Formal Verification of UI Using the Power of a Recent Tool Suite

Raquel Oliveira [1,2,3] **Sophie Dupuy-Chessa** [1,2] **Gaëlle Calvary** [1,2]

[1] Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
[2] CNRS, LIG, F-38000 Grenoble, France
[3] Inria
Emails: FirstName.LastName@imag.fr

ABSTRACT
This paper presents an approach to verify the quality of user interfaces in the context of a critical system for nuclear power plants. The technique uses formal methods to perform verification. The user interfaces are described by means of a formal language called LNT and ergonomic properties are formally defined using temporal logics written in MCL language. Our approach moves towards the powerfulness of formal verification of user interfaces, thanks to recent tools to support the process.

Author Keywords
User interface; critical systems; formal verification; temporal logic; ergonomic properties; process language

ACM Classification Keywords
H.5.2. Information Interfaces and Presentation (e.g. HCI): User Interfaces; I.6.4. Simulation and Modelling: Model Validation and Analysis

INTRODUCTION
User interfaces (UI) play an important role in the Human Computer Interaction (HCI) [6], specially in safety-critical systems, where failures may have disastrous consequences. This calls for high quality of user interfaces, which can be ensured by several ways. For example, [13] proposes four ways of evaluation: *formally* by some analysis techniques, *automatically* by a computerized procedure, *empirically* by experiments with users and *heuristically* by looking at the UI and passing judgement according to an expert opinion.

Although each approach has advantages and drawbacks, formal verification is suitable for safety-critical systems [11]. It allows exhaustive reasoning on the system models, unveiling subtle bugs that could be undetectable by testing or by simulation. In [11], an experiment was performed to compare the effectiveness of formal verification and testing at discovering errors, showing how testing failed to find errors that were found by formal verification. Besides, user testing can be expensive [15] considering that users in safety-critical systems are highly specialized and their time has a high cost.

Formal verification approaches can be seen as a complement of classical testing techniques to ensure UI quality. It allows an exhaustive analysis of the system, handling complex cases that are difficult for a human to reason about. For this purpose, it requires a model of the system to be verified. The formal model is an abstraction of the system's behavior. It is crucial for the model to be a meaningful approximation of the system in order to have a useful evaluation [15]. By using formal verification one can avoid the need for a runnable version of the UI, thus enabling design errors to be detected earlier in the development cycle.

In this paper we propose a formal approach to ensure quality of user interfaces in safety-critical systems. We revisit some techniques proposed in the '90s ([14, 8]), with changes in several directions. Besides, we use the newest versions of tools specialized in formal verification, namely the toolbox CADP (*Construction and Analysis of Distributed Processes*) [7], the formal language LNT (*Lotos NT*) [16], and MCL (*Model Checking Language*) [10], a language to express temporal logic formulas. We illustrate how features added in recent versions of these tools facilitate the formal verification of UIs. Our ideas are being applied in an industrial case study in the nuclear power plant domain.

The reminder of this paper starts by giving an overview over several ways to ensure quality of UIs using formal methods. Then it will present our approach step by step. A case study on which the approach has been applied will then be described to illustrate model checking in action. Finally the conclusion will summarize our current results and propose some perspectives.

RELATED WORK
Several approaches [12, 5, 11, 17, 9, 14, 8] propose to ensure quality of user interfaces using formal verification. Originally formulated for the modelling of user interfaces, and nowadays covering the modelling of full interactive systems (not only the user interfaces), the ICO (*Interactive Cooperative Objects*) formalism [12] enables one to prototype and to verify applications before they are fully implemented. ICO uses concepts from object-oriented approach to describe the static aspects of systems, and it uses high-level Petri nets to describe their dynamic aspects. The specification is validated

using proof tools, by the analysis of Petri net properties. The use of Petri net properties has limitations [12]. In particular, the analysis is usually performed on the underlying Petri net (a simplified version of the original Petri net), and the verification of properties in underlying Petri nets does not imply that these properties also hold in the original Petri net.

Other approaches that use formal methods to verify safety-critical systems (in the avionic domain) are described in [5, 11, 17]. In [5] the authors explore the UIs-related causes of several air plane accidents. In the context of a NASA project, [11] proposes a framework to translate some graphical models (e.g. Simulink and Stateflow) into textual specifications that can be given as inputs to model checkers (e.g. NuSMV, BAT, Kind). These approaches do not verify properties over models, but rather offer a support for formal techniques. In [17], however, the authors propose the computer-aided verification of properties (written in CTL - *Computational Tree Logic*) over a safety-critical system model developed in the MAL (*Modal Action Logic*) Interactors language. Similarly, the approach described in [9] uses models that were originally developed using MAL, and proposes a method to verify some requirements in the context of medical devices regulators.

Closer to our approach than the aforementioned ones, in [14] (later enriched by [8]) the authors use model checking to verify properties of user interfaces. With this goal, the UIs are first represented by a CTT (*Concur task trees*), later used to generate a formal specification in the LOTOS formal language using the CTTE tool.

Once the formal model describing the UI behavior is created, properties that need to be verified on the UI are specified using the ACTL temporal logic. Properties like *continuous feedback*, *reachability*, *reversibility*, *etc.* are verified over the formal specification. Then the authors use CADP toolbox to verify the satisfiability of these properties on the UI model.

We revisit this technique using a more powerful support, which enlarges the possibilities of UI verification. The main differences between our approach and this one will be deeply detailed in the next sections.

OUR APPROACH

Our approach is illustrated in Figure 1. It consists in verifying properties over a formal model of the user interface. In the following sections, we detail it step by step.

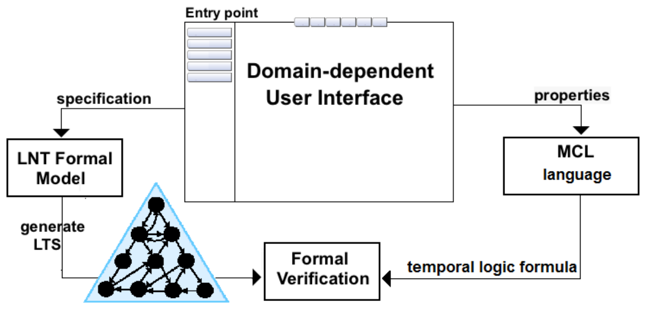


Figure 1: Formal verification of UI properties

Entry point

The starting point before any verification is to understand the UI in depth, i.e. the purpose and behavior of each UI element. This can be done using either the real system, or a prototype, or informal descriptions as entry point. We focus on the UI and thus consider only the UI and the parts of the functional core that have an impact on the UI behavior.

LNT Formal Model

Once the user interface behavior is well known, a model of it can be created. We use LNT to write the specification. Nowadays this model is manually written, in contrast to the work proposed in [14], that generates a LOTOS specification directly from a task model. In our case, a task model per se does not contain sufficient information to permit automatic generation of the formal model. It turns out that our formal model covers the user interface behavior and some aspects of the system's core. It is so written manually, to be as realist as possible.

The LNT model of the UIs is the first input of the formal verification in our approach. We apply model checking as technique, and for this purpose, a LTS (*labelled transition system*) of the formal model is needed. We use CADP [7] to generate the LTS from the LNT specification.

MCL Temporal Logic

With the UI formal model in hands, one can verify a set of properties on it. The right branch of Figure 1 illustrates this second input of our approach. A lot of works have been done [2, 1, 18, 13] to guide the identification of user interface properties. In our approach, the usability properties from the framework [1] was chosen. In contrast to [14], our approach suggest the usage of these ergonomic guidelines in order to identify UI properties, rather than defining them on demand.

In order to verify if the user interface satisfies the identified properties, the verification technique requires these properties to be written in a formal way too. We use MCL (*Model Checking Language*) to re-write them in a formal way.

Formal Verification

There are several techniques that can be used for verification, including (but not limited to) model checking, equivalence checking, visual checking [7]. Our approach applies model checking: we use CADP toolbox to reason over the LTS model of the UI and to verify properties satisfiability.

To end the process, as usual in model checking [14], once a property is not satisfied (meaning that it is false over the user interface in study), the tool provides a counter-example. A counter example is a set of ordered steps that should be followed, by interacting with the user interface, that leads to a UI state where the property is false. This diagnosis is one of the main benefits of using formal methods to verify UIs, furnishing a precise way to identify UI problems.

Advantages of our approach

The key enhancements brought by our approach is the usage of a more powerful support. In order to describe the UI behavior, we use the LNT formal language [16], which improves

LOTOS, and can be translated to LOTOS automatically. LOTOS is a formal description technique originally devised to support standardization of OSI (Open Systems Interconnection), but that has been used now more widely to model concurrent systems. In [14] the authors point out how difficult it is to model a system using LOTOS, when quite simple UI behaviors can easily generate complex LOTOS expressions.

Our approach alleviates this difficulty, by proposing the usage of LNT, which is a more intuitive language. In terms of expressiveness, LOTOS and LNT are equivalent, but they differ in terms of format and appearance. LOTOS consists of two orthogonal sub-languages: the data part, based on algebraic abstract data types (using equational programming style) and the control part, based on process algebra. In LNT, both parts (data and control) share a common syntax, using the imperative programming style (easier to learn and to read). In [16] the authors deeply argue about the benefits of LNT over LOTOS, notably the user friendliness and the richer data type definition, to mention only two advantages.

A user-friendly language decreases the learning curve of designers in the formal analysis domain, and it decreases the required labor time of writing a formal specification of the UI, enabling one to bypass the complexity of formal methods and more quickly take advantages of them.

The richer data type definitions of LNT permits more realistic UI models, thus widening the capabilities of verification, covering verifications on the data type of the UI fields, for instance.

Another point of improvement in our approach is the use of MCL to formalize the properties. MCL is more expressive than the ACTL logic used in [14]. As a matter of fact, MCL is an enhancement of the modal μ-calculus, a fixed point-based logic that subsumes all other temporal logics, aiming at improving the expressiveness and conciseness of formulas [10]. This allows us to identify, for example, the existence of complex unfair (infinite) cycles in the model's graph (i.e. the LTS generated from the formal model in Figure 1). An unfair cycle is an infinite sequence made by the concatenating sub-sequences satisfying the formula [10], e.g. a sequence of actions over the user interface that once started loops forever. For instance, in MCL it can be expressed that:

The UI will potentially respond (meaning provide a feedback) after ***at most*** *three user interactions (requests) occurring in any order.*

This is stated as follows in MCL:

$\nu Y(c : nat := 0).$

$\langle not(req_1 \vee req_2 \vee req_3)^* \,.\, resp\rangle true$

or

$((c < 3)\ and\ [req_1 \vee req_2 \vee req_3]\ Y(c + 1))$

and read as follows:

"Starting from the initial state, there exists a path leading to a UI response before the user has interacted three times with the UI."

The interest of this property is that, for instance, when user's requests require a large processing time on the system (e.g. in a website), it is guaranteed that at most after three interactions the UI is able to give some feedback to the user. Under the chosen framework for our approach [1], this is an example of *robustness* property, more precisely, a *response time* property.

The support to data-handling mechanisms on temporal logic formula is another advantage of MCL language (i.e. the declaration and initialization of the variable *"c"* in the formula above). This is possible to be expressed on classical modal μ-calculus, but it requires bigger (thus more difficult to read) formulas, and it is not possible to express in ACTL [10].

The set of tools is important to support formal analysis. Rather than developing our own tool to perform formal verification, we work in collaboration with the authors of CADP toolbox. Their know-how in formal methods and the maturity of their tools increased the confidence in our approach. In particular, CADP has continuously evolved in the past years. The last published work covering a similar technique [15] uses an earlier version of the CADP toolbox that dates from 2001, while we used the latest version of CADP (2014-c). In [7] the authors list the capabilities of the toolbox added in the last ten years.

By taking advantage of the new capabilities added to CADP, it is now possible for example to perform *compositional verification* on individual processes of the model, enabling to handle much larger state spaces. As explained in the Subsection *LNT Formal Model*, CADP creates a LTS from the formal model, and the reasoning is performed over this LTS. The more complex the system under evaluation is, the larger its LTS will be. *Compositional verification* is a way to avoid state space explosion, by creating an equivalent LTS for each process of the model [7], replacing a state space by an equivalent but smaller one. In practice, bigger models can be handled, so that we can consider more realistic UI models.

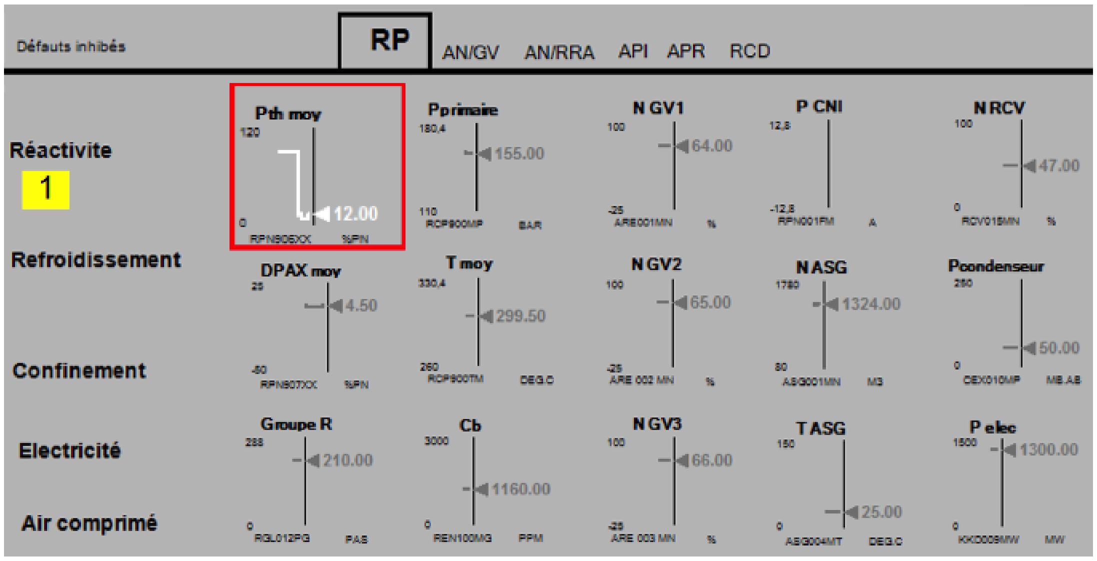
CASE STUDY

Our approach has been applied on a system's prototype of a nuclear power plant control room. The main UI of the prototype is illustrated in Figure 2a (in French).

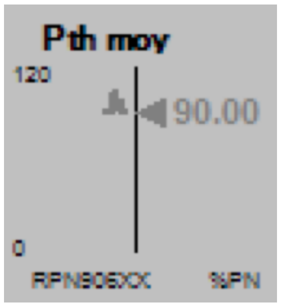
The main goal of the system is to provide a general overview of the plant status, advertising the control room operator in case of some discrepancies on the reactor [3].

On top of the user interface there are six tabs, namely RP, ANGV, ANRRA, API, APR and RCD. These tabs indicate the current status of the plant, which ranges from completely stopped to working on full capacity. For the purpose of this case study, we do not take into account neither who changes these states, nor how they are changed.

Depending on the plant status, different reactor's parameters are displayed in the middle of the UI. Each parameter is represented by the widget illustrated in Figure 2b. The top of the widget displays the name of the parameter (for instance, *Pth_Moy*, standing for *average thermal power*). The middle displays the current value of this parameter (90.00) and a line that shows the last values. This value varies between a min-



(a) Main view of the system



(b) One parameter zoomed out

Figure 2: UI prototype of a nuclear power plant overview

imum and a maximum range (0 – 120). Finally, the bottom of the widget displays the name of the sensor that monitors the parameter (*RPN906XX*) and also its measurement unit (*%PN*).

The system monitors the evolution of these reactor's parameters. If discrepancies occur in these values (for instance, if one achieves a value which is higher than the maximum expected value), then the parameter is highlighted in different ways, e.g. a colored frame around it (Figure 2a). Besides, the system generates an alarm signal in the reactor's function that is affected by this discrepancy, e.g. the box under the function *reactivity* (réactivite) in Figure 2a.

This system prototype has several other functionalities. However, we will not detail them here.

Formal model

Since LNT proposes a modular-based programming style [16] (inherited from LOTOS), the UI is described as several modules (Figure 3). Modularity is key for scalability. Which provides a means for structuring, abstraction, and reusability [16].

The modules are related to Presentation, Abstraction and Control, as defined in the PAC architecture style [4] (Figure 4). The *presentation* is in charge of the perceivable inputs and outputs for the user. The *abstraction* encompasses the functional core. The *control* ensures consistency between the abstraction and presentation [4].

LNT modules

The modules are identified as follows (Figure 3): *Plan state* describes the area in the UI where the plant state can be chosen, i.e.the 6 available states, and the current one. The module *menu* models the left part of the UI, where the operator has access to detailed views of the reactor according to the concerned function (e.g. reactivity, core cooling, confinement, etc.). This menu provides a hierarchical access to the UIs. By accessing a given function, for instance the menu option "reactivity", the user has access to other UIs that synthesize informations about this function [3], e.g. "boron concentration", "rods position", "boration/dilution" and "reactor control". The middle of the user interface is modelled by *reactor* and *generate signals* modules. The former has functions to evolve the reactor's parameter values in time, while the later generates discrepancies in these values, in order to simulate disturbances on the reactor. All those modules communicate with a central module called *selection*, that mediates the interactions on the UI and the calculations in the system's core.

PAC components

The separation of concerns is one of PAC's characteristic. Each one of the three components provides a way to address a different problem in the system under study. In our case, each LNT module of the UI is classified in the following way (Figure 4): *Plan state* and *menu* are in the *presentation* component, since they model the operator's input and the corresponding system's output in the UI. The *reactor* and *generate signals* are in the *abstraction* component, describing part of the functional core of the reactor. *Selection* is in the *control* component, providing the communication between *presentation* and *abstraction* components.

Beyond the PAC style, a special module called *user model* is included in the architecture, in order to describe part of the user's behavior. In this case study, the operator perceives the UI displayed by the system (arrow *interface* that exits the module *selection* in Figure 4) and reacts to it by interacting with the menu options (arrow *relevant menu options* that exits the module *user model* in Figure 4). This simulates a common monitoring activity of the operator [3]: the reaction to dis-

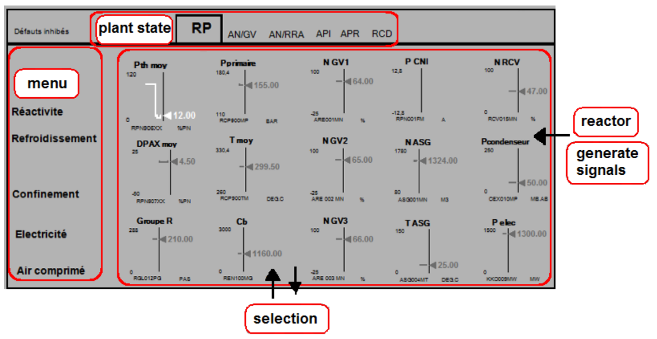


Figure 3: UI organized by modules

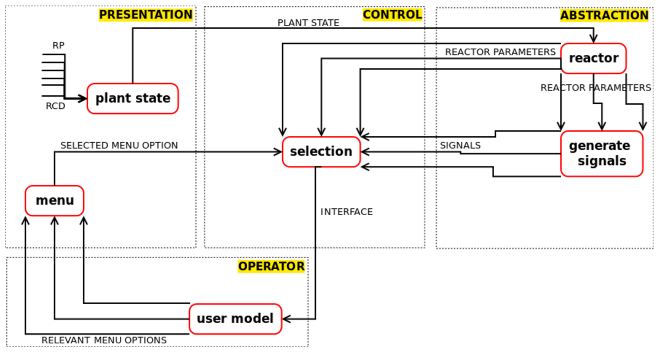


Figure 4: Formal model architecture

crepancies in the reactor. Consisting in accessing the views that have more details about the reactor failure.

The communication between the modules is done as follows: the *plant state* is set, and its value is passed to the module *reactor*. For its part, the reactor will evolve different parameters according to the plant state and to the scenarios that we have implemented. The parameters values are then passed to two modules: *generate signals* and *selection*. The former simulates failures in the parameters and the later identifies which UI should be displayed in order to monitor these failures. The *user model* module represents a human user, who perceives changes in the UI (e.g. the display of a failure), and interacts with the *menu* module to access the UI that provides detailed information about the failure. *Menu*, in its turn, passes to the *selection* module the option chosen by the operator, that in its part displays the corresponding UI.

All these modules are part of the whole LNT specification of the user interface, which contains 15 modules in total, and 3,339 lines of code.

Ergonomic properties

The properties that need to be verified have to be written in a formal way too. In our case, considering the framework [1], all the identified properties are classified as *robustness* properties, with the subcategory *observability>reachability*, which refers to the possibility of navigating through the observable system states of the system [1].

Five properties were identified as key for the project. We wrote them in MCL language. For example, the property:

"from any view, one can always go directly to the main view (i.e. without passing through any other view)"

is expressed in MCL in the following way:

$[true^*]$

$\langle (not(view))^* . \,'GLOBAL\_SYNTHESIS' \rangle\, true$

and may be read as:

From every reachable state

$\langle$*there exists a sequence of steps...*

...not passing through any view...

...and leading to the GLOBAL_SYNTHESIS view $\rangle$

This property ensures that, in all user interfaces, there is always the possibility to come back to the main view (called *global synthesis*, Figure 2a) with one single user interaction, i.e. without the need to access intermediate views before.

The other four properties are:

- a view is only accessible along the hierarchy of views [1]
- one can always come back to the parent view
- the SIGNAL DETAILS view is always directly accessible
- from any state one can always reach any view

Verification

We use CADP toolbox to perform formal verification. More precisely, we use `OCIS` (*Open/Caesar Interactive Simulator*, for step-by-step simulation with backtracking. We simulate scenarios over the formal model, and we test it interactively while an execution tree is created in the `OCIS` tool. This simulation allows one to explore all the possible executions of the model.

Another tool available in CADP is the `EVALUATOR 4.0` model checker (for handling MCL formulas [7]). We used it to evaluate the formula over the LTS of the formal model. In the end, we had a diagnosis of the evaluation: an example of steps that lead to a state where the property is true, or a counter-example otherwise (meaning a sequence of steps that leads to a state where the property is false).

The five properties defined before are evaluated to **true** over the model in question. In the case a property is evaluated as false, one can reconsider the essential questions in formal verification: is it a pertinent property? Is the property properly written in MCL language? Is the formal model a meaningful representation of the real system? Is the formal model properly written in LNT? If the answers for these questions are *yes*, then the formal verification rigorously indicates a problem in the system under verification, with a precise way to reproduce it.

[1]See in Subsection *LNT modules* the concept of hierarchy of views

CONCLUSION

The approach described in this paper aims at verifying the quality of user interfaces for a safety-critical system using model checking. Specifically, we verify the satisfiability of some ergonomic properties over formal models of the UIs. The UI models have been conceived in terms of PAC architecture, while the LNT formal language describes it. The MCL language is used to write the usability properties formally. The model checker used belongs to the CADP toolbox. Our approach is supported by recent versions of those tools, moving towards the powerfulness of formal verification of user interfaces.

The technique is being applied and validated in an industrial case study in the nuclear power plants domain. This corroborates the advantages of applying the strong capabilities of formal methods to ensure the quality of user interfaces in a real case study. It can be generalized for other safety-critical domains, though.

The following for the approach is to enrich the formal model, to cover visual aspects of the user interface. This would allow one to verify "static" properties, i.e. not necessarily requiring user interactions, for instance, color or position of widgets.

There are also other features of CADP that we aim to explore, for instance, the `BISIMULATOR` tool, which performs equivalence checking over models. This would allow us for instance to verify if different versions of the same user interface are equivalent or not. Such an approach would be applicable for adaptive user interfaces.

ACKNOWLEDGEMENTS

This work is funded by the French Connexion Cluster (Programme d'Investissements d'avenir / Fonds national pour la société numérique / Usages, services et contenus innovants).

REFERENCES

1. Abowd, G. D., Coutaz, J., and Nigay, L. Structuring the space of interactive system properties. In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, North-Holland Publishing Co. (Amsterdam, The Netherlands, The Netherlands, 1992), 113–129.

2. Bastien, J. C., and Scapin, D. L. Ergonomic criteria for the evaluation of human-computer interfaces. Tech. Rep. RT-0156, INRIA, June 1993.

3. Chériaux, F., Galara, D., and Viel, M. Interfaces for nuclear power plant overview. In *8th International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies 2012 (NPIC & HMIT 2012): Enabling the Future of Nuclear Energy*, NPIC & HMIT 2012, Curran Associates, Inc. (2012), 1002–1012.

4. Coutaz, J. Pac, an object oriented model for dialog design. In *Proceedings Interact*, vol. 87 (1987), 431–436.

5. Degani, A., Heymann, M., Meyer, G., and Shafto, M. Some formal aspects of human-automation interaction. *NASA Technical Memorandum 209600* (2000).

6. Galitz, W. O. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons, 2007.

7. Garavel, H., Lang, F., Mateescu, R., and Serwe, W. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *International Journal on Software Tools for Technology Transfer 15*, 2 (2013), 89–107.

8. Markopoulos, P., Johnson, P., and Rowson, J. Formal architectural abstractions for interactive software. *Int. J. Hum.-Comput. Stud. 49*, 5 (Nov. 1998), 675–715.

9. Masci, P., Ayoub, A., Curzon, P., Harrison, M. D., Lee, I., and Thimbleby, H. Verification of interactive software for medical devices: Pca infusion pumps and fda regulation as an example. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, ACM (New York, NY, USA, 2013), 81–90.

10. Mateescu, R., and Thivolle, D. A Model Checking Language for Concurrent Value-Passing Systems. In *FM 2008*, J. Cuellar and T. Maibaum, Eds., vol. 5014 of *Lecture Notes in Computer Science*, Springer Verlag (Turku, Finlande, 2008), 148–164.

11. Miller, S. P., Whalen, M. W., and Cofer, D. D. Software model checking takes off. *Commun. ACM 53*, 2 (Feb. 2010), 58–64.

12. Navarre, D., Palanque, P. A., Ladry, J.-F., and Barboni, E. Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact. 16*, 4 (2009).

13. Nielsen, J., and Molich, R. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (1990), 249–256.

14. Paternó, F. Formal reasoning about dialogue properties with automatic support. *Interacting with Computers 9*, 2 (1997), 173–196.

15. Paternò, F., and Santoro, C. Support for reasoning about interactive systems through human-computer interaction designers' representations. *Comput. J. 46*, 4 (2003), 340–357.

16. Sighireanu, M., Chaudet, C., Garavel, H., Herbert, M., Mateescu, R., and Vivien, B. Lotos nt user manual. *INRIA, june* (2004).

17. Sousa, M., Campos, J., Alves, M., and Harrison, M. Formal verification of safety-critical user interfaces: a space system case study. In *Formal Verification and Modeling in Human Machine Systems: Papers from the AAAI Spring Symposium*, AAAI Press, AAAI Press (Stanford, 26 March 2014), 62–67.

18. Vanderdonckt, J. *Guide ergonomique des interfaces homme-machine*. No. 13 in Collection "Travaux de l'Institut d'Informatique". Presses Universitaires, Namur, 1994.