Using Formal Models to Cross Check an Implementation

Raquel Oliveira, Sophie Dupuy-Chessa, Gaëlle Calvary
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
FirstName.LastName@imag.fr

Daniele Dadolle
Atos Worldgrid, Grenoble, France
daniele.lanneau@atos.net

ABSTRACT
Interactive systems are developed according to requirements, which may be, for instance, documentation, prototypes, diagrams, etc. The informal nature of system requirements may be a source of problems: it may be the case that a system does not implement the requirements as expected, thus, a way to validate whether an implementation follows the requirements is needed. We propose a novel approach to validating a system using formal models of the system. In this approach, a set of traces generated from the execution of the real interactive system is searched over the state space of the formal model. The scalability of the approach is demonstrated by an application to an industrial system in the nuclear plant domain. The combination of trace analysis and formal methods provides feedback that can bring improvements to both the real interactive system and the formal model.

ACM Classification Keywords
D.2.1. Software Engineering: Requirements/Specifications; D.2.4. Software Engineering: Software/Program Verification

Author Keywords
formal methods, traces, interactive systems, requirements

INTRODUCTION

An interactive system is expected to implement a set of requirements, which are usually *informal*. According to [13], *informal* models are genuinely ambiguous and heavily rely on human intuition. They are expressed using natural language or loose diagrams, charts, tables, etc. Requirements from which interactive systems are implemented are usually informal, for instance, documentation, prototypes, information gathered through meetings, exchange of e-mails, etc. Such a lack of formality in the requirements may be a source of problems. A question that arises is whether a given interactive system correctly implements its requirements or not.

We propose an approach to *validating* the implementation with respect to its requirements. *Validation* can be defined as *"the process of providing evidence that the software and its associated products satisfy system requirements allocated to software at the end of each life cycle activity, solve the right problem, and satisfy intended use and user needs"* [1]. The requirements of the system can be seen as a representation of the user needs. In this paper, requirements are validated by using formal methods in combination with system logs.

Formal models are system descriptions expressed in languages which, unlike natural human languages, do not allow for any double meanings. Such languages have a precisely defined syntax and a formal semantics, such as algebraic data types, input/output automata, etc [13]. Once one has framed a system in a formal model, it can be deduced precisely what are the consequences of the assumptions made. In this paper, formal models are used as a support to cross check an implementation with respect to the initial requirements.

The reminder of the paper starts by presenting the related work, followed by a description of our approach to validating systems using formal models. Analysis of traces is then applied to an industrial case study, in which a set of traces extracted from a system is analyzed over the system formal model. Finally, we discuss the advantages and drawbacks of the approach, concluding with current results and perspectives.

RELATED WORK

The validation of interactive systems with respect to their requirements can be ensured in different ways, such as through property verification [21, 27, 19], testing [28, 11, 17], analysis of traces [30, 4], or through comparison of models [5]. One possible outcome of such validation is the improvement of the requirements, which is the goal of *requirement engineering* [15, 2, 25, 16, 26, 3, 5].

The initial requirements of a system can be expressed as properties, and model checking can be used to verify the satisfiability of such properties over a model of the system [21, 27, 19]. However, the real interactive system is not analyzed, since the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org. EICS'16, June 21-24, 2016, Brussels, Belgium©2016 ACM. ISBN 978-1-4503-4322-0/16/06... $15.00 DOI: http://dx.doi.org/10.1145/2933242.2933257

properties are extracted from the system requirements, instead of from trace executions of the system.

Automatic test case generation have been widely discussed in the literature, and test cases can be used to validate a system with respect to its requirements. For instance, test cases can be extracted from models representing the GUI navigation, such as the `SNet` [28], in which navigation paths are extracted representing the possible test scenarios. However, only GUI navigation is covered by the approach. Alternatively, in [11] test cases are extracted from examples and counter examples resulted from the application of model checking over `Lustre` models of interactive systems. The inconvenience here is the expressiveness of `Lustre` models, in which interactive systems are described by Boolean data flows. In [17], the specification language of the PVS (*Prototype Verification System*) theorem prover is used. Such formal specification is derived from the code source of the system, and used to extract test cases. Although the use of the system code source is convenient, it becomes a limitation once such code is not available.

Analysis of traces was also used to verify an avionic system [30], in which some properties are verified over a translation of traces. However, the focus here is not *validation* with respect to the requirements. Alternatively, analysis of traces can be used to support unit and system level testing [4], in a "semi-formal" method. The combination with formal models would bring to the approach a larger state space to explore.

Other approaches propose to improve specifically the requirements of systems. The *requirement engineering process* consists of four key activities [15, 2]: requirements elicitation, analysis, specification, and validation. Several approaches exist to validate requirements. For instance, using *personas* to support requirements engineering [25], or specifying requirements at different hierarchical levels of abstraction [16] and using verification to ensure traceability between them [26], or finally using formal notations such as Petri nets to support validation [3]. However, the real interactive system is not the focus of these approaches, rather improvements in the requirements are the main goal. We aim at validating the interactive system with respect to the requirements, and improvements in the requirements is one possible outcome of the analysis.

An alternative validation approach was proposed in [5], in which formal specifications are used for validation purposes. As an outcome of this work, several ambiguities and omissions were discovered in the requirements, and several inadequacies and errors were discovered in the formal specification, which are some of the motivations of this paper (in addition to bringing improvements to the real system). In the authors' proposition, two formal specifications (one written in algebraic notation and another using a synchronized transition system) describing the system are compared to each other, to help to understand and debug the informal requirements. Although divergences may emerge from such a comparison, the approach does not focus on the real interactive system.

FORMAL MODELS TO VALIDATE AN IMPLEMENTATION

We aim at validating an interactive systems with respect to its requirements using formal support. An interactive system and its requirements are fundamentally different: while requirements express the user needs in an *informal* way, the actual interactive system *formalizes* solutions for the user needs. Given this fundamental difference, both the requirements and the implementation can not be directly compared to each other, and a gap between them will always exist.

Furthermore, a formal model describing the behavior of an interactive system also has a *formal* nature, and can not be directly compared to the informal system requirements either. However, given the formal nature of both the implementation and the formal model of the system, they can be compared to each other (Figure 1).

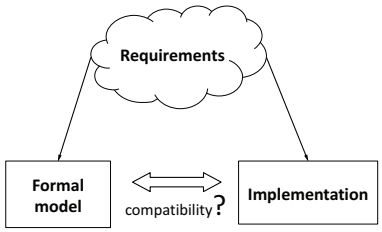

Figure 1: Formal models to validate an implementation

Although both an implementation and a formal model have the same formal nature, they are not at the same level of abstraction. Formal models describe some aspects of the implementation, and what will be included in the model depends entirely on the validation goals: only aspects of the system we want to validate are included in the formal specification. In our approach, we do not take into account mode changes or the relation between state attributes that are internal to visible attributes, for instance. In this paper, we propose to derive such a formal specification from the initial requirements.

We suggest using formal models to cross check the interactive system with respect to its requirements, by validating whether the formal model and the implementation are compatible. The requirements here can be any system requirement, including but not limited to UI requirements. This covers UI navigation, graphical aspects of UIs, UI interaction capabilities, and functional requirements as well. Such requirements can come from the documentation, prototypes, etc. The ultimate goal is to give clues about whether the real system implements the requirements as expected. If the system formal model and the real system are inter-operable, then it is *probable* that both the formal model and the implementation interpreted the requirements in the same way.

Since the formal model is a representation of the real system, as for all model-based approaches, there is no guarantee that the model is correct. It may contain faults, therefore, may not be a correct representation of the system. To mitigate this issue, the formal model can be assessed by a domain expert. Such an evaluation brings a certain confidence about the realism of the model. This is the standard problem of assessing whether a model is adequate.

Since our approach relies on models, it cannot fully determine whether the real system implements the requirements as expected. Yet, it can give directions. The key of the approach rationale is *redundancy*: when two groups of people develop in parallel different artifacts based on the same source, if both interpret the requirements in the same way, there is a high probability that they are both correct (not neglecting the possibility that both interpret the requirements equally wrongly, if the requirements are ambiguous). If the formal model and the implementation diverge, then at least one of both groups of people misunderstood the requirements. The fact that these two views are taken may result in ambiguities in the requirements being detected. Indeed, divergence and disagreement between models from different viewpoints can be exploited to actually enhance our understanding of design issues [12]. Arnold *et al.* [5] describe how an informal comparison of two formal models allowed them to detect ambiguities in a requirements document and to correct mistakes in each model.

We propose three different ways to use formal models to validate interactive systems. For instance, the formal model can be used as a basis to *derive test suites*, which can be executed on the interactive system, which provides the real inputs. A test execution engine is required to execute the test cases on the interactive system and to implement the test oracles that output the test verdicts. The ultimate goal is to generate test cases that are feasible, meaning that they can be executed over the interactive system. Once a feasible test is executed over the interactive system and does not pass, it can indicate a flaw either in the interactive system or in the formal model.

Alternatively, both the formal model and the interactive system can be executed in parallel in *co-simulation*. The outputs of one are connected to the inputs of the other and vice versa. In this case, a conversion in two directions is needed (i.e., from the interactive system to the formal model and vice versa), or the formalization of a language common to both the interactive system and the formal model. If the interactive system and the formal model can execute in parallel in co-simulation, it means that both are aligned in the way they implement the requirements.

Finally, formal models can be integrated to *analysis of traces* to validate interactive systems. In this case, log files generated by the implementation are interpreted and used to check whether the formal model can simulate the same sequence of traces or not (Figure 2). With this purpose, a translation of the log files into a format that can be treated by the formal model is required. Once the log files are transformed into this format, one can check if a given trace is included in the set of traces described in the formal model, meaning that the formal model simulates the scenario described in the log file in the same way as the interactive system.

In this paper, we detail how *analysis of traces* can be used to validate interactive systems, mainly because it does not require major changes in the development methodology of the analyzed interactive system, and most interactive systems already include logging mechanisms.

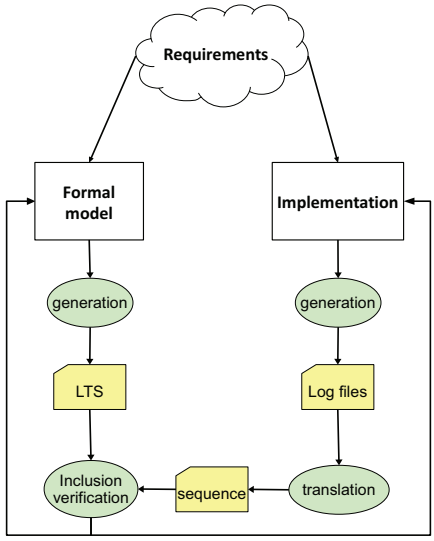

Figure 2: Analysis of traces

ANALYSIS OF TRACES IN DETAILS

This section presents the languages and tools used to implement our approach, how they are instantiated, and the parser that we developed to translate log files.

Languages and Tool Support

The choice of the toolbox was mainly motivated by its maturity, continuous evolution, support, and the numerous tools it includes [21]. CADP [14] is a toolbox for verifying asynchronous concurrent systems: systems whose components may operate at different speeds, without a global clock to synchronize them. Such components are described by *modules*, and they communicate and exchange information from time to time by channels. In our approach, we mainly used EVALUATOR[14], the model checker of CADP.

In model checking, a system is represented as a finite-state machine, which is subject to exhaustive analysis of its entire state space to determine whether a set of properties holds or not. The analysis feedback is mainly supported by the generation of *counter-examples* when a property is not satisfied. Counter-examples furnish a precise way to identify potential problems in the modeled system. The results of the analysis permit the modeled system to be improved.

One of the input languages of CADP is LNT [8]. The LNT specification language proposes a modular-based programming style, which suits well the modeling of interactive systems by composition. LNT has a syntax close to the imperative programming style (easier to learn and to read). We use LNT to write the formal models of the interactive system.

CADP can generate graph-based models called LTSs (*Labeled Transition Systems*) from the LNT model. An LTS is a graph composed of states and transitions between states. Transitions are triggered by actions, which are attached to the LTS transitions as labels. LTSs are suitable to describe systems

whose status change through actions of some kind. Intuitively, an LTS represents all possible evolutions of a system formal model.

We use MCL (*Model Checking Language*) [18] to formalize the expected properties of the interactive system. MCL is an enhancement of the modal μ-calculus, a fixed point-based logic that subsumes many other temporal logics, aiming at improving the expressiveness and conciseness of formulas. Specifically, MCL adds data-handling mechanisms, a fairness operator, and constructors inspired from functional programming (e.g., `let`, `if-else`, `case`, `while`, `repeat`, etc.) [18]. 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:

$$\begin{array}{r} \nu Y(c : nat := 0) \; . \\ \langle not(req_1 \vee req_2 \vee req_3)^* \; . \; resp \rangle \; true \\ or \\ ((c < 3) \; and \; [req_1 \vee req_2 \vee req_3] \; Y(c+1)) \end{array} \quad (1)$$

and read as follows: *"Starting from the initial state, there exists a path leading to a UI response (i.e., resp) before the user has interacted three times with the UI (i.e., req_1, req_2, and req_3)"*. The support to data-handling mechanisms is illustrated in this formula by the declaration and initialization of the variable `c`.

Instantiation using the Tools

The entry points of the approach are both the formal model and the real interactive system (Figure 3). On the left of the figure, the formal model of the interactive system is specified [21] in LNT, and it is then automatically transformed into an LTS using the CADP toolbox.

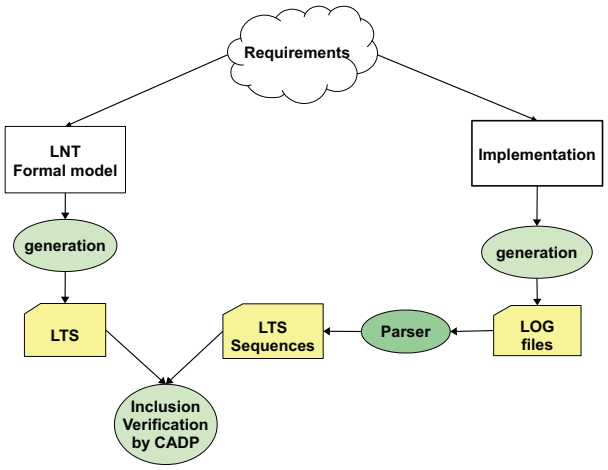

Figure 3: Analysis of Traces in details

The right part of Figure 3 consists of the execution of several scenarios in the real system, and subsequently generation of the log files.

The horizontal part of the figure, which links the formal model and the implementation, consists of a translation of the log files into a sequence that can be searched in the LTS of the formal model. We developed a Parser in Java to automate this translation. The output of this translation is a set of properties in MCL containing the scenarios included in the initial logs. The approach is fully tool supported. The Parser is in dark green in Figure 3 to indicate that we developed this tool. The other tools in light-green ellipses are either provided by the CADP toolbox or by the environment of the real system.

Finally, the EVALUATOR model checker of CADP is used to check whether the LTS sequences are included in the LTS of the formal model. The results of the analysis permit the formal model and/or implementation to be improved. However, the approach does not allow *refinement* (moving in meaning-preserving and property-preserving steps from abstract to more concrete models). Formal specification and refinement require a knowledge that not all developers have.

Parsing the Trace Files

The Parser takes as input a log file, extracts the lines containing the executed scenario, and translates them into a sequence of transition labels of the LTS. The Parser then encloses such sequence of labels into a temporal formula in MCL, concatenating the sequence of transition labels. The MCL formula has the following format:

$$\langle \; true^* \; . \; L_1 \; . \; L_2 \; . \; \ldots \; . \; L_n \; \rangle \; true^* \quad (2)$$

This formula expresses that, starting from the initial state of the LTS, there is an arbitrary path (matched by the regular expression "$true^*$" in the possibility modality $\langle\rangle$) leading to the sequence of transitions labeled with $L_1 \; . \; \ldots \; . \; L_n$, which is the sequence of transition labels initially generated. Intuitively, this MCL formula makes it possible to check whether the sequence of steps representing the scenario can be found in the LTS (the state space) of the formal model or not (Figure 4). If the trace is found, it means that the formal model simulates the scenario in the same way as the real system. Knowledge of MCL is necessary to apply our approach. Readers can find more details about this language in [18] and in the on-line manual of the language[1]. The inclusion checking of the MCL property is done using the EVALUATOR tool.

CASE STUDY

The proposed approach was developed considering an industrial power plant case study. Our research laboratory and Atos Worldgrid[2] investigated a prototype of a control room system developed by EDF[3] (*Électricité de France*). A formal model of the system was developed, aiming at performing formal verification of properties [21], and Atos Worldgrid implemented some user interfaces of the system on their industrial product called ADACS-N^{TM} (***A**dvanced **D**ata **A**cquisition and **C**ontrol **S**ystem for **N**uclear power*). In this context, a question that

[1] http://cadp.inria.fr/man/mcl.html
[2] http://fr.atos.net
[3] https://www.edf.fr/

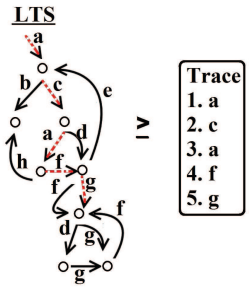

Figure 4: Example of trace inclusion checking

arises is whether ADACS-N^{TM} correctly implements the EDF prototype or not. The requirements here consist of informal requirements given by EDF during project meetings, exchange of e-mails with questions/answers, some documentation, print screens of the EDF prototype, and a video illustrating some functionalities of the EDF prototype. This forms the basis from which ADACS-N^{TM} implements the EDF prototype. However, a chance exists that the ADACS-N^{TM} part implementing the EDF prototype could be improved, by making requirements more precise. Since both ADACS-N^{TM} and the formal model implement some functionalities of the EDF prototype, we propose to use the formal model describing part of the EDF prototype to validate part of the implementation.

The EDF Prototype

The EDF prototype implements several control room activities. The main goal of the system is to provide a general overview of the plant status, and to inform the operator about faults, disturbances and unexpected events in the plant leading to a status change in the plant systems [9].

The main UI of the system is called *Global Synthesis* (Figure 5, in French). At the top, six tabs indicate the current status of the plant, ranging from RP (working at full capacity) to RCD (completely stopped), with intermediate status between them.

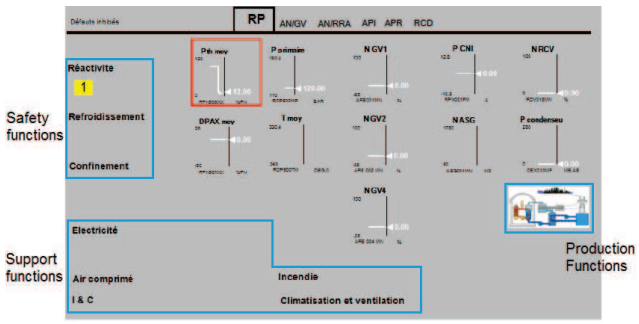

Figure 5: A monitoring system of nuclear-plant control rooms

Depending on the plant status, different reactor parameters are displayed in the middle part of the UI (for instance, the average thermal power of the reactor, "Pth moy"). The Global Synthesis UI groups the reactor parameters that must be constantly monitored. Other parameters are dispersed in other UIs. Each parameter is represented by a widget (Figure 6) containing: the parameter name at the top, a curve with the current and past values of the parameter, minimum and maximum thresholds for the parameter value, and at the bottom, the parameter sensor and its measurement unit.

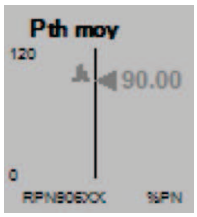

Figure 6: One reactor parameter zoomed out

The system monitors the evolution of the reactor parameters over time. If an *error* occurs in some of them, the parameter is highlighted (with a colored frame around it, for instance). According to [6], an *error* is *"in the total system state, a part of states that may lead to its subsequent failure."*. In this context, an error would be a given reactor parameter whose value increases more that expected (for instance, the average thermal power of the reactor). A *failure* is *"an event that occurs when the perceived behavior of system deviates from its correct behavior."*. In the case of the EDF prototype, a failure would happen if some part of the plant equipment does not behave correctly because the reactor average thermal power is too high. Finally, in [6] a *fault* is defined as *"the adjudged or hypothesized cause of an error"*, which in our example would be the original cause that makes the average thermal power increase. In this paper, we refer to all unexpected events of the reactor that are monitored by the EDF prototype as *errors*.

If an error occurs in a reactor parameter, the parameter is highlighted somehow on the user interface, and a *signal* (e.g., an alert or an alarm condition) is triggered in the left zone of the UI on the corresponding *function* (e.g., under the "réactivité" function in Figure 5). The system monitors 38 functions, 50 reactor parameters, and 5 kinds of signals, triggered by parameters errors: *threshold overflow*, *threshold underflow*, *gradient excess*, *loss of redundancy*, and *invalid measurement*.

The ADACS-N^{TM} System

Atos Worldgrid implemented part of the EDF prototype in their own product called ADACS-N^{TM}, a commercial product that is deployed in actual nuclear plants. ADACS-N^{TM} is a real-time system designed to completely monitor and control a nuclear power plant. It aims at assisting users (i.e., operators) in their daily tasks in a control room [29].

Particularly relevant to our study are the UIs that implement the EDF prototype, such as the UI illustrated in Figure 7. This type of UI presents in curves the evolution of a group of reactor parameters over time, displaying their current and past values, and potential unexpected events in the nuclear unit.

ADACS-N^{TM} structures data by means of *objects*, a configurable component that has inputs, a processing unit, and outputs. Objects can be plugged to each other. An input can be either an acquired value (measurements, signals, etc., accompanied by complementary information such as validity,

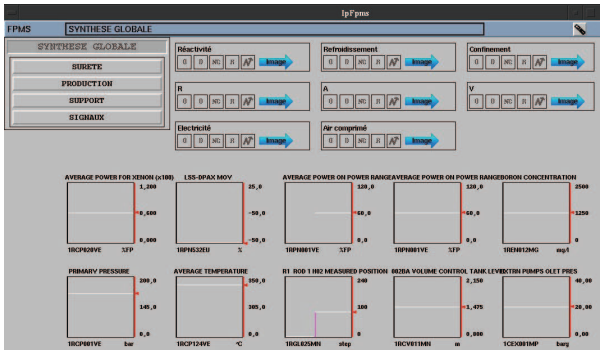

Figure 7: Surveillance display

and time stamps) or an output computed by another object. The input data undergo several calculations in the processing unit, generating the object outputs. In this case study, objects implement the evolution of reactor parameter values in time.

ADACS-N^{TM} can be connected to a stimulator that generates input data (Figure 8). The results of object calculations can be displayed on the ADACS-N^{TM} user interfaces, allowing users to be aware of the current status of the nuclear unit and to take actions correspondingly. ADACS-N^{TM} includes a logging mechanism that records the executions in trace files.

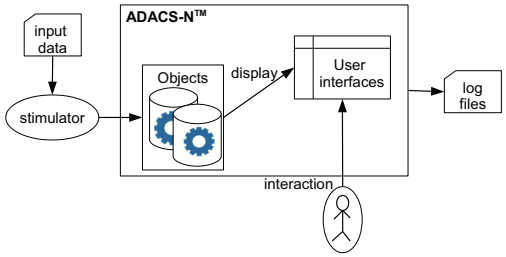

Figure 8: ADACS-N^{TM}: simulation mode and data logging

As the EDF prototype, ADACS-N^{TM} also monitors 38 functions and 50 reactor parameters. However, it can trigger 4 kinds of signals: *super threshold overflow* and *super threshold underflow*, in addition to *threshold overflow* and *threshold underflow* that are implemented by the EDF prototype.

ADACS-N^{TM} also has a simulation mode, which is used either for training sessions to nuclear-plant users, or for data engineering and software testing. Simulation can be used for disproving certain properties by showing examples of incorrect behaviors. Even though this mode provides an environment for preparing the staff before starting their daily activities, simulation explores a part of the system state space. On the contrary, the use of formal models to support validation would consider the entire state space and can thus prove or disprove properties for all possible behaviors [13].

APPLICATION OF ANALYSIS OF TRACES

Analysis of traces (Figure 2) is used to cross check part of the ADACS-N^{TM} implementation.

Formal Model of the EDF Prototype

Following the separation of concerns proposed by the ARCH architecture [7], the formal model of the control room system contains modules describing part of the system functional core, the user interfaces (UIs) and the dialog controller (Figure 9).

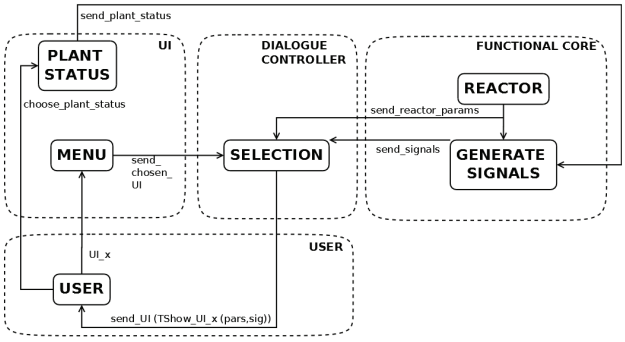

Figure 9: Formal model structure of the EDF prototype

In order to describe part of the functional core, the *reactor* and *generate signals* modules simulate the evolution of several reactor parameters and signals over time. The *selection* module mediates the calculations in the functional core and the interactions on the UIs. Two modules are created to describe the user interfaces, namely *plant status* and *menu*. Beyond ARCH, a special module called *user* is included in the formal model, in order to describe part of the user's behavior.

These modules are coupled as follows: the *user* sets the current *plant status*, which determines the signals that are simulated by the *generate signals* module. This module also receives from the *reactor* module a list of reactor parameters with their current value and status, to simulate signals accordingly. A list of reactor parameters and signals are then sent to the *selection* module from the *reactor* and the *generate signals* modules. *Selection* also receives from the *menu* module the last menu option selected by the user, and filters the parameters and signals for the UI accessible by the menu option. These filtered parameters and signals are then sent to the user, representing the display of such information on the UIs. Thus, we cover UI navigation, UI interaction capabilities, and UI appearance.

Each one of these modules are specified in LNT. In total, the formal model contains 13 modules describing: some activities of the functional core, seven user interfaces and two main activities of the users, within 4444 lines of LNT code (Table 1). The formal model so described can be used to perform formal verification. With this aim, we use CADP to generate an LTS representing the state space of formal model.

Instantiation of the Approach

Figure 10 illustrates how *analysis of traces* is applied to the case study. The right part of the figure consists in stimulating ADACS-N^{TM}. With this goal, a simulation mode allows ADACS-N^{TM} to be connected to a stimulator from which ADACS-N^{TM} receives input data. These input files contain scenarios that change the parameter values of the nuclear

ARCH component	*File*	*# loc*
user interface	plant status	19
user interface	menu	234
functional core	reactor	404
functional core	generate signals	178
functional core	scenarios	451
dialog controller	selection	90
(user)	user	211
(auxiliary file)	scheduler	93
(auxiliary file)	main	111
(auxiliary file)	library	2191
(auxiliary file)	library.tnt	433
(auxiliary file)	reactor.tnt	3
(auxiliary file)	reactor.fnt	26
	TOTAL	4444

Table 1: Summary of the formal model

unit. ADACS-N^{TM} reacts to such stimulation following several physics laws. The results of such calculations are displayed on ADACS-N^{TM} user interfaces, allowing users to be aware of the current status of the nuclear unit and to take actions correspondingly, by interacting with the UIs.

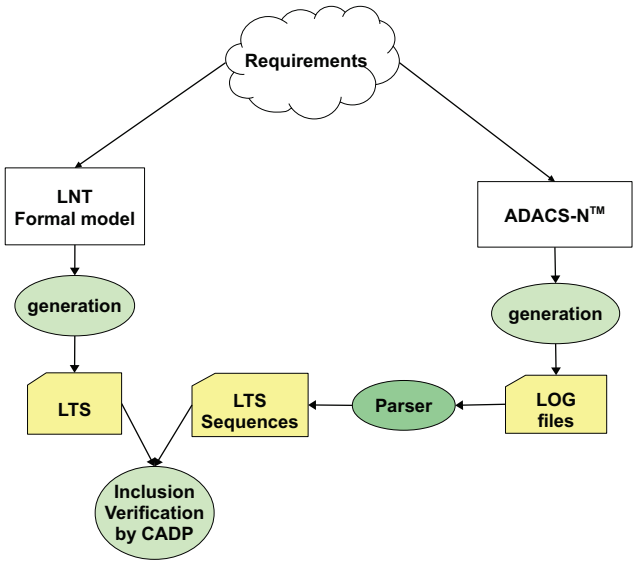

Figure 10: The approach applied to ADACS-N^{TM} validation

The horizontal part of the Figure 10, which links the LNT formal model and ADACS-N^{TM}, consists of a translation of ADACS-N^{TM} log files into a trace that can be searched in the LTS of the formal model. The Parser is used to automate this translation. The information extracted from the trace files are translated into a sequence of transition labels enclosed by a property, such as the one illustrated in Figure 11. Between "<" and ">" (lines 5 and 32), each line corresponds to a labeled transition in an LTS, and should be enclosed by "". Figure 11 identifies the three kinds of actions that are simulated in the scenarios, as they are encoded in the LNT formal model: lines 6 and 7, in red, represent calculations resulting from the stimulation, i.e., the evolution of reactor parameter values over time; lines 12 and 13, in blue, represent the display of such information on the user interfaces; and line 17, in green, represents the menu option the user selected. For legibility reasons, only the beginning of the transition labels are illustrated.

The last step of the approach consists in checking whether the sequence of labels extracted from ADACS-N^{TM} log file is included in the LTS of the formal model, using the EVALUATOR tool. If the trace is found, it means that the formal model simulates the scenario in the same way as ADACS-N^{TM}.

Simulated Scenarios

In order to stimulate ADACS-N^{TM} objects, input files containing 38 scenarios are manually created, containing scenarios that will change an object value. In this study, four kinds of scenarios are analyzed: *threshold overflow*, *threshold underflow*, *super threshold overflow*, and *super threshold underflow*. Figure 12 illustrates one of those scenarios: *super threshold underflow* on the `RPN010MA` reactor parameter. The scenario has 11 instances: it initializes the parameter with its mean value, decreasing it progressively according to internal formulas for each instance. At the instance n.4, the parameter value exceeds its first inferior threshold. At this point, a *threshold underflow* is triggered on the reactor parameter, and an alert signal is generated. The parameter value continues to decrease,

```
1  property P1
2      "SCENARIO N.1"
3  is
4      "main_svl.bcg" |=
5          < true* .
6              "VOIR_PARAMS_REACTEUR..."
7              "VOIR_SIGNAUX"
8              "VOIR_PARAMS_REACTEUR..."
9              "VOIR_SIGNAUX"
10             "VOIR_PARAMS_REACTEUR..."
11             "VOIR_SIGNAUX"
12             "VOIR_VUE !SYNTHESE_GLOBALE"
13             "VOIR_SELECTION !AFFICHER_VUE..."
14             "VUE_SURETE"
15             "VOIR_VUE !SURETE"
16             "VOIR_SELECTION !AFFICHER_VUE..."
17             "VUE_SURETE_REACTIVITE"
18             "VOIR_VUE !SURETE_REACTIVITE"
19             "VOIR_SELECTION !AFFICHER_VUE..."
20             "VUE_SURETE_REACTIVITE_POSITIONGRAPPES"
21             "VUE_SYNTHESE_GLOBALE"
22             "VOIR_VUE !SYNTHESE_GLOBALE"
23             "VOIR_SELECTION !AFFICHER_VUE..."
24             "VOIR_PARAMS_REACTEUR..."
25             "VOIR_SIGNAUX"
26             "VOIR_VUE !SYNTHESE_GLOBALE"
27             "VOIR_SELECTION !AFFICHER_VUE..."
28             "VOIR_PARAMS_REACTEUR..."
29             "VOIR_SIGNAUX"
30             "VOIR_PARAMS_REACTEUR..."
31             "VOIR_SIGNAUX"
32         > true;
33     expected TRUE
34 end property
```

Figure 11: An example of property

until it exceeds its second inferior threshold at instance n.6, triggering a *super threshold underflow* and an alarm condition signal. The parameter value then progressively increases until it reaches its mean value again.

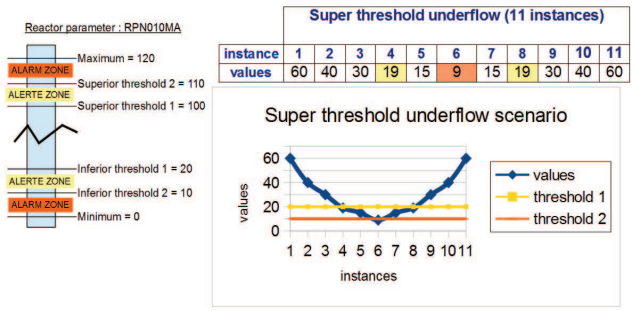

Figure 12: *Super threshold underflow* scenario on a parameter

The stimulator executes the scenario step by step. The results of the calculations are displayed on the user interfaces, in the visual component representing the object. Once an error occurs in some of them, the parameter is highlighted with a colored frame around it, for instance. The user, in turn, interacts with the system by navigating through the UIs in order to have more details about the error. The entire sequence is logged in trace files: the ADACS-N^{TM} object calculations resulted from the stimulation, the display of these information on the user interfaces and the user interactions. Such trace files are then translated into an LTS sequence that can be searched in the LTS of the formal model.

Coverage of the Validation

The LNT model from the case study simulates errors over 50 reactor parameters. In ADACS-N^{TM}, 20 of such parameters have acquired data as input, and can be stimulated with input values (the other 30 parameters are calculated according to the values of the acquired parameters). In this validation, we cover all these 20 parameters at least once. In terms of number of parameters, this validation covers 40% (20/50) of the reactor parameters of the formal model.

The LNT model simulates seven errors over the reactor parameters (i.e., (1) *threshold overflow*, (2) *threshold underflow*, (3) *super threshold overflow*, (4) *super threshold underflow*, (5) *gradient excess*, (6) *loss of redundancy*, and (7) *invalid measurement*). ADACS-N^{TM} simulates four of them: (1)-(4). In this case study, 38 ADACS-N^{TM} log files were analyzed, each one containing an error scenario in one parameter. The formal model simulates the seven kinds of errors over 50 parameters, making a total of 350 scenarios. In terms of simulated scenarios, this validation covers 10% (38/350) of the simulated scenarios of the formal model. Restricting the scope to the 20 parameters and four scenarios that can be simulated in ADACS-N^{TM}, this validation covers 38 scenarios over the 80 (20 parameters * 4 scenarios) possible ones in ADACS-N^{TM}, covering 47% of the scenarios that can be simulated in this part of ADACS-N^{TM}.

The connection of the LNT formal model to ADACS-N^{TM} permitted the analysis of an intersection "zone" (Figure 13).

For the part of ADACS-N^{TM} implementing the EDF prototype, it is a significant coverage: all the four scenarios currently implemented in this part of ADACS-N^{TM} are covered, and all the 20 reactor parameters which can be stimulated in the systems are covered at least once. It is not exhaustive, though. To be exhaustive, input files containing all the four scenarios for all the 20 parameters would be necessary. As a proof of concept of the approach, the current coverage was sufficient.

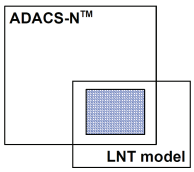

Figure 13: Analyzed part of the system and the formal model

Results of the Validation

The 38 log files containing the scenarios have in total 1710 lines. The Parser was used to generate properties aiming at checking the inclusion of the scenarios in the LTS (the state space) of the formal model. The translation of the scenarios into these properties took around three seconds, and the generated file contained 1700 lines.

Inclusion Checking

We applied model checking to check the satisfiability of the 38 properties describing the scenarios. Preliminary analysis have shown that the traces are not included in the LTS. A deeper analysis showed that ADACS-N^{TM} and the formal model are divergent in the way errors in reactor parameters are synthesized in the signals, under the corresponding reactor function on the left of the UI (cf. Figure 5). This is due to a lack of alignment in the way the requirements were communicated to Atos Worldgrid and our laboratory, making both implementations diverge.

Once this first divergence was observed, the formal model was modified to synthesize errors in the signals in the same way as ADACS-N^{TM}. In this new version of the formal model, all the 38 scenarios are found in the LTS of the formal model, the total time of the inclusion checking is around four minutes.

Our approach demonstrated that several aspects of the EDF prototype are implemented in the same way in both ADACS-N^{TM} and the formal model, which may indicate that the real system implements these aspects of the system according to the requirements, considering the *redundancy* aspect of the approach (i.e., two groups of people interpreting the same set of requirements, to create a formal model and to implement the real system). The following information is present in ADACS-N^{TM} **and** in the LNT formal model:

1. the name of the 20 reactor parameters that are stimulated;

2. the value a reactor parameter is assigned to at each instance of the four analyzed scenarios (i.e., *threshold overflow*, *threshold underflow*, *super threshold overflow*, and *super threshold underflow*);

3. the occurrence of an error once a given reactor achieves a value which is higher than its first superior threshold;
4. the occurrence of an error once a given reactor achieves a value which is lower than its first inferior threshold;
5. the occurrence of an error once a given reactor achieves a value which is higher than its second superior threshold, and this error accumulates with the error # 3;
6. the occurrence of an error once a given reactor achieves a value which is lower than its second inferior threshold, and this error accumulates with the error # 4;
7. the display of such errors on the user interface, in the corresponding signal (among all signals present on the left of the UIs, cf. Figure 5);
8. the list of signals on the left of each user interface;
9. the user interactions, by accessing the menu options;
10. the transmission of the signals between the user interfaces, once the user navigates through them;
11. the update of the user interfaces once a reactor parameter is not in an error state anymore.

Improvements on the Formal Model and Implementation

An advantage of connecting the formal model to a real system is that it brings improvements to both the formal model and the implementation. The following modifications are implemented in the formal model:

1. Alignment of the list of reactor parameters: 27 new parameters are added.
2. Adjustment in the parameter names: some parameters have different names in the requirements and in ADACS-N^{TM}. The formal model now references both terminologies.
3. Definition of the minimum and maximum values of the reactor parameters according to ADACS-N^{TM} thresholds.
4. Addition of four new thresholds to each parameter: two superior and two inferior thresholds, increasing the number of thresholds from 1 to 3 at each extremity (Figure 14). When the value of the parameter cross these thresholds, *alerts* and/or *alarm conditions* are triggered on the system.
5. Addition of two new scenarios, *super threshold overflow* and *super threshold underflow*, increasing the number of scenarios from five to seven. Such errors are triggered in a reactor parameter when its value exceeds its *superior/inferior* thresholds n.2 (Figure 14), in contrast to the *threshold overflow* and *threshold underflow* errors, already present in the model, and triggered when a reactor parameter value exceeds its *superior/inferior* thresholds n.1. The *maximum* and *minimum* thresholds in Figure 14 are not expected to be exceeded.

On the other hand, our approach allowed ADACS-N^{TM} system to be improved in several directions. Some corrections were made in the system, and the following improvements were implemented:

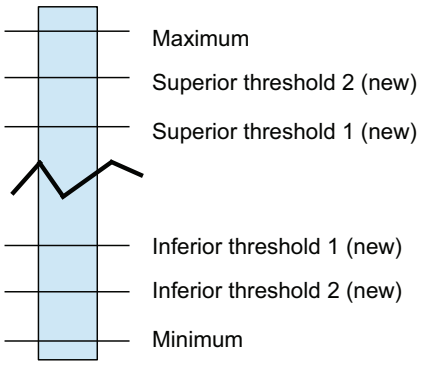

Figure 14: New thresholds in the reactor parameters

1. generation of signals once a reactor parameter receives values that are beyond its thresholds;
2. enhancement of the log with traces of the control-room process, i.e., the acquired values of parameters, the thresholds overflow and underflow, and the generation of alerts and alarm conditions;
3. addition in the log: user interactions with the user interfaces;
4. addition in the log: all UI status changes for inputs, threshold overshooting, triggered alerts;
5. generation of a unique log file, including all logged events, sorted by timestamps.

Limitations of the Validation

Some information is not covered by the validation (e.g., the *gradient excess*, *loss of redundancy*, and *invalid measurement* scenarios; 30 reactor parameters; the name of the user interface displayed once the user accesses a menu option, etc.). Even though they are represented in the formal model, they are not present in the trace files generated by the ADACS-N^{TM} portion implementing the EDF prototype.

Several modules of the ADACS-N^{TM} system were not covered by the formal model, and were not subject to validation by this approach.

Currently, the Parser accepts only ADACS-N^{TM} log files, and generates LTS sequences for this case study. An effort would be necessary to generalize the Parser so that it could be applicable to other case studies.

DISCUSSION

Formal methods have been largely used for improving systems under design, and to enhance requirements before developing the system. Alternatively, we propose to use formal methods to improve existing systems too. Clearly, an advantage is that the approach can be integrated to the V&V environment of already implemented and still evolving systems, which justifies the creation of formal specifications (by experts in formal methods) of systems that have already been implemented.

Checking whether an interactive system implements its requirement as expected is a classic activity of validation, which has been largely done in system design. The novelty of our proposition is the integration of log files with formal methods to perform such validation.

Besides the validation of the real interactive system with respect to its requirements using formal models, the connection of a formal model to a real system is a fruitful source of improvements to both the formal model and the implementation. It improves both the real interactive system and the formal model: the results of the analysis are used to polish the real system, and to increase the realism of the formal model. Preliminary analyses improve the formal model in several directions and approximates it to the real interactive system.

A positive effect of the approach is that it allows the formal model to be cross checked too. One of the challenges in model-based approaches is to ensure the reliability of the model. Since these approaches highly rely on the models, models are expected to be as representative as possible. Hand-written models have the advantage of being subject to human analysis and reasoning. Depending on the designer expertise, a good understanding of the system may result in a good model. However, even good designers may have a misunderstanding of the system, and consequently model it incorrectly, not to mention that hand-written models are error-prone. The approach proposed in this paper provides a means to mitigate such difficulties. The application of such techniques aiming at connecting the formal model to a real system mitigates the *fidelity* issue (i.e., there is no guarantee that the models really correspond to the system), one of the reasons why there are few case studies of formal methods applied to industrial systems [10]. The connection of the formal model to the implementation brings fidelity to the formal model.

However, *analysis of traces* relies on the quality of the traces. It is not possible to completely re-construct a model of the system from the system logs, and this is not the goal of the approach. The approach is rather meant to validate certain system functionality with respect to the requirements. In other words, to check whether certain scenarios executed on the real system can be found in the state space of the formal model, which may indicate that the system implements the requirements as expected. This does not call for exhaustive analysis, and it does not mean that the checking coverage relies on the testing coverage. The checking rather relies on the quality of such log files: the more the system logs cover aspects of the system, the larger the coverage of the system validation will be. Rather than propose techniques that rely on all parts of a system being covered, a more piecemeal approach allows progress to be made and useful results determined, when the focus of the modeling activity is narrowly directed at only certain parts of a system [12].

In order to check a given functionality, the functionality should be included in the logging mechanism of the system, and with a certain level of details so that the analysis is useful. In order to automate the execution of the system, and subsequently generation of log files, testing tools such as `Selenium`[4] can be used, which allows the execution of systems to be automated.

Once a certain number of log files are available, the usage of a formal model to validate the interactive system through the inclusion checking provided by the analysis of traces is beneficial to the real system. Not to mention that the formal model itself can be used in other ways to improve quality of the analyzed interactive system [20], such as the verification of properties [27, 19, 23, 24].

Finally, the approach can be applied to more general cases, for instance, to compare two formal descriptions (regardless whether formal specifications or implementations) that are independently derived from informally stated requirements. Going further, *bisimulation equivalence* can be verified between them [22], for instance.

CONCLUSION

This paper presents our investigations of using formal models to validate interactive systems with respect to the initial requirements. A trace-based approach is described in detail. The novelty of this work is the coupling of formal techniques and analysis of traces, by checking system traces over the state space of the formal model. Besides validating the initial requirements of the system, such analysis can bring improvements to both the real interactive system and its formal model.

The approach was applied to analyze part of an industrial system in the nuclear plant domain called ADACS-N^{TM}. After an initial alignment, several correspondences were shown between ADACS-N^{TM} and the formal model, which may indicate that the real system implemented the requirements correctly, given the *redundancy* aspect of the approach. More importantly, one major mismatch was found in the way both ADACS-N^{TM} and the formal model synthesize signals once a reactor parameter has an error.

The application of our approach to the analysis of an industrial system indicates that the approach scales well to real-life application. A Parser was implemented to translate logs into LTS sequences, of which the main ideas can be re-used to further connect other formal models to interactive systems. Further work is needed, though, to make the Parser more generic and applicable to any case study. In the future, further investigation could be conducted on the two alternative propositions: *test case generation* and *co-simulation*, as well as a study about the rationale of using each one of them. In addition, we also aim at using log file analysis in combination with other testing techniques, such as mutation testing, in an integrated V&V environment for interactive systems.

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). We thank Frédéric Lang and Hubert Garavel, researchers at INRIA Rhône-Alpes, and Franck Etienne and Olivier Deschamps, from Atos Worldgrid, for their collaboration to this work.

[4] http://www.seleniumhq.org/

REFERENCES

1. IEEE Std 1012-2004. 2005. IEEE Std 1012 - 2004 IEEE Standard for Software Verification and Validation. (2005).
2. Hanan Al-Zawahreh and Khaled Almakadmeh. 2015. Procedural Model of Requirements Elicitation Techniques. In *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication (IPAC '15)*. ACM, New York, NY, USA, Article 65, 6 pages.
3. Ermeson Andrade, Paulo Maciel, Gustavo Callou, and Bruno Nogueira. 2009. A methodology for mapping sysml activity diagram to time petri net for requirement validation of embedded real-time systems with energy constraints. In *Digital Society, 2009. ICDS'09. Third International Conference on*. IEEE, 266–271.
4. James H. Andrews and Yingjun Zhang. 2000. Broad-spectrum Studies of Log File Analysis. In *Proceedings of the 22Nd International Conference on Software Engineering (ICSE '00)*. ACM, New York, NY, USA, 105–114.
5. A Arnold, MC Gaudel, and B Marre. 1998. An Experiment on the Validation of a Specification by Heterogenous Formal Means: The Transit Node. *DEPENDABLE COMPUTING AND FAULT TOLERANT SYSTEMS* 10 (1998), 37–56.
6. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.* 1, 1 (Jan. 2004), 11–33.
7. Len Bass, Reed Little, Robert Pellegrino, Scott Reed, Robert Seacord, Sylvia Sheppard, and Martha R Szezur. 1991. The ARCH model: Seeheim Revisited. In *User Interface Developpers' Workshop*.
8. David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe, and Gideon Smeding. 2014. Reference Manual of the LNT to LOTOS Translator (Version 6.1). (Aug. 2014). INRIA/VASY and INRIA/CONVECS, 131 pages.
9. François Chériaux, Dominique Galara, and Marion Viel. 2012. 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)*. Curran Associates, Inc., San Diego, 1002–1012.
10. Darren Cofer. 2012. Formal Methods in the Aerospace Industry: Follow the Money. In *Proceedings of the 14th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering (ICFEM'12)*. Springer-Verlag, Berlin, Heidelberg, 2–3.
11. Bruno d' Ausbourg. 1998. Using Model Checking for the Automatic Validation of User Interfaces Systems. In *Design, Specification and Verification of Interactive Systems '98*, Panos Markopoulos and Peter Johnson (Eds.). Springer Vienna, 242–260.
12. Bob Fields, Nick Merriam, and Andy Dearden. 1997. *Design, Specification and Verification of Interactive Systems '97: Proceedings of the Eurographics Workshop in Granada, Spain, June 4–6, 1997*. Springer Vienna, Vienna, Chapter DMVIS: Design, Modelling and Validation of Interactive Systems, 29–44.
13. H Garavel and S Graf. 2013. Formal Methods for Safe and Secure Computer Systems. *Federal Office for Information Security* (2013).
14. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. 2013. 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.
15. Colin Hood, Simon Wiedemann, Stefan Fichtinger, and Urte Pautz. 2007. *Requirements management: The interface between requirements development and all other systems engineering processes*. Springer Science & Business Media.
16. Lester O. Lobo and James D. Arthur. 2005. Local and Global Analysis: Complementary Activities for Increasing the Effectiveness of Requirements Verification and Validation. In *Proceedings of the 43rd Annual Southeast Regional Conference - Volume 2 (ACM-SE 43)*. ACM, New York, NY, USA, 256–261.
17. Paolo Masci, Yi Zhang, Paul Jones, Paul Curzon, and Harold Thimbleby. 2014. *Fundamental Approaches to Software Engineering: 17th International Conference, FASE 2014, Grenoble, France, April 5-13, 2014*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Formal Verification of Medical Device User Interfaces Using PVS, 200–214.
18. Radu Mateescu and Damien Thivolle. 2008. A Model Checking Language for Concurrent Value-Passing Systems. In *FM 2008 (Lecture Notes in Computer Science)*, Jorge Cuellar and Tom Maibaum (Eds.), Vol. 5014. Springer Verlag, Turku, Finlande, 148–164.
19. David Navarre, Philippe A. Palanque, Jean-François Ladry, and Eric Barboni. 2009. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM TOCHI* (2009), 18:1–18:56.
20. Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73.
21. Raquel Oliveira, Sophie Dupuy-Chessa, and Gaelle Calvary. 2014. Formal Verification of UI Using the Power of a Recent Tool Suite. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '14)*. ACM, New York, NY, USA, 235–240.

22. Raquel Oliveira, Sophie Dupuy-Chessa, and Gaëlle Calvary. 2015a. Equivalence Checking for Comparing User Interfaces. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 266–275.

23. Raquel Oliveira, Sophie Dupuy-Chessa, and Gaëlle Calvary. 2015b. Plasticity of User Interfaces: Formal Verification of Consistency. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 260–265.

24. Raquel Oliveira, Sophie Dupuy-Chessa, and Gaëlle Calvary. 2015c. Verification of Plastic Interactive Systems. *De Gruyter publication Journal of Interactive Media (i-com)* 14(3) (2015), 192–204.

25. Lydia Schneidewind, Stephan Hörold, Cindy Mayas, Heidi Krömker, Sascha Falke, and Tony Pucklitsch. 2012. How Personas Support Requirements Engineering. In *Proceedings of the First International Workshop on Usability and Accessibility Focused Requirements Engineering (UsARE '12)*. IEEE Press, Piscataway, NJ, USA, 1–5.

26. Souvik Sengupta and Ranjan Dasgupta. 2015. Use of Semi-Formal and Formal Methods in Requirement Engineering of ILMS. *SIGSOFT Softw. Eng. Notes* 40, 1 (Feb. 2015), 1–13.

27. M. Sousa, J.C. Campos, M. Alves, and M.D. Harrison. 2014. 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, 62–67.

28. WT Tsai, X Bai, B Huang, G Devaraj, and R Paul. 2000. Automatic Test Case Generation for GUI Navigation. In *Quality Week*, Vol. 2000.

29. Atos Worldgrid. 2011. Modernizing Data Processing for EDF Energy at Dungeness – Improving Performance and Standardizing Technology for EDF Energy. (2011).

30. Ayesha Yasmeen, Karen M. Feigh, Gabriel Gelman, and Elsa L. Gunter. 2012. Formal Analysis of Safety-critical System Simulations. In *Proceedings of the 2Nd International Conference on Application and Theory of Automation in Command and Control Systems (ATACCS '12)*. IRIT Press, Toulouse, France, France, 71–81.