible by providing data about user performance which is very useful when tuning multi-modal interaction techniques [71]. The CIRCUS environment does not provide a code generation feature. However an ICO model (developed to describe application behavior) can be directly executed and therefore used

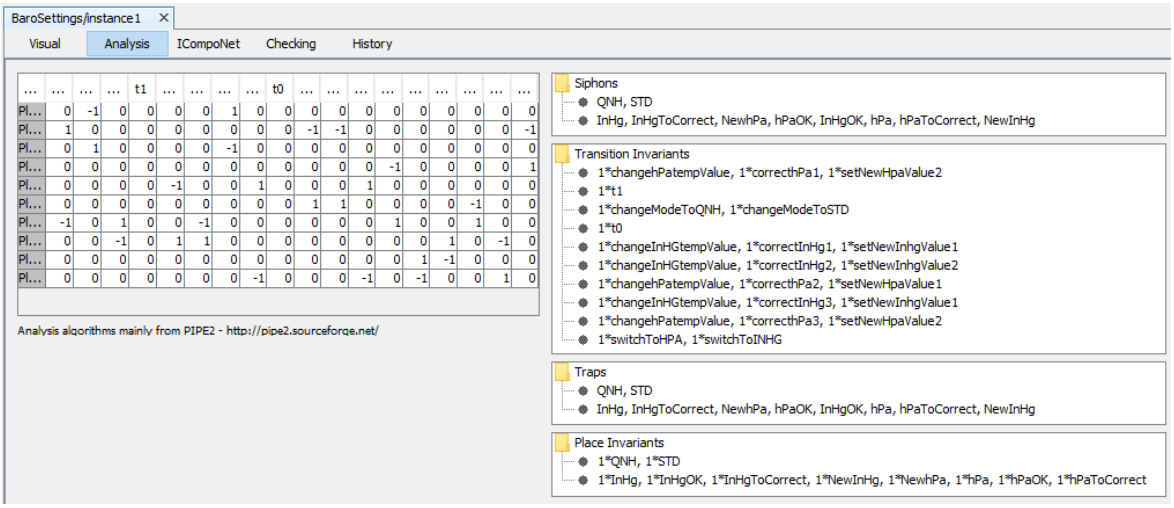

Fig. 9. Incidence matrix and invariants automatically calculated from the underlying Petri net model from figure 7

"as is" as ICO transitions can call methods written in Java. This technique has been used in aviation applications designed with the CIRCUS environment [8].

Summary prototyping criteria

support for 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); no low fidelity prototyping;

execution environment of the prototype: Java Virtual Machine;

human-machine interaction techniques: pre-WIMP, WIMP, post-WIMP, multimodal, multi-touch. run-time re-configuration of interaction techniques;

code generation: run-time execution of ICO models (to support prototyping and co-execution of task and system models).

5.4 Analysis and verification features

The CIRCUS environment provides two main analysis features: the PetShop tool supports automatic analysis of the structure of the underlying Petri net; the SWAN tool provides the means to analyze whether the task model is compatible with the system model. These two features are detailed below using the example.

5.4.1 Formal analysis and verification features of CIRCUS. CIRCUS enables the analysis of an ICO model to provide data that can be used to demonstrate that well-formedness conditions are true of the model. The raw analysis produces an "incidence matrix" (see Figure 9) that enables the calculation of:

siphon: a set of (one or more) places that cannot gain tokens from the initial marking (whatever transition is fired);

trap: a set of (one or more) places that cannot lose tokens from the initial marking (whatever transition is fired);

liveness: siphons and traps can provide information about the Petri net liveness;

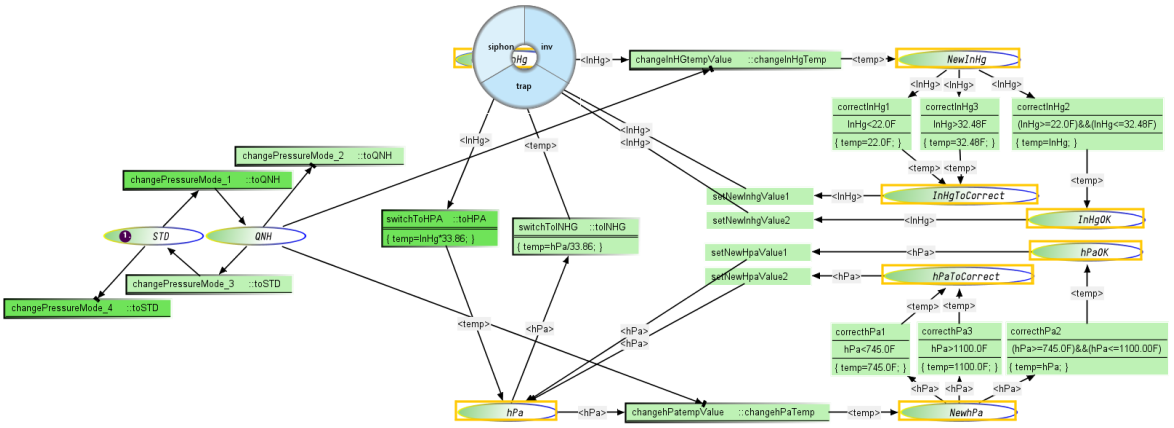

Fig. 10. Visual presentation of formal analysis results inside the PetShop tool. This model is the same as Figure 7 but presents visually the analysis results. Transitions are green, indicating that there is no evolution in the model that removes the availability of a transition. In terms of user interfaces, it means that whatever state the system is in, there is an interaction sequence that will enable the widget associated with a transition. The pop-up pie menu on top of the image has been used to display all the siphons including place InHg (under the pop-up menu).

place invariant: a set of (one or more) places that will always contain the same number of tokens (it cannot gain or lose tokens from initial marking);

transition invariant: a set of (one or more) transitions that indicate a possible loop in the net i.e. firing all of them brings the net back to the state it started from.

The information about analysis and verification results is presented in PetShop through two different types of visualizations:

- A dedicated panel (Figure 9) that presents the incidence matrix, the trap and siphons and the place and transition invariants. Figure 7 shows the Petri net model generated for the case study.
- A dedicated look and feel within the main editing view as shown in Figure 10. The green overlay on the places and transitions identifies the node part of the model's invariants otherwise the red overlay is used. Places with yellow borders are siphons whereas traps use a blue stroke. To determine which nodes belong to the same invariant, a modal pop-up menu is provided. The nodes belonging to the invariant are then framed with a yellow border. The siphon shown in Figure 10, for example, contains "InHg", "NewInHg", "InHgToCorrectOK", "hPa", "NewhPa", "hPaToCorrect", "hPaOK". The analysis mode can be activated without stopping either the editing or the simulation.

This raw data allows the verification of the use related properties of the FCU. The ICO model, unlike the models developed in PVS and IVY, involves the concurrent execution of processes. An initial step therefore involves establishing that the behavior of the barometer settings application is free of deadlock: the underlying Petri net is live as it contains two siphons that are marked with one token each and each of them contains a trap. The requirements established in Section 4 can be addressed. The style of proof using the ICO model differs from that described in the cases of PVS and IVY. While it is possible to demonstrate that requirements **R1-R3** hold true, as is indicated below, proof of **R4** requires more detail than the Petri net, as illustrated, provides. The specifications produced in PVS and IVY add additional detail to the design that leads to initial failure of some of the requirements as will be seen in the relevant sections.

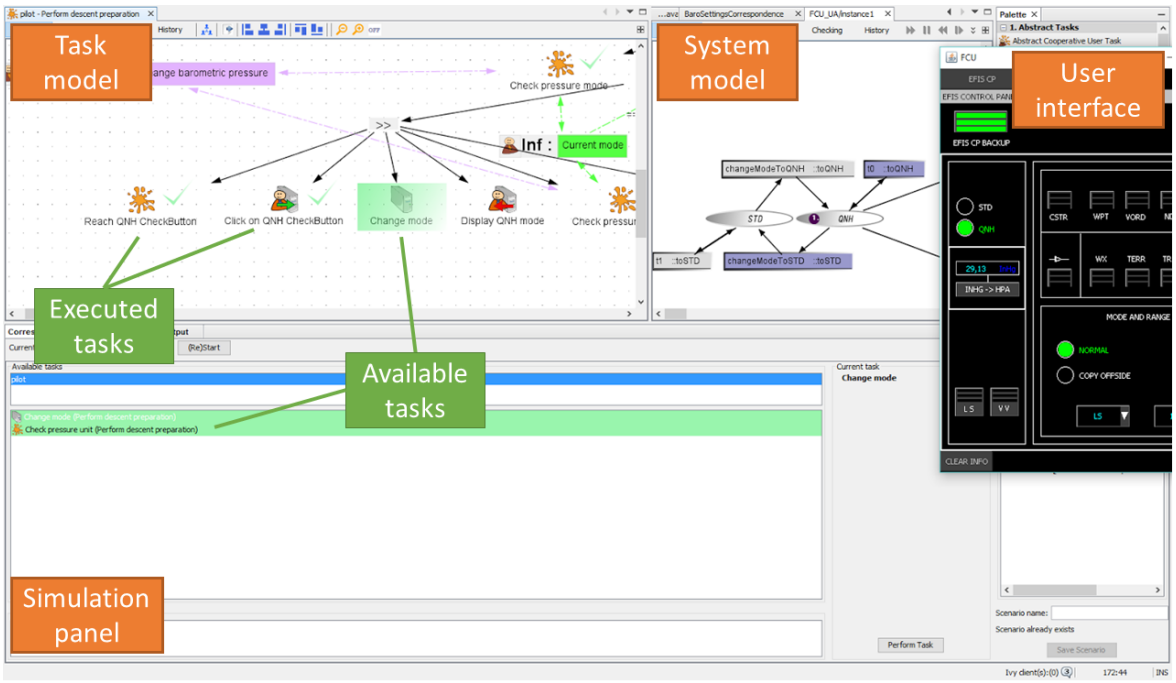

Fig. 11. SWAN example of co-execution.

R1: reinitialisability This is established by using the fact that the underlying Petri net is *live* and *bounded*.

R2: availability of buttons The *"STD"* and *"QNH"* radio buttons will always be active: the two corresponding events (*"stdClick"* and *"qnhClick"*) will remain available whatever action is triggered. This is because at least one transition corresponding to the availability of these events (the transitions *"changePressureMode_3"* and *"changePressureMode_4"* for the event *"stdClick"* and the transitions *"changePressureMode_1"* and *"changePressureMode_2"* for the event *"qnhClick"*) will always be available.

R3: mutual exclusion The *"STD"* and *"QNH"* modes are in mutual exclusion: the pair of places *"STD"* and *"QNH"* are part of a place invariant marked by only one token and therefore will always hold one (and only one) token. The two transitions in the model, *"changePressureMode_1"* and *"changePressureMode_2"*, correspond to the same event *"qnhClick"*. This could potentially lead to non-determinism in the model. However, as *"changePressureMode_1"* has place *"STD"* as input place and *"changePressureMode_2"* has place *"QNH"* as input place, non-determinism is avoided due to the mutual exclusive marking of these places. Whatever the evolution of the Petri net, there will always be exactly one token in one of these two places. There are four transitions connected to these places; two associated with the event click on the graphical widget STD (see event ::stdClick on the right-hand side of the name of the transitions) and two associated with the event click on the graphical widget QNH (see event ::qkhClick on the right-hand side of the name of the transitions). Following commoner's theorem (see [78]) this demonstrates that whatever state the Petri net is in, two transitions will be available (one with ::stdClick event and one with ::qnhClick event) which demonstrates that both QNH and STD buttons on the User interfaces will always be available.

R4: reversibility The models presented in Figure 7 do not provide enough information to demonstrate reversibility of actions at a generic level. At a specific level reversibility can be performed, for instance between STD and QNH states, as it is always possible to go from one of these states to the other one. More detailed modeling of the EditBoxNumeric could have represented the fact that an input that is not validated triggers the system back to the previously validated value (see for instance [31] page 175 Figure 9.15).

5.4.2 Analysis of the compatibility between the task models and the user interface models. This analysis step gathers assurances that the task model and the formal model of the user interface behavior are complete and consistent together (thus helping to guarantee that procedures followed by the operators are correctly supported by the system). This may be used, for instance, to guarantee the effectiveness factor of usability as defined in the ISO standard[9] as demonstrated in [32]. The SWAN tool provides several functionalities to perform and support this analysis.

First, it enables the editing of correspondences between the system model elements (e.g. places or transitions) and task model elements (e.g. interactive input and output tasks). The editing of correspondences between the task model (see Figure 6) and the ICO model (see Figure 7) of the case study is done by a dedicated editor (this is shown in the detailed description[10]). This editor connects interactive input tasks (from the task model) and system inputs (transitions from the system model) as well as system outputs (places from the system model) with interactive output tasks (from the task model).

The co-execution of models may be launched (via the co-execution panel at the end of the lower right-hand part), even if correspondence editing is not completed. Figure 11 depicts the co-execution of the system models (right part), the task models (left part) and the visible part of the application of the case study (right part). As explained before, the co-execution control panel stands in the lower part. The co-execution can be task driven (as depicted in Figure 11) or system driven.

When using the task driven co-execution, the user selects the available tasks (from the task model) through the co-execution control panel and the corresponding modification is done within the system model. For instance, in Figure 11, the "Click on QNH checkbutton" task has just been executed, on the system side. This corresponds to the fact that the "ChangePressureMode_1" transition has been fired, thus the token contained in the "STD" place transfers within the "QNH" place. At the same time, the user interface is updated (as presented in Figure 11 where the QNH checkbutton is selected, changing the previous STD selection).

The SWAN tool detects an error if the corresponding system element cannot be executed. Thus it provides support for validation as it makes it possible to find inconsistencies between the two models, e.g., sequences of user actions that should be available but are not because of inadequate system design. This feature allows the SWAN tool to provide support for automated scenario-based testing of an interactive application [16].

Summary of analysis and verification criteria

verification type: analysis is limited to static properties of the underlying Petri net; no representation of requirements; simulation-based analysis through model animation;

verification technology: theorem proving on the ICO model using invariants, traps and siphons calculation; limited to the underlying Petri net of the ICO model;

scalability of the analysis: no limitation on scalability as matrix transformations is used; scalability issue as properties have to be expressed outside of CIRCUS and there is no support to the analyst to check them;

user interface testing: automated execution of input test sequences recorded during interactions with the prototype, either from tasks model or from ICO models;

[9] https://www.iso.org/obp/ui

[10] https://sites.google.com/view/fcusoftware

support for the analysis of the wider socio-technical system: modeling of integrated views of the three elements of socio-technical systems (organization, human and interactive systems); however, FRAM[11]-based description of organization and variability of performance has only been addressed at a model level and not at a tool level (see [58]).

6 EVALUATION OF PVSIO-WEB

6.1 General aspects of PVSio-web

PVSio-web combines a theorem proving assistant PVS [70] with a web based environment. The result is a tool that enables the creation of realistic user interface prototypes that can be verified and validated against usability and safety requirements. A plug-in architecture allows developers to extend the tool with additional features, e.g., to introduce new prototyping front-ends, extend the widgets library, or introduce support for new verification technologies. The process supported by the tool includes the following steps:

Step 1. Define the behavior of the prototype. An executable PVS model is created that defines the logic of operation of the system. The model's structure follows an action-based pattern: a collection of state attributes characterizes the state of the prototype, and a set of transition functions over the state attributes defines which user actions are supported by the prototype. The Emucharts graphical editor provided by PVSio-web can be used to complete this step. Developers who are familiar with the PVS notation can also use the native Emacs-based model editor provided by PVS.

Step 2. Define the visual appearance of the prototype. The layout of the user interface and the characteristics (position, size, etc.) of widgets are defined. PVSio-web provides a Prototype Builder environment that can be used to complete this step. Developers who are familiar with web-based programming languages (HTML5 and JavaScript), can also use standard web development tools.

Step 3. Model validation. Developers can: 1) check the well-formedness of the model in the PVS theorem prover; and 2) assess the accuracy of the behavior of the prototype with respect to the real system by interacting with the prototype or by analyzing plausibility properties in PVS.

Step 4. Formal analysis. Properties of the model representing, e.g., safety requirements, are analyzed by means of simulation and theorem proving. The aim of simulation is to: 1) establish a common understanding among all stakeholders (developers, end users, safety bodies, etc.) of the characteristics of the system and the meaning of the requirements; and 2) perform lightweight formal verification based on exploration of relevant test scenarios, e.g., based on the execution of sample input key sequences demonstrating situations where a given requirement is either satisfied or fails. The formal analysis complements and extends the simulation-based analysis by allowing developers to check that requirements and properties of the model are satisfied in all reachable model states for all scenarios. In the current version of the toolkit, the formal analysis is carried out using the PVS theorem proving assistant.

Summary criteria

scope/purpose: user interface prototyping and analysis;
support for multi-disciplinarity: software engineers and human factors via prototypes;
related development process: user centered design, model based design;
tool features: interaction logic modeling, rapid prototyping of user interface software, verification of safety requirements and usability properties, code generation and documentation;
tool extensibility: PVSio-web has a plug-in based architecture that enables the rapid introduction of new modeling, prototyping, and analysis tools;

[11]Functional Resonance Analysis Method [47]

Prerequisites: state machines, PVS higher order logic and PVS theorem proving (only required for full formal verification);
IDE instance and principle: web technologies;
IDE availability: open source, downloadable at http://www.pvsioweb.org.

6.2 Modeling features of PVSio-web

The preliminary model of the proposed design of the EFIS is developed using Emucharts. This model serves the same purpose as a sketch prototype and does not require an understanding of PVS. This model is used to define the overall modal behavior of the data entry system. An enhanced model can then be developed from this initial model using PVS. The more detailed model describes accurately the following features: the modal behavior 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). For the purposes of illustration this model is configured as a collection of sub-models linked by import relations. Both the graphical Emucharts Editor and the PVS Emacs editor were used to develop the models.

The following fragments illustrate the models produced.

Modal behavior of the data entry system. The Emucharts diagram show in Figure 12 includes:

- 3 modes of operation (color-coded boxes labeled `STD`, `QNH`, and `EDIT PRESSURE`);
- 23 user actions and automatic data entry events (e.g., time-out timers);
- 9 status variables, representing the state of the system (units, display value, programed value, etc.).

The initial system mode is `STD`, denoted graphically by an arrow originating from a solid circle and entering the labeled box `STD`. In STD mode, the pilot can only change units. This is modeled using two actions, `click_hPa` and `click_inHg`, that leave and re-enter `STD`. These actions update the state variable corresponding to the pressure value based on the selected units (either hectopascal or inches of mercury). An action `click_QNH_RADIO` from `STD` to `QNH` represents the effect of the pilot selecting a different mode (`QNH`) which enables editing of the pressure value. A corresponding action `click_STD_RADIO` changes the data entry mode from `QNH` back to `STD`. An action `click_editbox_pressure` from `QNH` to `EDIT_PRESSURE` models the beginning of the interactive data entry phase, which is initiated by pilots when they select field `QNH` on the FCU. When in `EDIT_PRESSURE` mode, actions `click_digit_0` ... `click_digit_9` model the effect of pressing a button on the numeric keypad. Pressing the decimal point is modeled with `click_point`. Action `click_ESC` allows pilots to terminate data entry and restore the pressure value that was stored in the system before starting data entry. Action `click_CLR` models the effect of pressing the clear button, which resets the display value to zero. A further action `tick` models internal timers used by the data entry system to handle inactivity – data entry is automatically terminated and the input is discarded if key presses are not registered for 60 seconds. PVSio-web includes a model generator that translates the Emucharts diagram into a PVS model. This can be used a starting point for the enhanced model while at the same time producing an executable form of the model that can provide the basis for the sketch prototype.

Numerical computations of the data entry system. The PVS model defines how the display value is updated in `EDIT PRESSURE` mode, when the pilot interacts with the numeric keypad. The model includes a function `processKey` (see Listing 1) that computes the new display value based on the registered key press (function argument `key` of type `KEY_CODE`) and the current state of the data entry system (function argument `st` of type `state`).

Type `KEY_CODE` is an enumerated type adopted to assign unique identifiers (`KEY_1`, `KEY_2`, etc.) to each key. The body of function `processKey` is a switch statement `COND-ENDCOND` containing conditional expressions in the form *cond* → *expr*, where *cond* indicates the switch condition and *expr* indicates the expression to be evaluated when the condition is true. A function `digit?` checks whether the key code is a numeric key.

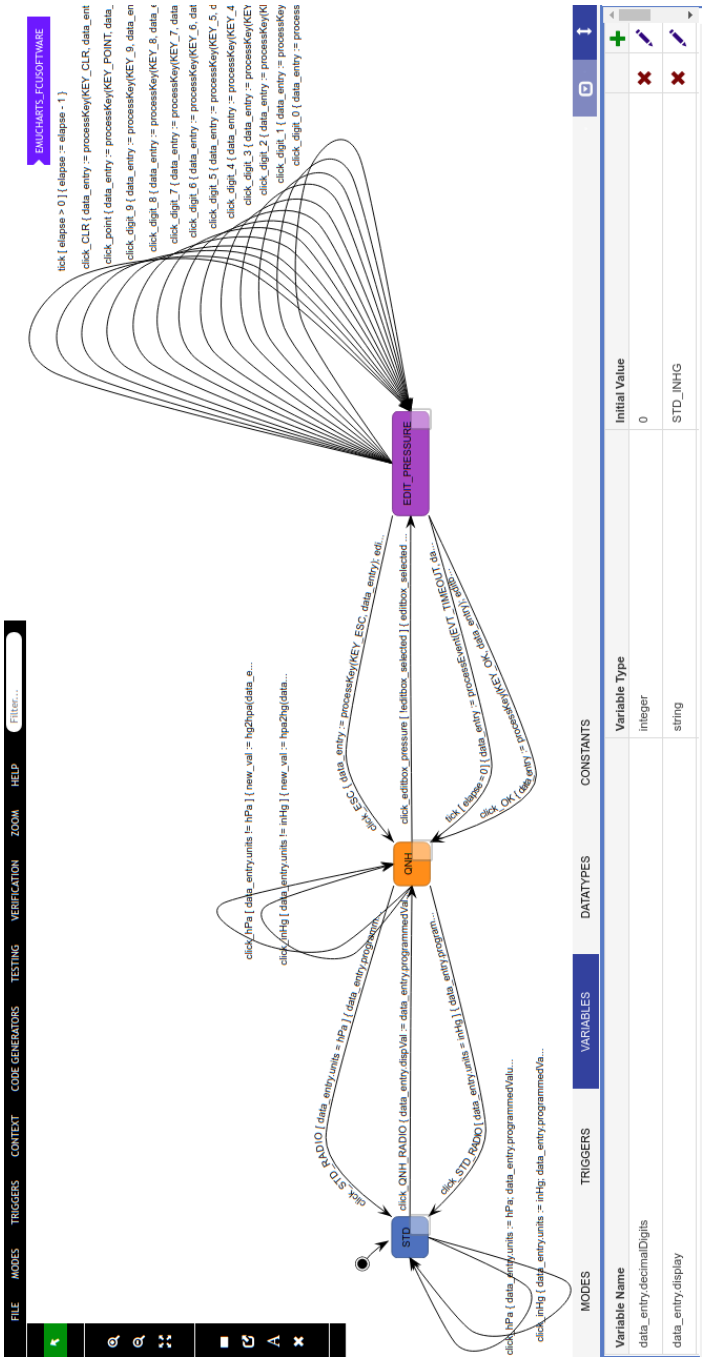

Fig. 12. Emuchart Editor with the diagram of the FCU data entry system.

```
processKey(key:KEY_CODE, st:state): state =
 COND
  digit?(key) -> click_digit(key)(st),
  key = KEY_POINT -> click_POINT(st),
  key = KEY_OK -> validate_data_entry(st),
  key = KEY_CLR -> clear_data_entry(st),
  key = KEY_ESC -> restore_display(st)
 ENDCOND
```

Listing 1. Function processKey in PVS.

Type `state` is a PVS record type with seven attributes (see Listing 2): a string `display` represents the content of the display during data entry; a non-negative real number `dispval` represents the numeric value shown in the display; a Boolean flag `pointEntered` indicates whether a decimal point key press has been registered; a non-negative real number `programmedValue` represents the pressure value currently stored in the system; a field `units` indicates the current units (either `inHg` or `hPa`); and two bounded natural numbers (`upto(MAX)` is a shorthand for identifying natural numbers up to a value `MAX`), `integerDigits` and `decimalDigits`, define how many integer and fractional digits have been registered. These two last attributes are necessary to limit the number of digits that can be entered during data entry, and thus avoid exceeding the display capabilities.

```
1 state: TYPE = [#
2   display: string,
3   dispVal: nonneg_real,
4   pointEntered: bool,
5   programmedValue: nonneg_real,
6   units: UnitsType,
7   decimalDigits: upto(MAX_DDIGITS),
8   integerDigits: upto(MAX_IDIGITS)
9 #]
```

Listing 2. Definition of type `state` in PVS.

The function `click_POINT` illustrates some of the fundamental features of the PVS language and some aspects of the adopted modeling patterns. The other functions defined in the PVS theory have a similar structure.

```
1 click_POINT(st: state): state =
2   IF pointEntered(st) THEN st
3   ELSE st WITH [
4     pointEntered := TRUE,
5     display := display(st) + "."
6   ] ENDIF
```

Listing 3. Definition of function `click_POINT`

The function has one argument, `st` of type `state`, representing the current state of the data entry system. The function returns the state of the data entry system after the execution of the function. The modeled behavior is as follows. If the decimal point has already been registered (this is checked by assessing the value of flag `pointEntered` in the current state, see line 2 in Listing 3), the state of the system does not change[12]. Otherwise, if the decimal point has *not* been entered, the `pointEntered` flag is set to `TRUE`, and a decimal point is concatenated

[12]Ignoring decimal point presses after the first decimal point has been entered ensures that the display value is always well-formed (i.e., the display value has at most one decimal point).

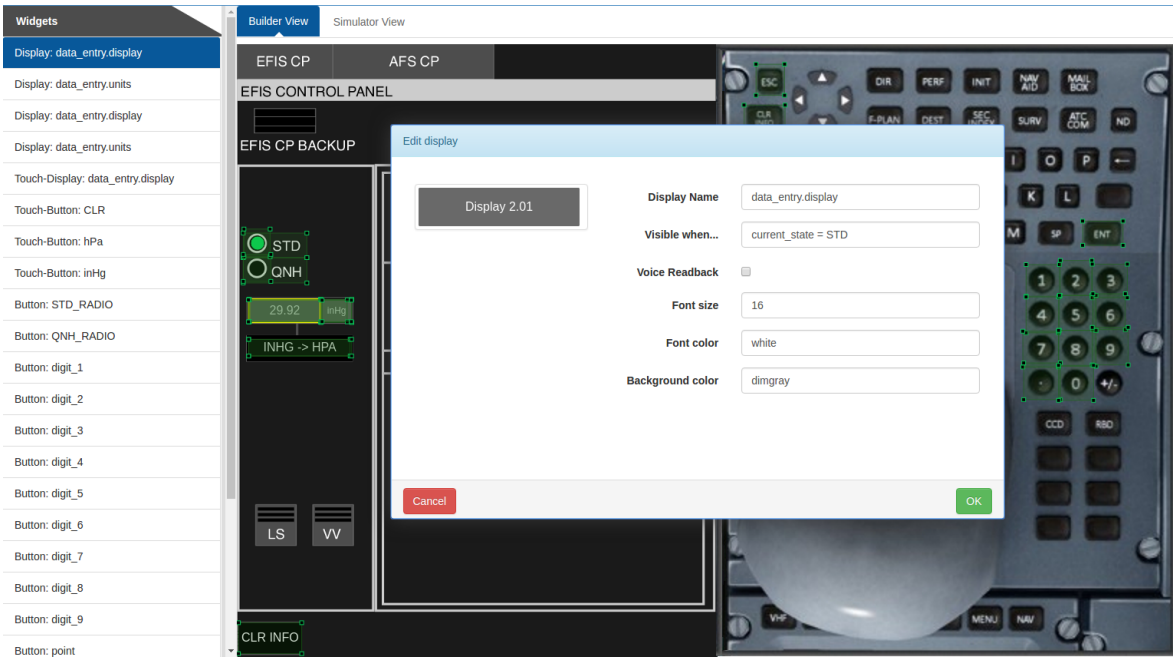

Fig. 13. PVSio-web Prototype Builder while developing the EFIS prototype.

to the string currently shown on the display. In the PVS language, string concatenation is obtained using the addition operator (+).

Summary modeling criteria

Modeling language: Emucharts, PVS;
Modeling paradigm: event-based, state-based, functional;
Structuring models: module-based;
Model editing features: graphical and textual editing of models, automatic generation of PVS models;
Suggestions for model improvements: strengthening of pre- and post- conditions of transition functions (based on proof obligations generated by PVS);

6.3 Prototyping features of PVSio-web

A visual prototype can be built whose behavior is driven by the formal models described in the previous subsection. The two approaches are illustrated. The PVSio-web Prototype Builder is designed to aid the construction of sketch prototypes while the alternative requires the use of standard web development tools and enables the development of functional prototypes that allow a more comprehensive customization of the front-end of the prototype.

Prototype Builder. A picture of the EFIS panel and of the Keyboard and Cursor Control Unit are loaded into the PVSio-web Prototype Builder (see Figure 13). Interactive areas are then created over relevant buttons and display elements (see Figure 12). Fifteen input areas were created over the picture of the Keyboard and Cursor Control Unit, to capture user actions over number pad keys, action keys (ENT, CLR, ESC), and the units conversion

button. Four display elements were created for rendering relevant status variables of the PVS model: two pairs of overlapping display elements are used to model the rendering of the value and units of the barometric pressure in STD mode and in QNH mode. Overlapping displays were necessary because in QNH mode the display elements change visual appearance and become responsive to tap actions performed over the element. These tap actions trigger the evaluation in the PVSio-web backend of function `click_editbox_pressure`. Two LED elements render the status of the STD and QNH CheckButtons based on the current mode of operation of the system.

Web Development Tools. A web page is created that uses a picture of the EFIS panel as a basis for the prototype. This is done by creating an HTML file with a section that includes a standard tag <img> (see Listing 4). The style of the section is used to render the image at a fixed position. The JavaScript module imported at the end of the HTML file (using the tag element <script>) contains the definitions of the widgets for the prototype. These definitions are constructed using the APIs provided by the PVSio-web widgets library.

```
<html>
<div style="position:absolute; top:0px; left:0px;">
     <img src="EFIS.png"></img></div>
<script type="text/javascript" src="require.js" data-main="index.js"></script>
</html>
```

Listing 4. File `index.html` used for importing the background image of the prototype and the JavaScript file with the widgets definitions.

An example widget definition is in Listing 5. It uses a constructor `BasicDisplay` to create the display element associated to the data entry display. The first parameter of the constructor is a unique identifier for the display element. The second parameter is a JavaScript object specifying the position and size of the display. The third parameter is another JavaScript object specifying optional characteristics of the display. Different widget types support different types of options. The options of the widget display allow developers to change the visual appearance of the button (font size, font color, and background color), specify the state variable in the PVS model associated with the display (`data_entry.display` in the example shown in Listing 5), and specify when the widget is visible (in this case, when the system is mode `STD`).

```
var display_val = new BasicDisplay("display_val",
                    { top: 333, left: 16, height: 28, width: 100 },
                    { fontSize: 16,
                      fontColor: "white",
                      backgroundColor: "dimgray",
                      displayKey: "data_entry.display",
                      visibleWhen: "mode = STD" });
```

Listing 5. Example display widget created using the APIs of the PVSio-web widgets library

The visual appearance of all widgets is refreshed every time the state of the PVS model changes. This is done in a function `onMessageReceived`, which is automatically invoked by PVSio-web when the PVSio-web back-end evaluates a new system state. The function has two parameters (see Listing 6): `err`, which is non-null only if an error occurred in the back-end; and `event`, a JavaScript object containing the textual string returned by the PVSio-web back-end. A state parser method provided by PVSio-web parses the string and transforms it into a JavaScript object. This object is then passed as parameter to the `render` method of each widget.

```
function onMessageReceived(err, event) {
   if (!err) {
      var state = stateParser.parse(event);
      display_val.render(state);
```

```
        ...
    }
    ...
}
```

Listing 6. Function used to refresh the visual appearance of widgets based on the current state of the PVS model

Summary prototyping criteria

support for prototype building: visual editing, based on a picture of the real system;
execution environment of the prototype: Javascript execution environment, Lisp;
human-machine interaction techniques: Pre-WIMP, WIMP, post-WIMP, multimodal;
code generation: 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)

6.4 Analysis and verification features of PVSio-web

Model Validation. The initial verification effort is dedicated to checking well-formedness of the model and ensuring that the behavior of the model is accurate with respect to the behavior of the real system.

Well-formedness is assessed by discharging proof obligations that are automatically generated by the PVS theorem prover when type checking the model. These proof obligations ensure coverage of conditions, disjointness of conditions, and correct use of data types. The PVS model generated from the Emuchart generates 22 proof obligations. The PVS model developed by hand generated 53 proof obligations. All the proof obligations were discharged automatically by the PVS theorem prover thereby providing assurance that the developed model of the FCU is well-formed.

Accuracy of the model's behavior can be assessed manually, by interacting with the prototype, that is by pressing buttons on the prototype user interface, and watching the results of the interaction using the prototype's displays. The prototype can also be assessed through usability evaluation, using appropriate communities of, for example, pilots (see [42] for example). More exhaustive forms of validation based, for example, on co-simulation of the model with the real system, could be realized using the PVSio-web infrastructure, but require substantial effort.

Formal Analysis. Further analysis can be used to prove that requirements, either established during the requirements elicitation process or as the design develops and details emerge, are true of the model. These requirements include use centered requirements as discussed in [43], in particular use centered safety requirements specified by external regulators (see [46, 59]). To perform the analysis, relevant requirements need to be translated into PVS theorems. PVSio-web helps developers to define and analyze use related safety properties. A property template editor allows developers to choose a property template, and to instantiate the template's parameters for the PVS model under analysis. A reversibility template allows easy formulation of the **R4** requirement described in Section 4 for example.

Formal analysis of the requirements described in Section 4 can be done using more details of the model described in Section 6.2. Further information is required about the moding behavior of the proposed design. A module which contains the data entry theory used for illustration in in Section 6.2 includes the data entry state. The attribute `data_entry` of `State` was used when discussing the data entry theory.

```
1 State: TYPE = [#
2     mode: Mode,
3     editbox_selected: bool,
```

```
    elapse: int,
    data_entry: state
  #]
```

Listing 7. Definition of type State in PVS.

The mode of the FCU is represented by the type: `Mode: TYPE = { EDIT_PRESSURE, QNH, STD }`. An action that describes the effect of selecting the STD button therefore is described by the function:

```
per_click_STD_RADIO(st: State): bool =
  (mode(st) = QNH AND
         (data_entry(st)`units = hPa )) OR
         (mode(st) = QNH AND ( data_entry(st)`units = inHg ))

click_STD_RADIO(st: (per_click_STD_RADIO)): State =
    COND
     mode(st) = QNH AND ( data_entry(st)`units = hPa )
      -> LET st = leave(QNH)(st),
              st = st WITH [ data_entry := data_entry(st)
              WITH [ programmedValue := STD_HPA ]],
              st = st WITH [ data_entry := data_entry(st)
              WITH [ dispVal := STD_HPA ]],
              st = st WITH [ data_entry := data_entry(st)
              WITH [ display := trim(STD_HPA) ]]
          IN enter(STD)(st),
     mode(st) = QNH AND ( data_entry(st)`units = inHg )
      -> LET st = leave(QNH)(st),
              st = st WITH [ data_entry := data_entry(st)
              WITH [ programmedValue := STD_INHG ]],
              st = st WITH [ data_entry := data_entry(st)
              WITH [ dispVal := STD_INHG ]],
              st = st WITH [ data_entry := data_entry(st)
              WITH [ display := trim(STD_INHG) ]]
          IN enter(STD)(st),
     ELSE -> st
    ENDCOND
```

Listing 8. Definition of function click_STD_RADIO

Two functions are described in Listing 8. The boolean function per_click_STD_RADIO specifies when the action is permitted to occur while click_STD_RADIO specifies the effect of pressing the button. Additions to the specification such as these are sufficient information to enable an understanding of the PVS theorems that are required to prove the requirements (R1-R4) of Section 4. The PVS theorems relating to R1-R3 are simply stated while **R4** is used to provide some insight into the proof process.

R1: reinitialisability.

This requirement is formulated in the PVS theorem specified in Listing 9.

```
REINITIALISIBILITY: THEOREM
FORALL (pre, post: State):
     ((mode(pre) = QNH AND per_click_STD_RADIO(post)
     AND post = click_STD_RADIO(pre))
```

```
5       IMPLIES (mode(post) = STD)) AND
6       (((mode(pre) =  EDIT_PRESSURE) AND
7       (post = click_STD_RADIO(click_ESC(pre))))
8       IMPLIES (mode(post) = STD))
```

Listing 9. R1: reinitialisability

The "initial" state has mode STD. When the mode is QNH then the initial state can be reached by pressing the STD button. When the mode is EDIT_PRESSURE, that is the pressure value is being edited, then an additional step is required to leave that mode. For the purpose of the proof the ESC button is used and the theorem proves true. It is clear therefore that the requirement, specified in Section 4, of a single action is not satisfied. The failure of the requirement could trigger discussion about whether the EDIT_PRESSURE mode can be treated as a special case. It should be noted that the failure of this property does not occur in the case of the ICO specification (Section 5.4.1) because the model in that case contains less detail relating to data entry. Similar issues arise in proving the other requirements.

R2: availability of buttons.

This property checks that STD and QNH are always available. In fact this is not the case. When editing the pressure, as discussed in relation to **R1** it is necessary to exit the mode first.

```
1 QNH_STD_available(st: State): boolean =
2 (mode(st) = STD OR per_click_STD_RADIO(st)) AND
3 (mode(st) = QNH OR per_click_QNH_RADIO(st))
4 AVAILABILITY: THEOREM
5 FORALL (x: real, pre, post: State):
6 QNH_STD_available(init(x)) AND
7 ((trans(pre, post) AND (post /=  click_editbox_pressure(pre)) AND
8 QNH_STD_available(pre)) IMPLIES QNH_STD_available(post))
```

Listing 10. R2: availability of buttons

This theorem (which can be found in Listing 10) requires a structural induction as will be discussed in more detail when addressing the requirement **R4**. A function QNH_STD_available is formulated to simplify the expression of the theorem. This property is not true of *all* states but it is true of all states that can be reached from the initial state init(x), and by any of the states that can be reached by using any permitted action (these states are linked by the relation trans). The proof of a theorem such as this one is discussed in relation to the requirement **R4**.

R3: mutual exclusion.

This requirement is formulated by PVS theorems (Listing 11) that treat each mode separately. The STD version of the theorem is illustrated here. It also uses a structural induction. Here we want to prove that for all *accessible* paths, if no explicit action is taken to change mode then the FCU will stay in the same mode.

```
1 EXCLUSIVITY_STD: THEOREM
2 FORALL (pre, post: State):
3       (trans(pre, post)
4       AND post /= click_QNH_RADIO(pre) AND mode(pre) = STD)
5       IMPLIES mode(pre) = mode(post)
```

Listing 11. R3: mutual exclusion (STD)

R4: Reversibility of user actions.

An action is reversible if the effect of the action on relevant state variables can be undone by some next action. For the FCU data entry system, reversibility can be used to check, for example, that the escape (ESC) button

always allows pilots to exit data entry and revert the pressure value to the value before starting data entry. In PVS, the property is expressed in Listing 12:

```
reversibility(st: State): bool =
(per_click_editbox_pressure(st) IMPLIES
 (per_click_ESC(click_editbox_pressure(st))
    AND
    click_ESC(
     click_editbox_pressure(st))`current_state
        = st`current_state))
```

Listing 12. R4: Function used in reversibility

To prove the property for all inputs and all states, the PVS theorem in Listing 13 is formulated:

```
REVERSIBILITY: THEOREM
FORALL (pre, post: State):
(init?(pre) IMPLIES reversibility(pre))
 AND ((reversibility(pre) AND
       trans(pre, post)) IMPLIES
                   reversibility(post))
```

Listing 13. R4: reversibility

The PVS theorem uses structural induction. As already mentioned in relation to **R2** and **R3** the property must hold for the initial system state, and given a generic state (`pre`) for which the property is true, then the property is true of the next state (post) reached through any of the transitions available in that state. A predicate `trans` identifies which transitions are available in a given state (`pre`), and uses the transition relation to link the given state (`pre`) to the next state (`post`). PVSio-web can generate the PVS theorem automatically, based on instantiations indicated by the developer through the template dialog. The tool also generates a PVS tactic for checking the property in the theorem prover. The tactic in this case involves the following steps:

- The first step is to remove the universal quantifier. This is performed using the PVS command `skosimp*`
- The second step is splitting the formula, to treat separately the base case of the induction and the induction step.
- The base case of the induction can be proved by using the predefined PVS strategy `grind`, which performs automatic expansion of definitions and propositional simplification.
- The induction step is proved by case splitting the available transition functions (this is done by expanding predicate trans and the permission predicates for each transition action), and then applying `grind` to all generated sub-goals.

This tactic is expressed as follows, using the language provided by the PVS proofLite extension:

```
REVERSIBILITY: PROOF
(then
(skosimp*)
(branch (prop)
 ((then(comment "induction base")
       (try (grind :if-match nil)
            (propax) (postpone)))
  (then(comment "inductive step")
       (expand "trans")
       (expand "per_click_editbox_pressure")
```

```
      (prop)
      (try (grind :if-match nil)
           (propax) (postpone)))))) QED
```

Using the tactic above, the PVS theorem prover is able to demonstrate that the reversibility property is true of the model for all inputs in all states. The proof was completed in 28.56 seconds on an Intel i5 processor. Note that, in the general case, the tactic above may not be sufficient to prove the property, e.g., because additional lemmas and assumptions need to be used. In that case, the proof is interactive, and carried out using the PVS theorem prover.

Summary of analysis and verification criteria

verification type: functional analysis, including: coverage of conditions, disjointness of conditions, correct use of data types, compliance with design requirements; simulation-based analysis through model animation;
verification technology: theorem proving; interactive simulations;
scalability of the analysis: user interface prototype of stand-alone devices, range of mainly medical examples see pvsioweb.org.
user interface testing: automated execution of input test sequences recorded during interactions with the prototype;
support for the analysis of the wider socio-technical system: modeling patterns based on distributed cognition theory have been explored in PVS but are not currently integrated in the IDE.

7 EVALUATION OF IVY

IVY is a tool that provides a front-end to the NuSMV model checker [23]. This front-end is designed to enable the analysis of interactive systems. The tools supported provide a modeling notation and the means of proving properties and diagnosing property failure.

7.1 General aspects of IVY

IVY enables the modeling and analysis of user actions and feedback attributes of critical user interfaces. The workflow supported by the tool includes four main steps:

Step 1. Modeling the interactive system. The focus of the model of the interactive system is to specify the user actions and the attributes of the states that are changed by the actions. Some of these state attributes are perceivable (usually visible). However a key aspect of the analysis is to understand the attributes that are not visible or understood in the context of the action. It is also important to recognize when and how an action is affected by an attribute that may or may not be perceivable. For example, for certain values, the action may not be permitted or its effect (its *mode*) may change. Finally not all actions are user initiated. Some are autonomous and these actions, that occur in the background, may have an effect on user action and/or may require particular types of response from the user. These actions must be understood and made explicit in the model. The focus of the model, and therefore the modeling language, is to specify actions and attributes (and whether they are perceivable). The simple notation used (Modal Action Logic) follows this pattern and is therefore considered to be easy to use.

Step 2: Validating the model. An important first step in the analysis process is to check the veracity of the model. This can be done in two ways using the IVY tool. The first is to check properties that show how sequences supported by the model reflect precisely the actual or planned behavior of the physical implementation. This is done by demonstrating that defined goals can be reached. A typical property checks that it is *never* possible to reach a goal, with the aim that the property fails and a counter-example (a sequence of states) is produced that shows one way in which the goal *can* be reached. This trace can be animated using IVY's animator and

alternative paths may also be investigated. The traces may then be explored dynamically and compared with sequences of actions in the intended or modeled actual design.

Step 3: Formulating properties. Properties of the model include those that capture assumptions about the perceivability of the state attributes and the potential ambiguity of actions. The IVY tool provides a set of property templates that can be used to simplify the process of formulating these properties. They relate to issues such as the visibility of the effect of an action, the transparency of the model of the system and the ease with which an action or actions can be reversed.

Further analysis involves scrutiny of paths that achieve activity goals when actions are constrained by information resources. This process is typically interdisciplinary, involving communication between modeling, domain and human factors experts. The IVY tool is designed to produce representations of relevant sequences that can be readily understood by this mix of disciplines.

Step 4: Visualizing traces. The process of analysis is algorithmic. The advantage of this is that proof is automatic (as opposed to interactive, as in the case of PVSio-web). The disadvantage is that because the approach aims to provide an exhaustive analysis of the states of the model, and there can be many states of the model, this may lead to a slow or intractable process for some model checkers. A typical proof process is interactive. A property is checked and the counter-example is scrutinized if it fails. Failure may arise because the property has been incorrectly formulated, or because the model does not capture the properties of the actual device or it may indicate that the device does not exhibit the required property. In this case, the exception may be acceptable and therefore be excluded in the property or it may indicate a flaw in the design.

The IVY tool has a plug-in architecture which enables communication with individual tools. Tools are currently under development to support simulation and prototypes based on models. There is also a tool to convert an Emuchart specification into a MAL model.

Summary criteria

scope/purpose: user interface software modeling and analysis;
support for multi-disciplinarity: Software engineers, HCI specialists via formalized usability heuristics;
related development process: user centered design, heuristic evaluation;
tool features: logic modeling, verification of safety requirements and usability properties, verification results (counter-examples) analysis;
tool extensibility: IVY uses a plug-in based architecture that facilitates the introduction of new functionalities; plug-ins need only comply with a predefined API, supporting both communication and information sharing;
Prerequisites: behavioral modelling, temporal logic (to express more complex properties, when the use of patterns is not feasible);
IDE instance and principle: Java;
IDE availability: downloadable at http://ivy.di.uminho.pt.

7.2 Modeling features of IVY

The modeling of the FCU using MAL is focused on the actions that are supported by the device, both actions that can be carried out by users and autonomous actions that do not require user initiative but will have effects that will require interpretation or action on the part of the user. The logic provides:

- a modal operator [_]_ : [*ac*]*expr* is the value of *expr* after the occurrence of action *ac* — the modal operator is used to define the effect of actions;
- a special reference event []: []*expr* is the value of *expr* in the initial state(s) — the reference event is used to define the initial state(s);

- a deontic operator *per*: *per*(*ac*) meaning action *ac* is permitted to happen next — to control when actions might happen;
- a deontic operator *obl*: *obl*(*ac*) meaning action *ac* is obliged to happen some time in the future. Note that *obl* is not used in this analysis.

The modal operator makes it possible to prescribe the effect of actions in the state but says nothing about when actions are permitted or required to happen. For this, permission and obligation operators must be used. As in [82], only the assertion of permissions and the denial of obligations are considered:

- per(*ac*) → *guard* — action *ac* is permitted only if *guard* is true;
- *cond* → obl(*ac*) — if *cond* is true then action *ac* becomes obligatory.

Permissions are asserted therefore by default and obligations are off by default. This makes it easier to add permissions and obligations incrementally when writing specifications. For example, the two permission axioms per(*ac*) → *guard1* and per(*ac*) → *guard2* together yield: per(*ac*) → (*guard*1 & *guard*2) (note that & is used to denote logical *and* — | for logical *or*, and ! for *not*). This logic is particularly appropriate for describing a system in which components can be reused.

The interactor presentation is defined by annotating actions and attributes to show that they are perceivable. The modality of the perceivable attribute/action is given using further attributes. For example `[vis]` asserts that the attribute/action is visibly perceivable. In addition if attached to an action it can be invoked by the user. Additional annotations are introduced for further modalities.

Attributes and action parameters are typed. Types are represented as enumerations of the "key values" or as subranges of integer:

```
types
 T_enum = { a, b, c }
 T_range = { 0 .. 10 }
```

The notation also supports the usual propositional operators. The fragments of the model of the example FCU below illustrate the style of the specification. This model was actually developed from the same Emuchart as the PVS model described in Section 6.2. The `main` interactor includes a further interactor `FCUDataEntry` that describes data entry properties. In fact this interactor is simplified because MAL does not handle real values and mode properties are the main focus of concern in the analysis of the requirements **R1-R4** of Section 4. The use of the inclusion `aggregates` specifies that elements within this included interactor are to be referred to by `fcud`. Attributes `st`, `msg`, `display`, `editboxselected`, and `elapse` are associated with types: some are standard types, for example `boolean`, the others are user defined.

```
interactor main
aggregates
      FCUDataEntry via fcud
  attributes
    st: MachineState
    msg: MsgType
    editboxselected: boolean
    display: string
    elapse: int
  actions
    clickCLR
    ... more defs. omitted for brevity ...
```

The fragment used for illustration specifies an action clickCLR. The action is permitted only if st = EDITPRESSURE.

```
per(clickCLR) ->
      (st = EDITPRESSURE)
```

The action clears the values of the displayed pressure to blank. The numbers in the case of this model have been simplified to make analysis tractable. The attribute elapse is used to determine the time between user actions and keep preserves the value of the specified attributes across the state transition. If there is no constraint on an attribute then it will change randomly.

```
[clickCLR]
    display[id]'=blank &
    display[pt]' = blank &
    display[dd]' = blank &
    elapse' = MAXELAPSE &
    keep(msg,st,editboxselected)
```

There is a further action that occurs as a result of triggering clickCLR. This is achieved by introducing an assertion that states that either action clickCLR or clickeditboxpressure occurs if and only if fcud.processclrKey is triggered. Here the prefix fcud ... indicates that the action is specified in the interactor FCUDataEntry. The equivalence is expressed as follows.

```
 effect(fcud.processclrKey) <->
 (effect(clickCLR) | effect(clickeditboxpressure))
```

The action therefore is defined to invoke the corresponding action in the data entry interactor and also to update the displayed value (the array display). The user has performed an action and so the elapsed time since the last user action is set to MAXELAPSE. The final part of the specification indicates that the remaining state attributes do not change value.

More details of MAL can be found in [17, 18, 20]. This specification captures, in simple terms, the interactive behavior of the FCU. A further fragment of the model is shown, within the IVY editor, in Figure 14.

The activity that is being modeled in this case has already been represented as a HAMSTERS task model in Section 5.2.1. The specific concern is with descent (though presumably the activity is similar in the case of climb). The cockpit receives a barometric target that consists of a value and an explicit notification of units. The target must be memorized and, if it differs from the current settings as displayed in the FCU, the information is used to make a change. This involves changing the value and potentially changing the units. The resources in this case involve two elements that must be remembered. The activity that is captured in this illustrative example is to change the barometric pressure: changepressure. The resources that are used in this process are the pressure value and the pressure units: mvalue and munits.

Summary modeling criteria

Modeling language: MAL;
Modeling paradigm: modal logic;
Structuring models: module-based;
Model editing features: text editor supporting model hierarchy, syntax highlighting and code completion;
Suggestions for model improvements: no explicit suggestions;

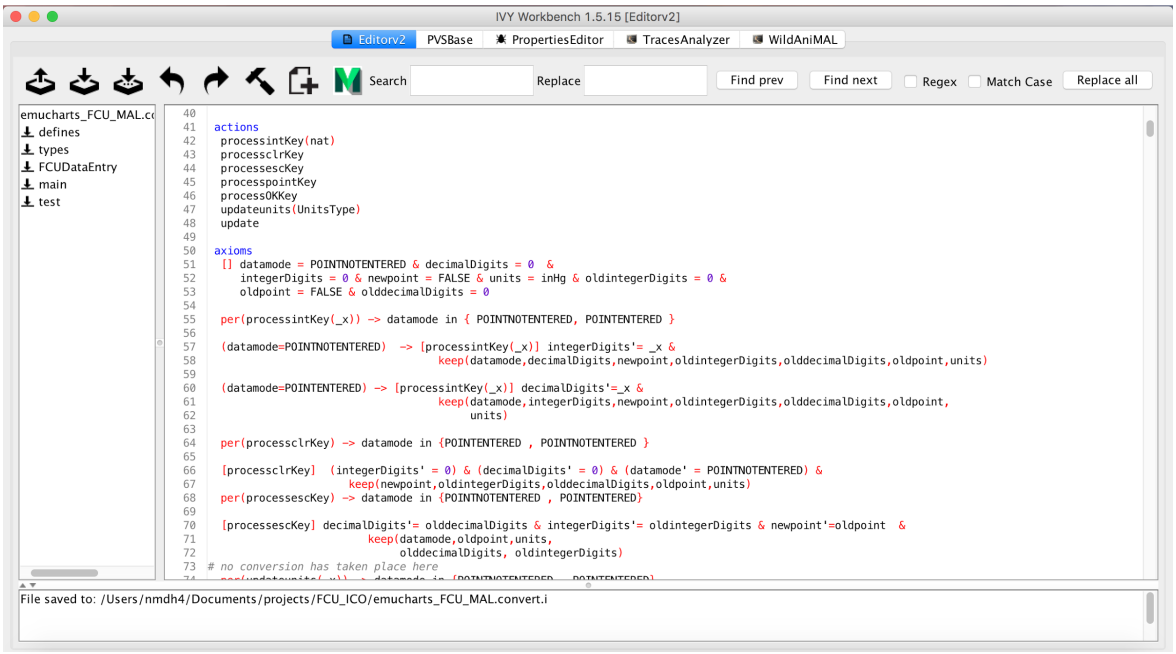

Fig. 14. The IVY editor showing a fragment of the model.

7.3 Prototyping and simulation features of IVY

IVY's simulation capabilities are designed to support an understanding of the behavior of the system from the perspective of the modeler or analyst. The goal is to support model validation as well as the exploration of alternative scenarios during counter-example analysis. The simulator therefore provides views of how the state of the system evolves over time rather than a traditional prototypical view of what the user interface will look like. Two views are provided: a tabular view and a state-based view (plus the log of the communication between IVY and NuSMV).

In the tabular view (see Figure 15) rows represent state attributes and action execution, and columns represent the states in a particular execution trace. Hence, in Figure 15 the sequence of actions (last row of the table) clickqhradio, clickeditboxpressure, clickdigitone and clickdigit zero have been executed. Actions are attached to the column that describes the target state of the transition associated with that action. Hence, nil, in column 1, indicates that in the initial state no action has been executed. As a consequence of these actions, the state of the FCU (row labeled st) changes from STD (in the initial state) to QNH to EDITPRESSURE. Yellow is used to highlight values that change with a state transition.

In the state-based view (see Figure 16) each interactor is represented as a lifeline. Each lifeline is the sequence of states of the execution trace for that interactor. Hence, in Figure 16 there is a lifeline for the main and fcud interactors. States are annotated with attributes and actions are represented as labels on the transitions between them.

Actions can be executed from the ACTIONS panel to the left. The panel shows all available actions at a given point. By selecting an action, the resulting state is presented in the STATE INFO panel. Executing it triggers

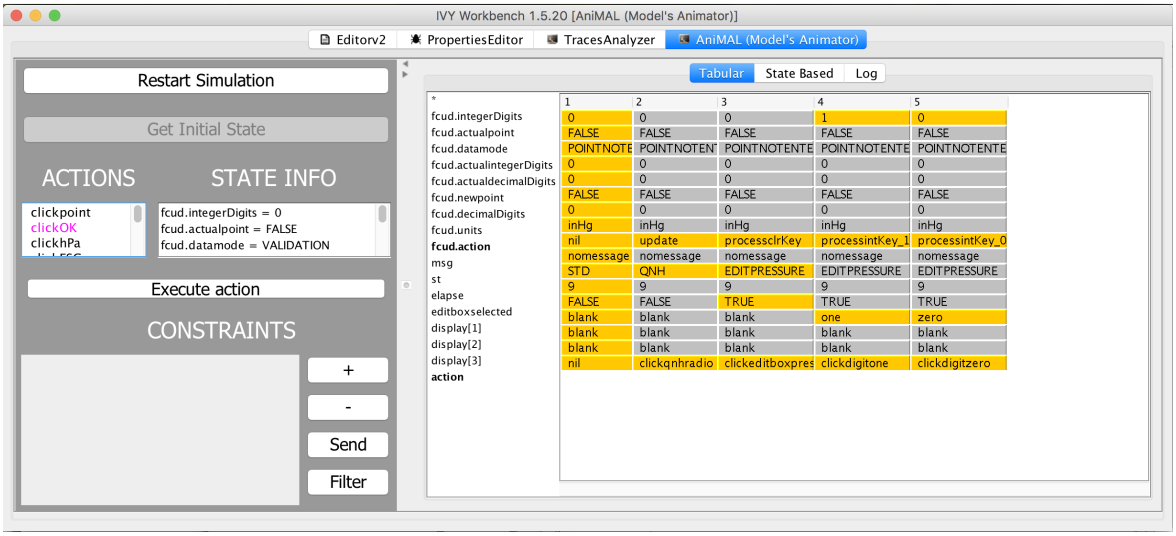

Fig. 15. IVY's animator tabular view

the corresponding transition in the model, updating tabular and model-based views and the ACTIONS panel contents.

MAL models are not necessarily deterministic and therefore the less specified the model, the larger the number of alternative behaviors. Such a model's action is likely to have more than one outcome. In the extreme, this can create conditions in which NuSMV is unable to provide the next possible states for the current state. To help mitigate this, it is possible to impose constraints on the generation of possible next states. This is done using the constraints panel (see bottom left of Figures 15 and 16).

The goal of the simulator is to support model validation. This is illustrated in Figure 16 where a situation is identified in which the model goes into a deadlock state. Only the nil action is available (see ACTIONS panel) and this indicates that no further action is possible at this stage. Executing the nil action, as illustrated in the figure, causes no change to the state. Investigation of clickOK, the last action to be executed, highlights an issue with the specification of the modal axiom, due to a modeling error committed by the developer. An alternative to using the simulator is to validate the model by attempting to prove basic properties of the model behavior. The model checker demonstrates counter-examples when a property fails. The benefit of the simulator is that alternative examples can be considered more flexibly thereby making it possible to recognize the appropriate qualification to properties more readily.

Summary prototyping criteria

support for prototype building: no facility for building prototypes is currently available (prototyping plug-in under development), animation of traces;
execution environment of the prototype: inside the tool (plug-in under development);
human-machine interaction techniques: pre-WIMP, WIMP, post-WIMP, multimodal;
code generation: run-time execution of the models via NuSMV simulation facilities, no code generation.

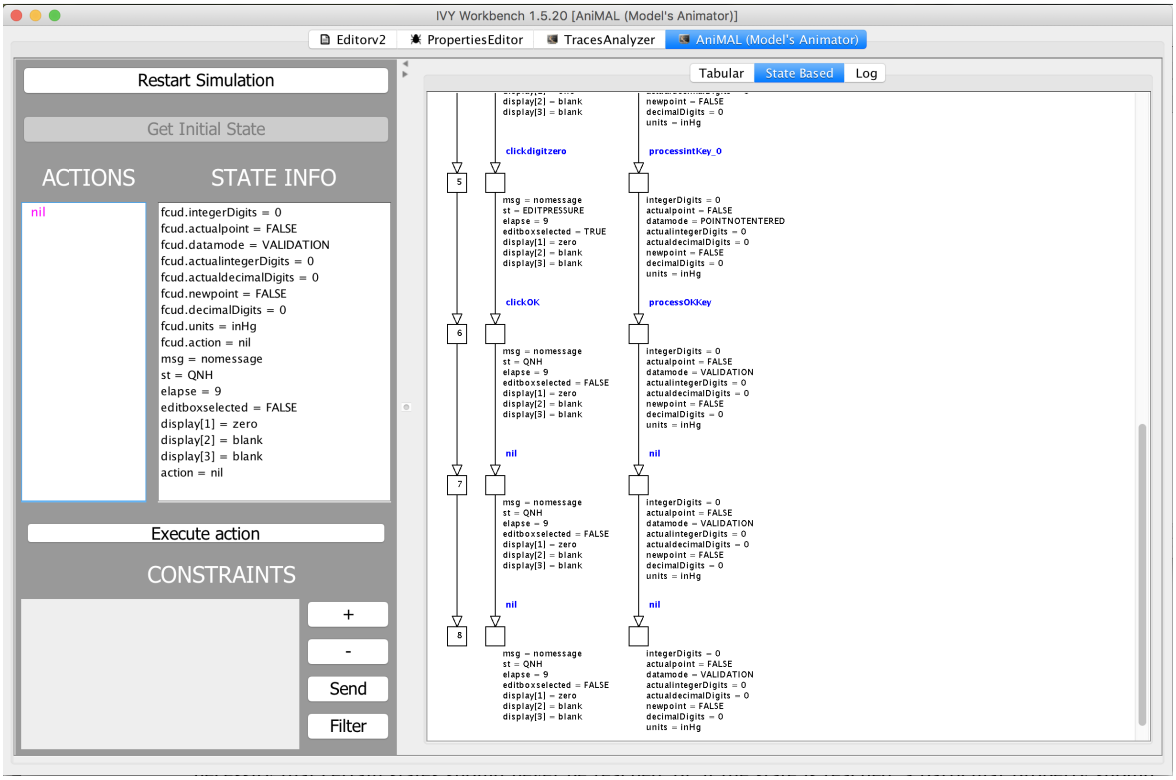

Fig. 16. IVY's animator state-based view

7.4 Analysis and verification features of IVY

Model validation. These properties include the necessity that certain states should never be reached, or, if the state is reached, a particular property should hold of the state. They are safety properties or they are liveness properties. CTL [24] enables the exploration of properties over the possible behaviors (paths) of a system.

CTL provides two kinds of temporal operator, operators over paths and operators over states. Paths represent the possible future behaviors of the system. When `p` is a property expressed over paths, `A(p)` expresses the property that `p` holds for all paths and `E(p)` that `p` holds for at least one path. Operators are also provided over states. When `q` and `s` are properties over states, `G(q)` expresses the property that `q` holds for all the states of the examined path; `F(q)` that `q` holds for some states over the examined path; `X(q)` expresses the property that `q` holds for the next state of the examined path; while `[qUs]` means that `q` holds until `s` holds in the path.

CTL contains a subset of the possible formulas that arise from the combination of these operators. `AG(q)` means that `q` holds for all the states of all the paths; `AF(q)` means that `q` holds for at least one state in all the paths; `EF(q)` means that `q` holds in at least one state in at least one path; `EG(q)` means that `q` holds for all states in at least one path; `AX(q)` means that `q` holds in the next state of all paths; `EX(q)` means that there is at least one path for which `q` holds in the next state; `A[qUs]` means that `q` holds until some other property `s` holds in all paths; `E[qUs]` means there exists at least one path in which `q` holds until some property `s`.

A typical property designed to assess the plausibility of the model is as follows.

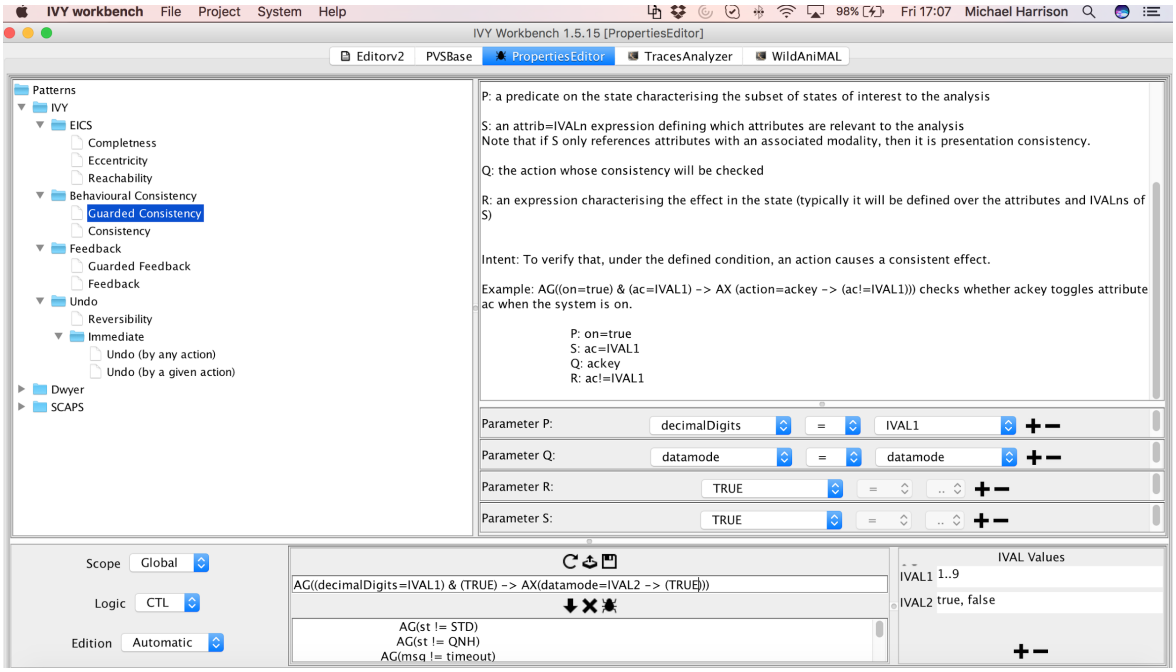

Fig. 17. Constructing guarded consistency properties

```
AG(!(display[id] = three &
     display[pt] = point &
     display[dd] = seven &
     fcud.actualintegerDigits = 3 &
     fcud.actualpoint &
     fcud.actualdecimalDigits = 7 &
     fcud.units = hPa))
```

This property asserts that there are no states of the model for which the number 3.7 is displayed and the internal stored actual value is also 3.7 and the atmospheric units are “hPa”. Checking this property yields an answer false with a counter-example represented by a trace (see Figure 18). The trace shows a sequence of states, numbered columns 1..8 and the actions and state attributes relative to the data entry interactor (fcud) and then the main interactor. The first main action, `Clickqnhrad` (column 2), changes state to `QNH` and ensures that the actual values are equal to the display values (`fcud.update`). The next step (column 3) involves the `clickeditbox` action that selects the edit box and clears the display. The next three states in the sequence 4..6 show the key strokes involved in entering “three”, “point” and “seven”. The last action before clicking “OK” changes the units of the pressure to `hPa`. Clicking “OK” sets the actual values of the device to the entered values. These actions and their effect are consistent with the behavior of the FCU.

To make MAL models simpler to understand, and CTL properties simpler to express, the notation allows definitions of enumerated sets and of expressions (for example the conjunctions of actions).

Formal Analysis. The requirements introduced in Section 4 can be expressed in CTL and proved of the model.

	1	2	3	4	5	6	7	8
fcud.action		update	processclrK	processintK	processpoir	updateunits	processintK	processOKK
actualdecimalDigits	0	0	0	0	0	0	0	7
actualintegerDigits	0	0	0	0	0	0	0	3
actualpoint	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE
datamode	POINTNOTE	POINTNOTE	POINTNOTE	POINTNOTE	POINTENTE	POINTENTE	POINTENTE	VALIDATION
decimalDigits	0	0	0	0	0	0	7	7
integerDigits	0	0	0	3	3	3	3	3
newpoint	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
units	inHg	inHg	inHg	inHg	inHg	hPa	hPa	hPa
main.action		clickqnhrad	clickeditbox	clickdigitthr	clickpoint	clickhPa	clickdigitsev	clickOK
display[1]	blank	blank	blank	three	three	three	three	three
display[2]	blank	blank	blank	blank	point	point	point	point
display[3]	blank	blank	blank	blank	blank	blank	seven	seven
editboxselected	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
elapse	9	9	9	9	9	9	9	9
msg	nomessage	nomessage	nomessage	nomessage	nomessage	nomessage	nomessage	nomessage
st	STD	QNH	EDITPRESSL	EDITPRESSL	EDITPRESSL	EDITPRESSL	EDITPRESSL	QNH

Fig. 18. Checking the plausibility of the MAL model

R1: reinitialisability. The property states that if the mode is not intially STD there always exists a path that leads eventually to a change of mode to STD.

```
AG(st != STD -> AX(EF(st = STD)))
```

R2: availability of buttons. The two properties expressing this requirement state that the actions that represent the buttons, namely `clickstdradio` and `clickqnhradio` will always immediately change the state to the required mode.

```
AG(st != STD -> AX(clickstdradio -> st = STD))
AG(st != QNH -> AX(clickqnhradio -> st = QNH))
```

R3: mutual exclusion. These properties state that if the mode is STD then the only action that will possibly change the mode is `clickqnhradio` and if the mode is QNH then the only actions that can change the mode are `clickstdradio` or `clickeditboxpressure`.

```
AG(st = STD -> AX(!clickqnhradio -> st = STD))
AG(st = QNH -> AX(!(clickstdradio | clickeditboxpressure) -> st = QNH))
```

R4: reversibility. The reversibility property is expressed in CTL as

```
AG(display[id] = $number1 &
   display[pt] = $pointval &
   display[dd] = $number2 ->
     AX(clickeditboxpressure ->
        EF(clickESC ->
           display[id] = $number1 &
           display[pt] = $pointval &
           display[dd] = $number2 ))
```

This property satisfying the **R4** requirement was created in the IVY property editor by instantiating the "reversibility" template with the relevant parameters, see Figure 19.

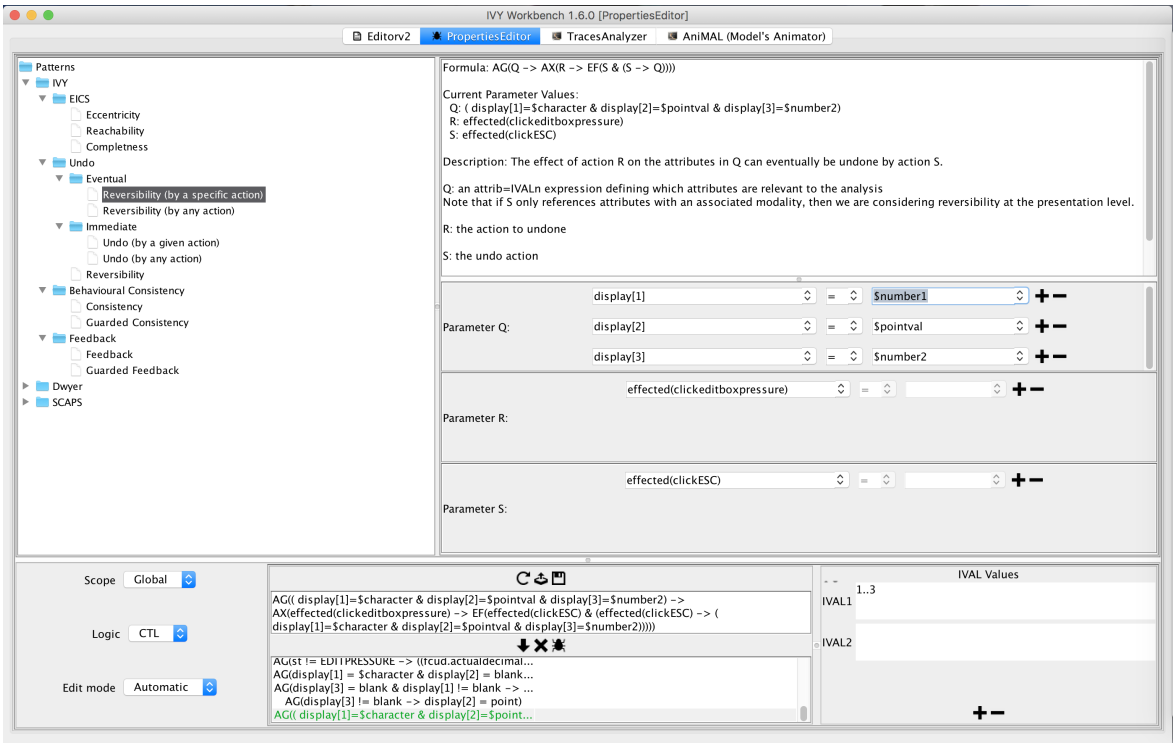

Fig. 19. Checking a reversibility property

Summary of analysis and verification criteria

verification type: well formedness of the model, requirements checking, simulation-based analysis through model animation;
verification technology: model checking;
scalability of the analysis: user interface prototype of stand-alone devices: medical devices and space systems.
user interface testing: automated execution of input test sequences recorded during interactions with the prototype;
support for the analysis of the wider socio-technical system: automated testing currently not supported, manual testing via simulation.

8 RESULTS

In this section, the results of the evaluation of the tools are discussed.

General aspects of the tools. From a high-level perspective, the scope of the three tools is the same — model-based analysis of critical user interfaces. The three tools support modeling and analysis of the interaction logic of the user interface. However, each tool offers a different modeling and analysis technology that is tailored to different, and complementary, styles of assessment of user interfaces. CIRCUS supports explicit modeling of user tasks and goals, allowing developers to simulate user tasks and check their compatibility with the interactive behavior of the system. PVSio-web supports rapid prototyping and formal verification of usability

and safety requirements. IVY facilitates modeling of general usability and safety properties. 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) and temporal logic (for IVY) is already sufficient to get started with illustrative examples. This is extremely useful to reduce the typical knowledge barriers faced by novice users that come from the software engineering community. The three tools are developed using standard technologies supported by multiple platforms (Netbeans Visual API for CIRCUS, Web technologies for PVSio-web, Java for IVY), and can be executed on a standard desktop/laptop computer.

Modeling. CIRCUS and PVSio-web provide graphical IDEs designed to assist developers in the creation of formal models. The notations used in IVY are textual and therefore the model editor is textual with some support for structure. CIRCUS uses specialized 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 modeling patterns to support the modeling process. Developers can use either a graphical notation (Emuchart 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 modeling by examples, checking how the Emucharts model translates into PVS; Emucharts models can be translated into popular formal modeling languages other 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 reuse. 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). The IVY tool is designed for simplicity. The MAL notation is designed to reflect a simple view of action and state attribute. A framework is provided for producing a constraint based model of the information resources used in interaction with the device that provides an alternative to the normative task modelling approach.

Prototyping. CIRCUS and PVSio-web 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). IVY has more limited prototyping features, and only mockup prototypes can be created using an experimental plugin. Thanks to their architecture that separates the formal model from the visual appearance of the prototype, all three tools promote the use of the Model-View-Controller (MVC [53]) paradigm, with a clear separation between the visual appearance of the prototype and the logic behavior. 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 behavior). Developers can define their own widgets library. For example, for the case study presented in Section 4, we created a library of widgets whose visual aspect and behavior is compatible with that described in the ARINC 661 standard. PVSio-web prototypes are developed in JavaScript, and their behavior 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 behavioral models developed using Emucharts [62]. A specialized 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 and verification. Multiple verification technologies are used in the three tools to enable the efficient analysis of human-machine interaction. They 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 analyze 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 provides initial support for 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 modeling 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.

The IVY tool supports an established set of property templates for the analysis of common usability concerns in critical systems. The property editor is designed to simplify the process of using these templates to generate properties that are specific to the requirements of the interactive system. Properties that can be represented include reachability properties that can only be proved using a theorem prover by representing possible sequences of interactions explicitly. This means that the analysis phase of the IVY process can be used to check the plausibility of the model where a prototyping approach would be required with PVSio-web. When properties fail to be true counter-examples are presented in a clear format that can be readily understood and acted upon.

Dealing with scale. The three tools have been applied to case studies based on real-world systems.

CIRCUS has been used on multiple case studies including Air Traffic Control workstations [55], large civil aircraft cockpit interactive applications [4] and satellite ground segments [55]. Each of these application domains brought specific concerns that resulted in extensions to both the notations and the tools. For the interactive cockpit work the modelling of the MPIA application compliant with ARINC 661 specification included more than 200 ICO models. This application brought interesting issues related to performance in the simulation of the models and Petshop kernel required refactoring to reach a 20ms response time on the user interface. These performance improvements have been reported in [7]. While each ICO model is usually small in size (this is due to the fact that the decomposition follows the object-oriented principles) the model of the ARINC 661 server was huge (hundreds of places and transitions). This required additional developments in PetShop for supporting

slicing of models (using layering of models) and the addition of virtual places (to reduce the length of the arcs by duplicating places close to the transitions using them). Descriptions of case studies and examples in CIRCUS are available[13].

PVSio-web and IVY have been used to analyze a range of medical devices. For example, in the case of PVS, full models of the interactive behavior of two infusion pumps (Alaris [43] and BBraun [41]) have been produced. The generated simulations provide full functionality for exploration of use. In the case of the IVY model, restrictions to the model relate to number entry. However full models of the devices' modal behavior were generated as described in [41]. The PVS model of the Alaris device, as discussed in [43], involves two main theories describing 38 pump actions and 18 user interface actions. The specification and theorems can be found on Github[14]. The analysis involved 138 theorems based on the templates. The theory files amount to approximately 4000 lines including comments, and the theorem files approximately 5000 lines. The run time for each proof is indicated in the template files. Times range from less than 1 second to 80 minutes. The PVS system was installed on an Apple Macbook Pro with a 2.9 GHz Intel Core i5.

The IVY tool was applied in the safety analysis of a neonatal dialysis machine [45]. In this case the model was based on the device's control table and currently forms part of the safety analysis. The model involved 682 lines, including 119 lines of state definitions and 152 lines of type and constant definitions. The model has not been decomposed into sub-models. However a model of the Alaris and BBraun models described in the previous paragraph had a similar structure to the PVS models. The development of the dialyser control component took about seven hours. It was possible to make most changes to the model and show the results interactively during meetings with the development team without disturbing the flow of the meeting. Hence the refinement of requirements and the careful analysis of the hazards were facilitated by the process. A set of 252 requirements were identified in the risk log of which 47 mitigations used the MAL analysis at least in part. The analysis involved 23 properties. These supported mitigations relating both to aspects of protection and design. On the rare occasions when it was not possible to refine a property during the meeting, for example when meta-attributes were required, this could be achieved within an hour outside the meeting. Verifying all the properties together on a MacBook Pro with Intel Core i5 clocked at 2.9GHz, with 8GB RAM and SSD memory, took 1.7 seconds.

9 OTHER RELATED TOOLS

The contribution of the three tool sets described in this paper is that they are distinctive in combining more than one element of the design process described in Section 3. Several tools exist that have value at the different stages of the process but do not link as effectively between elements of the design process.

9.1 Work and task analysis stage

ADEPT [51] is an early example of a tool that enabled the description of tasks. However this early tool is no longer available and provided no means of analysis of the tasks represented. More recently, tools have been developed that enable the analysis of task descriptions. For example MAESTRO [11] is concerned with "assuring fitness of purpose". It provides systematic guidance for the design and evaluation of work needs so as to produce designs that are fit for purpose. The technique develops a matrix that identifies clusterings of work functions in terms of their inputs and outputs. Other tools use task representations and use model checking analyses to explore the tasks, exploring the reachability of specific task goals and shared tasks (for example). In this category are tools produced by Mori and Paternò [64] using tools based on LOTOS, and Bolton [13] using tools based on EOFM (as task modelling languages) and SAL (as model checker). Paternò's work goes further developing a full range of tools covering a broader spectrum of the design process.

[13]https://www.irit.fr/recherches/ICS/documentation/
[14]http://github.com/haslab/hcispecs/archive/1.1.zip

An alternative approach at this early stage uses formal models to express assumptions about the context or the user. These assumptions are expressed as constraints that replace the task models and respond to criticisms that formal task representations such as those provide by HAMSTERS prescribe behaviors that do not necessarily represent typical user assumptions (see for example [87]). These constraints, referred to as resources, are described in a modelling context using IVY in [15]. These resources are assumed to be relevant when interacting with the system under analysis. A further layer of the model may be developed to incorporate these assumptions about the design. Resources may include the attributes that are specified in the model but may also include information that is not part of the device or specific user interface under consideration. It may also include, for example, assumptions about physical displays, labels or handy references to operating procedures. In addition to constraints based on attributes or other information, the specification may also include a specification of the assumed "activities" that the user is engaged in. Activities are defined as actions in the MAL model, they define the achievement of "goal" states and are also constrained by information resources.

Modeling may further include assumptions about the salience of information resources [44]. Both an understanding of resource and of salience may be part of the design process, driving a consideration of the role that actions play in the user's activities and what information is assumed to help the user choose actions that are appropriate to the use of the device. This layer of modeling is based on these activities subject to resource constraint rather than an explicit plan. It takes note of the approaches of cognitive work analysis [87] and distributed cognition [50]. These additional layers are to be seen therefore as an alternative to the task model described by the HAMSTER model of the CIRCUS toolset.

9.2 Early prototyping loop

Formal analysis tools, aimed at describing interfaces and systematically analyzing the properties of the interface, have been developed by a number of researchers. Degani [28] used statecharts [40] to describe interfaces. Thimbleby and others [36, 86] describe the interface as a graph, and prove graph properties of the available user actions also using MATLAB [85], while ADEPT [25] (a more recent tool called ADEPT than the task representation described in the previous section) describes interfaces as Labeled Transition Systems. Their focus is on matching operator understanding of an interface with the interface itself. The process therefore involves a model of operator assumptions. Analysis detects mismatches between the cognitive and system models. This work echoes earlier work of Rushby [81] and others who used murphi to perform a similar though more specific analysis. Rushby's work analyzes properties of a flight management system, exploring the potential for user error. Other work considers issues of salience [80] and integrates user errors [22] within a cognitive model. The latter work is performed as part of a hazard analysis. None of these techniques have been used to address problems of the scale of the tools compared in this paper.

Bowen and Reeves' work [14] separates the interactivity of the environment from its functionality. Their work employs a notation (Presentation Interaction Model - PIM) that describes the interactivity and Z or μcharts (a variant of statecharts [40]) that describes the functionality. A relation (Presentation Model Relation) is used to describe the relation between user interface and system functionality, and this relation provides a basis for refinement.

Other types of environment involving user interfaces have been a focus for research. For example, formal development environments have used Petri nets as a first stage in developing virtual environments. Initial work by Willans [88] has been extended in the Apex project [83]. Other research is concerned with functional mock-up interfaces involving the co-simulation of discrete logic models for controllers and continuous models based on differential equations (see for example [73, 74]). Matlab and Simulink have also been used to model these systems (for example [77]). Finally, industrial tools such as SCADE [21] and SCADE Display [54, 79] exploit a dataflow paradigm such as the one promoted by LUSTRE notation [27] that allow the checking of user interface properties.

However, these tools are rarely presented in scientific publications and thus hard to assess and compare with academic contributions.

10 CONCLUSIONS

In this paper, we presented a detailed evaluation of three tools for model-based analysis of critical user interfaces, CIRCUS, PVSio-web, and IVY. The evaluation results can be used by developers to understand which formal tool can be used most effectively for which kind of analysis of interactive systems. The tools were evaluated against 22 criteria covering important elements in user centered design. A common example was used to assess the tools against the identified criteria. The result of this comparison led to the conclusion that the three tools are complementary rather than competitive. Whilst they have roughly the same scope (formal development of critical user interfaces), the tools enable different kinds of modeling and analysis: CIRCUS is tailored to task modeling and analysis; PVSio-web is tailored to rapid prototyping using PVSio for formal verification; IVY is tailored to formal verification of general usability concerns. These three types of analysis are complementary, and provide different insights about how to develop high-confidence user interfaces which corresponds to real problem in industry particularly in the area of safety-critical systems. The three tools presented here provide formal modeling and formal analysis that go beyond HCI toolkit research that have been presented in the survey of [26]. The set of criteria presented in this paper have the overall objective of being descriptive, comparative and generative as expected from "good" models as argued in [9]. We have demonstrated their descriptive and comparative nature. We hope they will be used by HCI and Software Engineering researchers to contribute to the definition of formal approaches and tools for interactive systems design, development and evaluation. This is the way to go to increase deployment of HCI contributions in terms of interaction techniques to dependable and safe safety critical command and control systems.

REFERENCES

[1] J. Accot, S. Chatty, S. Maury, and P. Palanque. 1997. Formal transducers: Models of devices and building bricks for the design of highly interactive systems. In *Design, Specification and Verification of Interactive Systems'97, Proceedings of the Fourth International Eurographics Workshop, June 4-6, 1997, Granada, Spain (Eurographics)*, M. D. Harrison and J. C. Torres (Eds.). Springer, 143–159. https://doi.org/10.1007/978-3-7091-6878-3_10

[2] SAS Airbus. 2016. Airbus A380 Flight Crew Operating Manual. http://www.airbus.com/.

[3] Airlines Electronic Engineering Committee. 2002. ARINC 661 specification: Cockpit Display System Interfaces To User Systems. Aeronautical Radio Inc.

[4] E. Barboni, S. Conversy, D. Navarre, and P. Palanque. 2006. Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. In *Interactive Systems. Design, Specification, and Verification, 13th International Workshop, DSVIS 2006, Dublin, Ireland, July 26-28, 2006. Revised Papers*. Springer, 25–38. https://doi.org/10.1007/978-3-540-69554-7_3

[5] E. Barboni, S. Conversy, D. Navarre, and P. Palanque. 2007. Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. In *Interactive Systems. Design, Specification, and Verification*, G. Doherty and A. Blandford (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 25–38.

[6] E. Barboni, J.-F. Ladry, D. Navarre, P. Palanque, and M. 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, 165–174. https://doi.org/10.1145/1822018.1822043

[7] E. Barboni, D. Navarre, P. Palanque, and S. Basnyat. 2006. Exploitation of formal specification techniques for ARINC 661 interactive cockpit applications. In *Proceedings of the HCI Aero Conference (HCI Aero 06)*. Cepadues, 81–89.

[8] R. Bastide, D. Navarre, P. Palanque, A. Schyn, and P. Dragicevic. 2004. A Model-based Approach for Real-time Embedded Multimodal Systems in Military Aircrafts. In *Proceedings of the 6th International Conference on Multimodal Interfaces (ICMI '04)*. ACM, 243–250. https://doi.org/10.1145/1027933.1027974

[9] M. Beaudouin-Lafon. 2000. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proceedings of the CHI 2000 Conference on Human factors in computing systems, The Hague, The Netherlands, April 1-6, 2000*, T. Turner and G. Szwillus (Eds.). ACM, 446–453. https://doi.org/10.1145/332040.332473

[10] D. Billman, C. Fayollas, M. Feary, C. Martinie, and P. Palanque. 2016. Complementary Tools and Techniques for Supporting Fitness-for-Purpose of Interactive Critical Systems. In *International Conference on Human-Centred Software Engineering*. Springer, 181–202.

[11] D. Billman, S-C. Wu, and C. Fan. 2016. Representing work for device design and evaluation using biclustering. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, Vol. 60. SAGE Publications Sage CA: Los Angeles, CA, 138–142.

[12] B Boehm. 1986. A Spiral Model of Software Development and Enhancement. *SIGSOFT Softw. Eng. Notes* 11, 4 (Aug. 1986), 14–24. https://doi.org/10.1145/12944.12948

[13] M.L. Bolton, N. Jiménez, M.M. van Paassen, and M. Trujillo. 2014. Automatically Generating Specification Properties from Task Models for the verification of Human-Automation Interaction. *IEEE Transactions of Human Machine Systems* 44, 5 (2014), 561–575.

[14] J. Bowen and S. Reeves. 2017. Combining models for interactive system modelling. In *The Handbook of Formal Methods in Human-Computer Interaction*. Springer, 161–182.

[15] J. C. Campos, G. Doherty, and M. D. Harrison. 2014. Analysing interactive devices based on information resource constraints. *International Journal of Human Computer Studies* 72 (2014), 284–297.

[16] J. C. Campos, C. Fayollas, C. Martinie, D. Navarre, P. Palanque, and M. 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, 138–148. https://doi.org/10.1145/2933242.2948735

[17] J. C. Campos and M. D. Harrison. 2001. Model checking interactor specifications. *Automated Software Engineering* 8 (2001), 275–310.

[18] J. C. Campos and M. D. Harrison. 2008. Systematic analysis of control panel interfaces using formal tools. In *Interactive systems: Design, Specification and Verification, DSVIS '08 (LNCS)*, N. Graham and P. Palanque (Eds.). Springer, 72–85.

[19] J. C. Campos and M. 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, 35–44. https://doi.org/10.1145/1570433.1570442

[20] J. C. Campos and M. D. Harrison. 2009. Interaction engineering using the IVY tool. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, G. Calvary, T.C.N. Graham, and P. Gray (Eds.). ACM, 35–44.

[21] J. L. Camus. 2012. *SCADE: Implementation and Applications.* Published 2012 by ISTE Ltd, Chapter 6, 225–272.

[22] A. Cerone, P. A. Lindsay, and S. Connelly. 2005. Formal analysis of human-computer interaction using model-checking. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*. IEEE, 352–361.

[23] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*. Springer, 359–364.

[24] E. M. Clarke, O. Grumberg, and D. A. Peled. 1999. *Model Checking*. MIT Press.

[25] S. Combéfis, D. Giannakopoulou, and C. Pecheur. 2015. Automatic detection of potential automation surprises for ADEPT models. *IEEE Transactions on Human-Machine Systems* 46, 2 (2015), 267–278.

[26] Ledo D., Houben S., Vermeulen J., Marquardt N., Oehlberg L., and Greenberg S. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, Article Paper 36, 17 pages. https://doi.org/10.1145/3173574.3173610

[27] B. d'Ausbourg, C. Seguin, G. Durrieu, and P. Roché. 1998. Helping the Automated Validation Process of User Interfaces Systems. In *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering, ICSE 98, Kyoto, Japan, April 19-25, 1998*, K. Torii, K. Futatsugi, and R. A. Kemmerer (Eds.). IEEE Computer Society, 219–228. https://doi.org/10.1109/ICSE.1998.671121

[28] A. Degani. 2003. *Taming HAL: designing interfaces beyond 2001.* Palgrave, Macmillan.

[29] D. J. Duke and M. D. Harrison. 1993. Abstract Interaction Objects. *Computer Graphics Forum* 12, 3 (1993), 25–36.

[30] R. Fahssi, C. Martinie, and P. A. Palanque. 2015. Enhanced Task Modelling for Systematic Identification and Explicit Representation of Human Errors. In *Human-Computer Interaction - INTERACT 2015 - 15th IFIP TC 13 International Conference, Bamberg, Germany, September 14-18, 2015, Proceedings, Part IV (Lecture Notes in Computer Science)*, J. Abascal, S. D. J. Barbosa, M. Fetter, T. Gross, P. A. Palanque, and M. Winckler (Eds.), Vol. 9299. Springer, 192–212. https://doi.org/10.1007/978-3-319-22723-8_16

[31] Camille Fayollas. 2015. *Generic Software Architecture and Model-Based Approach for the Dependability of Interactive Critical Systems. (Architecture logicielle générique et approche à base de modèles pour la sûreté de fonctionnement des systèmes interactifs critiques).* Ph.D. Dissertation. University of Toulouse, France. https://tel.archives-ouvertes.fr/tel-01241504

[32] C. Fayollas, C. Martinie, D. Navarre, and P. Palanque. 2015. A Generic Approach for Assessing Compatibility Between Task Descriptions and Interactive Systems: Application to the Effectiveness of a Flight Control Unit. In *i-com: Vol. 14, No. 3.* De Gruyter, 170–191.

[33] C. Fayollas, C. Martinie, P. Palanque, E. Barboni, R. Fahssi, and A. Hamon. 2017. Exploiting Action Theory as a Framework for Analysis and Design of Formal Methods Approaches: Application to the CIRCUS Integrated Development Environment. In *The Handbook of Formal Methods in Human-Computer Interaction*, B. Weyers, J. Bowen, A. J. Dix, and P. Palanque (Eds.). Springer, 465–504.

[34] C. Fayollas, C. Martinie, P. Palanque, Y. Deleris, J.-C. Fabre, and D. 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, 198–209. https://doi.org/10.1109/EDCC.2014.17

[35] C. Fayollas, C. Martinie, P. Palanque, P. Masci, M.D. Harrison, J.C. Campos, and S.R. Silva. 2017. Evaluation of formal IDEs for human-machine interface design and analysis: the case of CIRCUS and PVSio-web. In *Proceedings of the Third Workshop on Formal Integrated Development Environment (Electronic Proceedings in Theoretical Computer Science)*, Vol. 240. 1–19. https://doi.org/10.4204/EPTCS.240.1

[36] A. Gimblett and H. W. Thimbleby. 2013. Applying theorem discovery to automatically find and check usability heuristics. In *Proceedings of the 5th ACM SIGCHI symposium on engineering interactive computing systems*. 101–106.

[37] A. Hamon, P. Palanque, J. L. Silva, Y. Deleris, and E. Barboni. 2013. Formal Description of Multi-touch Interactions. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*. ACM, 207–216. https://doi.org/10.1145/2494603.2480311

[38] A. Hamon, P. A. Palanque, and M. Cronel. 2015. Dependable multi-touch interactions in safety critical industrial contexts: Application to aeronautics. In *13th IEEE International Conference on Industrial Informatics, INDIN 2015, Cambridge, United Kingdom, July 22-24, 2015*. IEEE, 980–987. https://doi.org/10.1109/INDIN.2015.7281868

[39] A. Hamon, P. A. Palanque, M. Cronel, R. André, E. Barboni, and D. Navarre. 2014. Formal modelling of dynamic instantiation of input devices and interaction techniques: application to multi-touch interactions. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'14, Rome, Italy, June 17-20, 2014*, F. Paternò, C. Santoro, and J. Ziegler (Eds.). ACM, 173–178. https://doi.org/10.1145/2607023.2610286

[40] D. Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8 (1987), 231–274.

[41] M.D. Harrison, J.C. Campos, and P. Masci. 2015. Reusing models and properties in the analysis of similar interactive devices. *Innovations in Systems and Software Engineering* 11, 2 (June 2015), 95–111.

[42] M.D. Harrison, P. Masci, and J.C. Campos. 2018. Formal modelling as a component of user interface design. In *Software Technologies: Applications and Foundations STAF 2018 collocated workshops (revised selected papers) (Lecture Notes in Computer Science)*, M. Mazzara, I Ober, and G. Salaün (Eds.). Springer, 274–294.

[43] M.D. Harrison, P. Masci, and J.C. Campos. 2019. Verification Templates for the Analysis of User Interface Software Design. *IEEE Transactions on Software Engineering* 45, 8 (2019), 802–822.

[44] M. D. Harrison, J. C. Campos, R. Ruksenas, and P. Curzon. 2016. Modelling information resources and their salience in medical device design. In *EICS '16 Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, 194–203.

[45] M. D. Harrison, L. Freitas, M. Drinnan, J. C. Campos, P. Masci, C. di Maria, and M. Whitaker. 2019. Formal techniques in the safety analysis of software components of a new dialysis machine. *Science of Computer Programming* 175 (2019), 17 – 34. https://doi.org/10.1016/j.scico.2019.02.003

[46] M. D. Harrison, P. Masci, J. C. Campos, and P. Curzon. 2017. Verification of User Interface Software: the Example of Use-Related Safety Requirements and Programmable Medical Devices. *ACM Transactions on Human Machine Systems* 47, 6 (2017), 834–846. https://doi.org/10.1109/THMS.2017.2717910

[47] E. Hollnagel. 2017. *FRAM: the functional resonance analysis method: modelling complex socio-technical systems*. CRC Press.

[48] I. Horrocks. 1999. *Constructing the User Interface with Statecharts*. Addison-Wesley Longman Publishing Co., Inc.

[49] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. 1986. Reachability Trees for High-level Petri Nets. *Theor. Comput. Sci.* 45, 3 (1986), 261–292. https://doi.org/10.1016/0304-3975(86)90046-0

[50] E. Hutchins. 1994. *Cognition in the Wild*. MIT Press.

[51] P. Johnson, S. Wilson, P. Markopoulos, and J. Pycock. 1993. Adept: Advanced design environment for prototyping with task models. In *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*. 56.

[52] B. Kirwan and L. Ainsworth. 1992. *A Guide to Task Analysis*. Taylor and Francis.

[53] G. E. Krasner and S. T. Pope. 1988. A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming* 1, 3 (Aug. 1988), 26–49. http://dl.acm.org/citation.cfm?id=50757.50759

[54] T. Le Sergent, A. Bouakaz, and G. Goretkin. 2018. SCADE AADL.

[55] C. Martinie, E. Barboni, D. Navarre, P. Palanque, R. Fahssi, E. Poupart, and E. Cubero-Castan. 2014. Multi-models-based engineering of collaborative systems: application to collision avoidance operations for spacecraft. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'14, Rome, Italy, June 17-20, 2014*. 85–94. https://doi.org/10.1145/2607023.2607031

[56] C. Martinie, P. Palanque, and M. 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, 589–609. https://doi.org/10.1007/978-3-642-23765-2_40

[57] C. Martinie, P. A. Palanque, E. Bouzekri, A. Cockburn, A. Canny, and E. Barboni. 2019. Analysing and Demonstrating Tool-Supported Customizable Task Notations. *PACMHCI* 3, EICS (2019), 12:1–12:26. https://doi.org/10.1145/3331154

[58] C. Martinie, P. A. Palanque, M. Ragosta, M. A. Sujan, D. Navarre, and A. Pasquini. 2013. Understanding Functional Resonance through a Federation of Models: Preliminary Findings of an Avionics Case Study. In *Computer Safety, Reliability, and Security - 32nd International Conference, SAFECOMP 2013, Toulouse, France, September 24-27, 2013. Proceedings (Lecture Notes in Computer Science)*, F. Bitsch, J. Guiochet, and M. Kaâniche (Eds.), Vol. 8153. Springer, 216–227. https://doi.org/10.1007/978-3-642-40793-2_20

[59] P. Masci, A. Ayoub, P. Curzon, M.D. Harrison, I. Lee, O. Sokolsky, and H. Thimbleby. 2013. Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example. In *Proceedings ACM Symposium Engineering Interactive Systems (EICS 2013)*. Association of Computing Machinery, 81–90.

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

[61] P. Masci, P. Oladimeji, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby. 2015. PVSio-web 2.0: Joining PVS to HCI. In *Computer Aided Verification: 27th International Conference, CAV 2015, Proceedings, Part I*, D. Kroening and S. C. Păsăreanu (Eds.). Springer International Publishing, 470–478. https://doi.org/10.1007/978-3-319-21690-4_30 Tool available at http://www.pvsioweb.org.

[62] G. Mauro, H. Thimbleby, A. Domenici, and C. 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).

[63] A. F. Monk, M. Curry, and P. C. Wright. 1991. Why industry doesn't use the wonderful notations we researchers have given them to reason about their designs. In *User-centred requirements for software engineering*, D.J. Gilmore, R.L. Winder, and F. Detienne (Eds.). Springer, 185–189.

[64] G. Mori, F. Paternò, and C. Santoro. 2002. CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Transactions of Software Engineering* 28, 8 (2002), 797–813.

[65] C. A. Muñoz and R. Butler. 2003. Rapid prototyping in PVS. http://ntrs.nasa.gov/search.jsp?R=20040046914 NASA/CR-2003-212418, NIA Report No.2003-03.

[66] D. Navarre, P. Dragicevic, P. Palanque, R. Bastide, and A. Schyn. 2005. Very-High-Fidelity Prototyping for Both Presentation and Dialogue Parts of Multimodal Interactive Systems. In *Engineering Human Computer Interaction and Interactive Systems*, R. Bastide, P. Palanque, and J. Roth (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–199.

[67] D. Navarre, P. Palanque, R. Bastide, and O. Sy. 2001. A Model-Based Tool for Interactive Prototyping of Highly Interactive Applications. In *Proceedings of the 12th International Workshop on Rapid System Prototyping (RSP '01)*. IEEE Computer Society, 136–. http://dl.acm.org/citation.cfm?id=882480.883731

[68] D. Navarre, P. Palanque, J.-F. Ladry, and E. 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, Article 18 (Nov. 2009), 56 pages. https://doi.org/10.1145/1614390.1614393

[69] D. Navarre, P. A. Palanque, R. Bastide, and O. Sy. 2001. A Model-Based Tool for Interactive Prototyping of Highly Interactive Applications. In *12th IEEE International Workshop on Rapid System Prototyping (RSP 2001), 25-27 June 2001, Monterey, CA, USA*. IEEE Computer Society, 136–141. https://doi.org/10.1109/IWRSP.2001.933851

[70] S. Owre, J. M. Rushby, and N. Shankar. 1992. PVS: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction (CADE-11)*. Springer Berlin Heidelberg, 748–752. https://doi.org/10.1007/3-540-55602-8_217

[71] P. Palanque, E. Barboni, C. Martinie, D. Navarre, and M. Winckler. 2011. A Model-based Approach for Supporting Engineering Usability Evaluation of Interaction Techniques. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '11)*. ACM, 21–30. https://doi.org/10.1145/1996461.1996490

[72] P. A. Palanque and R. Bastide. 1997. Synergistic Modelling of Tasks, Users and Systems using Formal Specification Techniques. *Interacting with Computers* 9, 2 (1997), 129–153. https://doi.org/10.1016/S0953-5438(97)00013-1

[73] M. Palmieri, Cinzia B., and P. Masci. 2017. Co-simulation of semi-autonomous systems: the line follower robot case study. In *International Conference on Software Engineering and Formal Methods*. Springer, 423–437.

[74] M. Palmieri, C. Bernardeschi, and P. Masci. 2019. A framework for FMI-based co-simulation of human–machine interfaces. *Software and Systems Modeling* (2019), 1–23.

[75] M. Palmieri, C. Bernardeschi, and P. Masci. 2019, to appear. A Framework for FMI-based Co-Simulation of Human-Machine Interfaces. *Software and Systems Modeling* (2019, to appear).

[76] F. Paternò and G. Faconti. 1992. On the Use of LOTOS to Describe Graphical Interaction. In *People and Computers VII: HCI '92 Conference*, A. Monk, D. Diaper, and M. D. Harrison (Eds.). BCS HCI Specialist Group, Cambridge University Press, 155–174.

[77] A. Rajhans, S. Avadhanula, A. Chutinan, P. J. Mosterman, and F. Zhang. 2018. Graphical Hybrid Automata with Simulink and Stateflow. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*. 267–268.

[78] W. Reisig. 1985. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science, Vol. 4. Springer. https://doi.org/10.1007/978-3-642-69968-9

[79] V. Rossignol. 2009. Optimized Safety-Critical Embedded Display Development with OpenGL SC. *SAE International Journal of Aerospace* 2, 2009-01-3140 (2009), 91–94.

[80] R. Rukšėnas, J. Back, P. Curzon, and A. Blandford. 2009. Verification-Guided Modelling of Salience and Cognitive Load. *Formal Aspects of Computing* 21 (2009), 541–569.

[81] J. Rushby. 2002. Using Model Checking to Help Discover Mode Confusions and Other Automation Surprises. *Reliability Engineering and System Safety* 75, 2 (Feb. 2002), 167–177.

[82] M. Ryan, J. Fiadeiro, and T. Maibaum. 1991. Sharing Actions and Attributes in Modal Action Logic. In *Theoretical Aspects of Computer Software*. LNCS, Vol. 526. Springer, 569–593.

[83] J. L. Silva, J. C. Campos, and M. D. Harrison. 2014. Prototyping and analysing ubiquitous computing environments using multiple layers. *International Journal of Human Computer Studies* 72, 5 (2014), 488 – 506.
[84] J. L. Silva, C. Fayollas, A. Hamon, P. A. Palanque, C. Martinie, and E. Barboni. 2013. Analysis of WIMP and Post WIMP Interactive Systems based on Formal Specification. *ECEASST* 69 (2013). https://doi.org/10.14279/tuj.eceasst.69.967
[85] S. Simakov. 2005. *Introduction to MATLAB graphical user interfaces.* Technical Report. DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA) MARITIME ….
[86] H. W. Thimbleby. 2007. *Press on: principles of interaction programming.* MIT press.
[87] K. J. Vicente. 1999. *Cognitive Work Analysis.* Lawrence Erlbaum Associates.
[88] J.S. Willans and M.D. Harrison. 2001. A tool supported approach for designing and testing virtual environment interaction techniques. *International Journal of Human-Computer Studies* 55, 2 (2001), 145–165.
[89] F. Zambonelli, A. Omicini, B. Anzengruber, G. Castelli, F. L. De Angelis, G. Di Marzo Serugendo, S. Dobson, J. L. Fernandez-Marquez, A. Ferscha, M. Mamei, et al. 2015. Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive and Mobile Computing* 17 (2015), 236–252.