Plasticity of User Interfaces: Formal Verification of Consistency

Raquel Oliveira **Sophie Dupuy-Chessa** **Gaëlle Calvary**

Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
Emails: FirstName.LastName@imag.fr

ABSTRACT

Plastic user interfaces have the capacity of adapting themselves to their context of use while preserving usability. This property gives rise to several versions of the same UI. This paper addresses the problem of verifying UI adaptation by means of formal methods. It proposes three approaches, all of them supported by the CADP toolbox and LNT formal language. The first approach permits the reasoning over the adaptation output, i.e. the UI versions: some properties are verified over the UI models thanks to model checking. The second solution proposes to verify the plasticity engine. The last approach compares UI versions thanks to equivalence checking. These approaches are discussed and compared on an example of a system in the nuclear power plant domain.

Author Keywords

Plastic user interface; formal verification; plasticity engine, property verification, critical system

ACM Classification Keywords

D.2.4. Software/Program Verification: Formal method; H.5.2. User Interfaces: Evaluation/methodology; I.6.4. Model Validation and Analysis

INTRODUCTION

The advent of ubiquitous computing and the increasing diversity of platforms and devices change user expectations. According to Calvary et al. [6], there is a need for systems to be able to adapt themselves to their context of use while preserving their usability. This property, named plasticity, provides users with different versions of a user interface (UI). The problem addressed in this paper is to verify to which extent these versions are consistent.

Given the large number of possible versions of a UI, it is time-consuming and error prone to check consistency by hand. Some automation must be provided to verify plasticity. This paper proposes the usage of formal methods to perform such verification. It describes and compares three approaches, using different verification techniques. Two of them consist in verifying properties on the formal models, either by directly checking the UI versions or by checking the plasticity engine used to generate the UI versions. The third approach is based on the comparison of UI versions.

The reminder of this paper starts by presenting the state of the art about verification of different UI versions, followed by a description and comparison of the three verification approaches. Finally, a conclusion summarizes the approaches and proposes perspectives.

Paste the appropriate copyright statement here. ACM now supports three different copyright statements:
• ACM copyright: ACM holds the copyright on the work. This is the historical approach.
• License: The author(s) retain copyright, but ACM receives an exclusive publication license.
• Open Access: The author(s) wish to pay for the work to be open access. The additional fee must be paid to ACM.
This text field is large enough to hold the appropriate release statement assuming it is single spaced.

RELATED WORK

In [14], the authors propose the automatic generation of consistent user interfaces when the platform changes. They do not consider, however, changes in the user nor in the environment. Besides, since their approach aims at generating UIs, it can not be applied to verify consistency of existing UIs.

Several approaches to reasoning over UIs propose to verify the satisfiability of properties over UI models, using model checking and/or theorem proving [11, 17, 18, 13, 15]. While such approaches are powerful thanks to their reasoning tools, none of them considers several versions of a UI or plasticity.

In [5] a formal approach to verify if a UI is a refinement of another UI is proposed, by verifying *functional* equivalence, for instance. This approach verifies if a UI provides at least as much functionalities as another UI. However, they do not verify different levels of equivalence and they do not cover UI appearance.

Other approaches addressing different versions of a UI are mainly based on regression testing. Some authors [1, 10, 9] use the *Capture-and-Replay* technique. This technique allows one to execute again (*replay*) test cases that had their execution recorded before (*capture*). However, the scripts generated in the *capture* part are vulnerable to GUI layout change, which can render entire automated test suites inept [4]. In [4] an approach based on image recognition for comparing UIs using the *Visual GUI Testing* technique is proposed. This technique is less based on hard coded tests, but it is very sensitive to the GUI elements positioning. Alternatively, GUIDIFF tool [3] performs regression testing of different versions of a UI, providing a list of detected deviations. The drawback of testing approaches is that they are usually performed late in the system development cycle. Besides, none of them consider UI versions generated thanks to plasticity, whereas plasticity takes into account specific adaptation means [19], such as the *remolding* of one interactor.

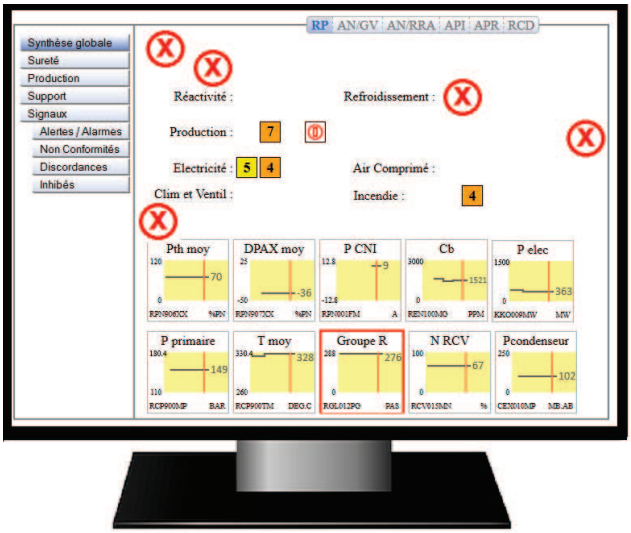

Figure 1: UI in Expert Mode on a large screen computer

Figure 2: UI in Training Mode on a smartphone

APPROACHES FOR PLASTICITY VERIFICATION

We present three approaches to verifying UIs, illustrated on a example of a system in the nuclear power plant domain.

A Non-Trivial Example

This example consists in a system prototype of a nuclear power plant. The system provides an overview of the plant state and notifies the control room operator about all unexpected events in the plant. The main UI contains four zones:

- The top zone displays six tabs for selecting the plant status;
- The *Default Signal* ("Signaux de défaut") zone synthesizes signals triggered in reactor functions, according to unexpected events occurred in the reactor parameters;
- At the bottom, the *Parameter* ("Paramètres") zone displays various reactor parameters (e.g. the pressure), each one represented by a widget containing: the parameter name, its current value, a curve displaying the value evolution over time, the minimum/maximum value bounds, the sensor that monitors the parameter and its measurement unit;
- On the left, users can access other screens by a menu.

This paper uses only two versions of the UI:

- An expert version on a large screen computer (Fig. 1, in French), which displays the four zones of the UI, but some guidances (here labels) are removed, providing expert users with less loaded UIs. Label omission are highlighted with crosses in Fig. 1. For instance, the title of the Default Signal and Parameter UI zones are omitted;
- A training version on a smartphone (Fig. 2) where all labels are displayed, but only reactor signals and parameters currently affected by a failure are shown. The parameter widgets are re-molded to fit on the size-reduced screen (i.e. a text instead of a curved line). Moreover, the menu is accessible via a blue button at the top-left of the UI.

The system provides a meta UI to choose the mode (expert or training) and some element distribution over devices. For instance, if two screens are available for displaying the UI, it is possible to display parameters on one screen and signals on the other one (UI *redistribution* [19]).

Considering these two UI versions, we can either check if they are somehow equivalent in terms of appearance and interaction capabilities, or verify common properties in both of them. For instance, all versions must always display the reactor parameters whatever their form are. Such verifications can be performed thanks to different formal approaches we describe now.

Common Approach

The common part of the three proposed approach consists in specifying formal models for the UI versions, in order to perform formal verification (Fig. 3). The formal model covers not only the user interfaces, but also some aspects of the system core. Therefore, the entry point of the approach is the interactive system that needs to be verified. The formal specification is written in LNT formal language [7]. LNT was chosen for its expressiveness and its tool support, CADP [8].

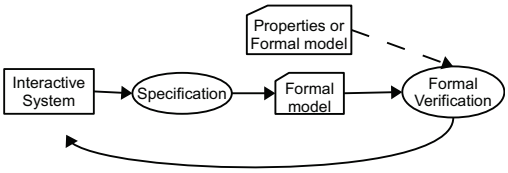

Figure 3: Global approach

CADP generates a Labeled Transition System (LTS) from LNT models, in which the UI behavior and appearance, user interactions, UI context of use, etc., are expressed as transition labels. This LTS representation allows the usage of several formal techniques. For instance, *model checking* can be used to verify that all UI versions display reactor signals,

and *equivalence checking* can compare the smartphone and the large screen versions to check consistence in the display of signals (either both of them display it or none of them do).

In order to obtain more reusable results, interactive system models are represented following the ARCH architecture [2]. In ARCH, systems are decomposed in three main parts: the user interfaces, the functional core and a dialog controller which ensures the consistency between the functional core and the UIs. We took into account this separation of concerns when we wrote the formal models the interactive system. In Fig. 4, each box represents one or more LNT modules. In our example, the UI LNT modules contain the four zones of the UI, i.e., the plant status, the default signals, the parameters and the menu. In addition, we created a user LNT module, which represents the expected user behavior defined by nuclear power plant procedures.

Figure 4: Arch architecture usage in formal verification

This global structure is used in the three approaches for plasticity verification, presented in the following.

Three Proposed Approaches

Verification of UI Versions

Considering the ARCH architecture, the first approach contains modules for the functional core, the dialog controller and the UI. Plasticity can generate several versions of a UI. In our example of nuclear power plant control system, there are two UI versions: one for the expert mode on a large screen device and one for training mode on a smartphone. They are both described in the UI modules. The goal of this approach is to directly verify the different UI versions.

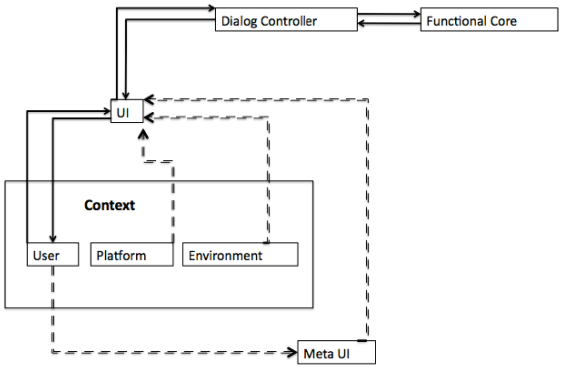

Figure 5: Modeling UI versions

In addition, the formal model describes the context of use, in terms of platform, user and environment. The concept of context is not explicitly described in the formal model. However, platform, user and environment are formally described by one or more LNT modules. In our example, the environment is not taken into account. The platform can be a large screen device or a smartphone. For users, the module describes the expected user behavior when interacting with the UI (arrows for and to the UI) and the user profile. Changes in the context of use are communicated to the UI, which will adapt accordingly.

The user can also interact with a meta UI, which allows to configure UIs and to choose the UI mode (i.e. expert or training). A communication between the meta UI and UI modules permits the UI versions to modify according to the user's selection in this meta UI, allowing the UI to adapt according to the mode, for instance.

The UI modules receive all the information concerning the user (i.e. behavior and profile), the platform, the choices in terms of interaction through the meta UI and possibly the environment. From these information, the UI modules can choose the appropriate representation of the UI. This means that the UI modules contain the UI adaptation logic. The UI adaptation rules in our example represent all the adaptation effects in the corresponding UI zones. For instance, in the UI parameter (resp. signal) zone, there are two cases: the displaying of only the problematic parameter (resp. signal) on a smartphone and the displaying of all parameters (resp. signals) on large screens (Fig.6).

```
PARAMETERS.LNT

case platform in
        Smartphone ->
                display_failure_param();
        PC ->
                display_all_params();
end case;

SIGNALS.LNT

case platform in
        Smartphone ->
                display_failure_signal();
        PC ->
                display_all_signals();
end case;
```

Figure 6: Pseudo code of UI LNT modules

The UI modules send the generated UI version to the user. The LTS generated from such formal model contains all cases of the adaptation rules, meaning that all UI versions are represented in the LTS.

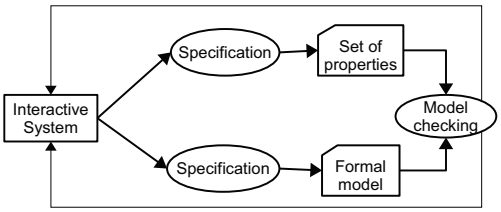

Figure 7: Verification of properties by model checking

Once the formal model is created, model checking formal technique can be used to verify properties over the model Fig. 7. Due to the exhaustive reasoning provided by model checking, the verification of properties cover all the UIs that are generated by the adaptation.

For instance, we can verify that all UI versions display reactor signals, which is expressed by the following property: "*From every reachable state, there exists a sequence of steps leading to the display of signals*". Using Model Checking Language (MCL [12]) to formalize this property, we have this temporal formula:

$$[true^*]\langle true^* \,.\, 'DISPLAY_.*_SIGNAL.*'\rangle true$$

In this MCL formula, the first ".*" is a regular expression that will match in the LTS transitions labeled either with "DISPLAY_FAILURE_SIGNAL" or with "DISPLAY_ALL_SIGNAL".

In this first approach, the UI versions are modeled in the UI modules, which contains the description, the behavior of the UIs and the adaptation logic. The adaptation logic is embedded into the description of the UI and is specific to the modeled example. However, model checking permits to perform verification of properties over all the UI versions.

Verification of the Plasticity Engine

In the second approach, we consider that a plasticity engine is formally described in the model (Fig. 8). The goal is to explicitly verify the adaption logic of the engine.

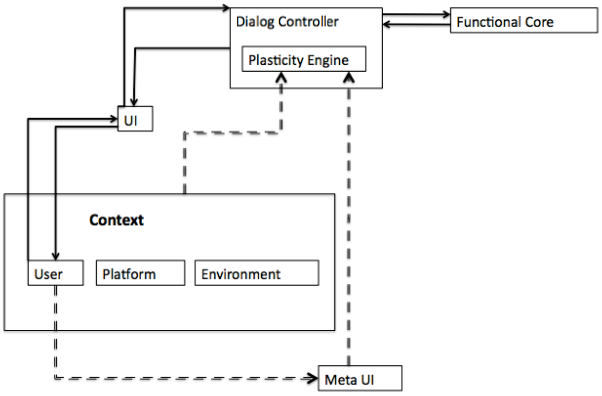

Figure 8: Modeling a plasticity engine

Following the common approach, the formal model is consistent with the ARCH architecture. In this case, the dialog controller also contains the description of the adaptation engine. The engine receives information directly from the *context* modules and from the meta UI. From this information, it can calculate the most appropriate UI version.

The plasticity engine implements all the transformation rules for adaptation (e.g. pseudo code in Fig.9). In this approach, the UI, the context of use, the meta UI and the functional core are specified to give the appropriate information to the plasticity engine (i.e. which UI is currently displayed, the context of use, the UI mode and which information should be displayed). The idea is to verify that the plasticity engine correctly answers to these information: it must perform all transformations in the UI, leaving the UI modules in charge of displaying such UI and managing user interactions.

PLASTICITY_ENGINE.LNT

```
case platform in
        Smartphone ->
                display_failure_param();
                display_failure_signal();
        PC ->
                display_all_params();
                display_all_signals();
end case;
```

Figure 9: Pseudo code of adaptation rules

In order to verify the plasticity engine, attention should be paid to the transformation rules. One kind of verification that can be done is to verify to which extent the transformations have an effect on the behavior of the UI versions. This can be verified by checking if the same interaction capabilities are present in all UI versions generated by the transformations. Such verification attests the quality of the transformation rules by reasoning over the output of the transformations: the generated UI versions.

For instance, in the example described in this paper, the large-screen UI (Fig. 1) displays all reactor signals in the *Default Signal* zone (i.e. failure and non-failure signals), while the smartphone UI displays in the same zone only the failure signals (Fig. 2). In any case, once a failure signal occurs, the UI is always expected to display it. We can verify that the UI transformation preserves such requirement by the following property: "*From every reachable state, once a failure signal occurs, there exists a sequence of steps leading to the display of this signal*". In MCL temporal logics, this is expressed as:

$$[true^* \,.\, 'FAILURE_SIGNAL_\backslash(.*\backslash)']$$
$$\langle true^* \,.\, 'DISPLAY_FAILURE_SIGNAL_\backslash 1'\rangle true$$

where $\backslash(RE\backslash)$ is a regular expression that matches whatever the unadorned RE matches (here, RE is any character, represented by ".*"), and the expression $\backslash n$ matches the same string of characters that was matched by an expression enclosed by "\(" and "\)" earlier. Here, n is a digit representing the n-th occurrence of "\(" counting from the left. Intuitively, the first regular expression will match a failure in a given reactor signal (i.e. "FAILURE_SIGNAL_*signalname*") and the reactor signal name will be matched in the second regular expression (i.e. "DISPLAY_FAILURE_SIGNAL_*signalname*"), expressing the requirement that once a failure signal occurs, such signal will always be displayed in the UI.

Going further, we propose to verify the plasticity engine itself by verifying, for instance, the coverage of the transformation rules. The transformations rules are expected to cope with changes in the context of use and correspondingly adapt the UI version. In this case, we could verify whether the engine takes into account all possible changes in the context of use or not.

For instance, in the nuclear power plant system, two changes in the context of use are described: the platform (i.e. PC or smartphone) and the user (i.e. expert or training). If we consider the platform, we can verify if the plasticity engine has transformation rules to cover all changes in the platform of the considered scope (i.e. PC and smartphone), which is expressed by the property: *"Starting from the initial state, there exists a sequence of steps leading to the display of the smartphone version of the user interface"*. The same property can be written to verify the display of the PC version of the UI. In MCL, this is expressed by the following formula:

$$\langle true^* \,.\, 'DISPLAY_SMARTPHONE_UI.*' \rangle true$$

This second approach enables to explicitly specify and to verify the plasticity engine. Once the engine is formally verified, its specification can be used, for example, to suggest appropriate transformation rules to automatically generate the code of the engine.

Formal Model Comparison

In this last approach, there is no explicit representation of plasticity in the formal model. We simply make a comparison over two UIs, with no considerations about how they have been obtained. Before creating the formal models, designers must have the rendering of the UI versions adapted to each context of use, allowing the specification of one formal model for each UI version. For instance, in the example of the nuclear power plant control room, the combination of the two platforms and the two expertise modes can give rise to four formal models. Then, the formal models are compared to each other. This approach requires as many formal models as the number of contexts of use to cope with.

In this approach, the formal models also follow the ARCH architecture. However, no module for the context of use nor for the meta UI is included. All the specificities brought from the context of use is included in the corresponding UI version formal model. The UI formal models reflect their context of use by describing the UI generated for such context.

These formal models are then compared, two by two, using the equivalence checking formal method (Fig. 10). In [16], we describe how we can measure to which extent the user interfaces are the same with respect to their interaction capabilities and appearance. In case they are not equivalent, the UI divergences are highlighted, and the possibility of leaving these divergences out of the analysis is provided, reinteracting the equivalence verification. Furthermore, the approach can demonstrate that one UI contains at least all interaction capabilities of another (inclusion).

For instance, in the nuclear power plant example, the two UI versions are considered as equivalent: in a first step, the equivalence checker considers that their appearance diverges because of the missing labels, the remolding of the parameter widget and the accessibility of the menu; then some abstractions are made [16] to reason about the interaction capabilities and not their appearance. After these abstractions, the two UI versions are shown equivalent.

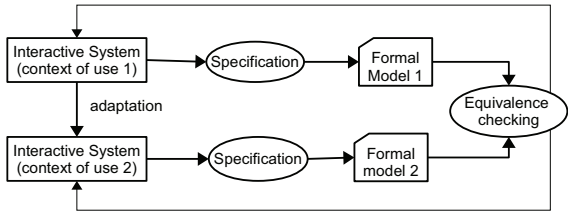

Figure 10: Equivalence checking of plastic user interfaces

The comparison approach enables the verification on UI versions without having to specify how they were obtained. It requires to completely describe each UI version. While considering the multiplicity of context of use, it is quite difficult to imagine all possible UIs. However, the comparison can be performed for some identified and well-defined contexts of use. This is relevant for safety critical systems, allowing one to verify that critical features are present in all UI versions.

Comparison of the Proposed Approaches

Each approach has its strengths and drawbacks. To compare them, we focused on the following criteria: the number and complexity of the UI versions to be modeled, the number and complexity of adaptation rules, and which kind of verification can be performed.

In the first approach, which represents all the UI versions and the adaptation logic inside the UI modules, all the complexity is embedded into these modules. If the UI complexity is below a certain threshold and the number of adaptation rules for the UI is low, it can be a good and simple solution. Once adaptation representation becomes too onerous in the formal model, prejudicing its readability, it is recommended to separate it into another module. In terms of verification to be performed, this approach allows the verification of the same properties on all UI versions. In addition, it can also be used to check different properties over UI versions. For instance, in the smartphone version, the menu options must always be *accessible* while in the large screen version, they must always be *visible*. However, UI remolding [19] cannot be verified in this approach: only properties related to interactions can be checked using model checking.

The novelty of the second approach is the representation and verification of a plasticity engine. One can imagine to develop an engine from a proven specification. However, according to the number and the combination of rules, the description of the engine can become too complex. The limit here is not related to the UI modules but to the formalization of numerous transformation rules and their combination. Moreover, as in the previous approach, model checking restricts verification only to interaction capabilities (i.e. remolding cannot be verified).

Comparison of formal models bypasses the difficulties of modeling the adaptation logic, and it ensures remolding verification. It is based on the existing versions of the UI, and there is no need to represent the plasticity engine in the formal model. The formal models focus on representing the

UI versions, without knowing how they have been produced. Thanks to equivalence checking, it is possible to compare appearance and interaction. However, it requires to create one model for each UI version and to compare UI versions two by two. It can be time-consuming if the number of UI versions is significant. Finally, with this approach, designers do not verify some properties, but rather than, they check the consistency between two UI versions.

CONCLUSION

This paper presents three possible approaches to verify plasticity. Each of them permits to reason about different aspects: the first one allow designers to verify some properties common to all the UI versions; the second one aims at verifying the plasticity engine; the third one is based on UI comparison regardless the way they have been produced. For the moment, we have mainly explored the third solution [16]. We have shown its usefulness, in particular in a critical domain such as the nuclear power plant one. Deeper investigation must be performed for the two first approaches to broadly identify their limits, in particular in terms of the description of the adaptation engine logic. More globally, these approaches must be analyzed on other case studies to observe its scalability and to identify other potentials and drawbacks.

ACKNOWLEDGMENTS

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. Al-Zain, S., Eleyan, D., and Garfield, J. Automated User Interface Testing for Web Applications and TestComplete. In *CUBE*, ACM (2012), 350–354.
2. Bass, L., Little, R., Pellegrino, R., Reed, S., Seacord, R., Sheppard, S., and Szezur, M. R. The ARCH model: Seeheim Revisited. In *User Interface Developpers' Workshop* (1991).
3. Bauersfeld, S. GUIdiff - A Regression Testing Tool for Graphical User Interfaces. In *ICST*, IEEE (2013), 499–500.
4. Borjesson, E., and Feldt, R. Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry. In *ICST*, IEEE Computer Society (2012), 350–359.
5. Bowen, J., and Reeves, S. Refinement for User Interface Designs. *Electronic Notes in Theoretical Computer Science 208* (2008), 5–22.
6. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers 15*, 3 (2003), 289–308.
7. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., and Smeding, G. Reference Manual of the LNT to LOTOS Translator (Version 6.1). INRIA/VASY and INRIA/CONVECS, 131 pages, Aug. 2014.
8. 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.
9. Jung, H., Lee, S., and Baik, D.-K. An Image Comparing-based GUI Software Testing Automation System. In *Software Engineering Research and Practice* (Las Vegas, NV, USA, May 2012), 318–322.
10. Lu, L., and Huang, Y. Automated GUI Test Case Generation. In *CSSS*, IEEE Computer Society (2012), 582–585.
11. 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 EICS'13*, ACM (2013), 81–90.
12. 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.
13. 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).
14. Nichols, J., Myers, B. A., and Rothrock, B. UNIFORM: Automatically Generating Consistent Remote Control User Interfaces. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, ACM Press (New York, NY, USA, 2006), 611–620.
15. Oliveira, R., Dupuy-Chessa, S., and Calvary, G. Formal Verification of UI Using the Power of a Recent Tool Suite. In *EICS* (2014), 235–240.
16. Oliveira, R., Dupuy-Chessa, S., and Calvary, G. Equivalence Checking for Comparing User Interfaces. In *EICS*, ACM (2015).
17. Paternó, F. Formal Reasoning about Dialogue Properties with Automatic Support. *Interacting with Computers 9*, 2 (1997), 173–196.
18. 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: AAAI Spring Symposium*, AAAI Press (2014), 62–67.
19. Vanderdonckt, J., Calvary, G., Coutaz, J., and Stanciulescu, A. *Multimodality for Plastic User Interfaces: Models, Methods, and Principles*. Springer, 2008, chapitres d'ouvrages 4, 61–84. D. Tzovaras (ed.), Lecture Notes in Electrical Engineering, Springer-Verlag, Berlin, 2007.