HAL
open science

Formal Specification and Verification of Interactive Systems with Plasticity : Applications to Nuclear-Plant Supervision

Raquel Araùjo De Oliveira

To cite this version:

Raquel Araùjo De Oliveira. Formal Specification and Verification of Interactive Systems with Plasticity : Applications to Nuclear-Plant Supervision. Computation and Language [cs.CL]. Université Grenoble Alpes, 2015. English. NNT : 2015GREAM025 . tel-01253619

HAL Id: tel-01253619

https://theses.hal.science/tel-01253619v1

Submitted on 11 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Raquel Araújo de Oliveira

Thèse dirigée par **Sophie Dupuy-Chessa** et **Hubert Garavel**
et co-encadrée par **Frédéric Lang** et **Gaëlle Calvary**

préparée au sein du **Laboratoire d'Informatique de Grenoble**,
du **Centre de Recherche Inria Grenoble-Rhône-Alpes**
et de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Formal Specification and Verification of Interactive Systems with Plasticity:

Applications to Nuclear-Plant Supervision

Thèse soutenue publiquement le **03 décembre 2015**,
devant le jury composé de :

M. Ioannis Parissis
Professeur à l'Institut Polytechnique de Grenoble, Président
M. Philippe Palanque
Professeur à l'Institut de Recherche en Informatique de Toulouse (IRIT), Rapporteur
M. Yamine Aït-Ameur
Professeur à l'École Nationale Supérieure d'Électronique, d'Électrotechnique, d'Informatique, d'Hydraulique et des Télécommunications, Toulouse, Rapporteur
M. José Creissac Campos
Maître de conférence à University of Minho, Portugal, Examinateur
M. François Chériaux
Ingénieur-chercheur à Électricité de France (EDF), Paris, Examinateur
Mme Sophie Dupuy-Chessa
Professeur à l'Université Pierre-Mendès-France, Grenoble, Directrice de thèse
M. Hubert Garavel
Directeur de Recherche à l'Inria, Grenoble, Co-directeur de thèse
M. Frédéric Lang
Chargé de Recherche à l'Inria, Grenoble, Co-encadrant de thèse
Mme Gaëlle Calvary
Professeur à l'Institut Polytechnique de Grenoble, Co-encadrante de thèse

I dedicate this thesis to you, NH. Your continuous support and love throughout the writing of this thesis and also within my own life helped me in more ways than you probably realize. In the vastness of space and immensity of time, it is my joy to spend a planet and an epoch with you.

Acknowledgments

I am very grateful to my thesis committee members: Philippe Palanque and Yamine Aït-Ameur, for accepting to evaluate this manuscript, Ioannis Parissis, for agreeing to head the committee, José Creissac Campos and François Chériaux, for agreeing to be examiners. I would like to express my gratitude to Laurence Nigay and Radu Mateescu, directors of the IIHM and CONVECS teams, where I worked for the past three years, for their words of encouragement. Immeasurable appreciation and deepest gratitude to my Ph.D. advisors: Frédéric Lang, whose strong theoretical and technical skills were of immense help for the accomplishment of this work; Hubert Garavel, whose rigor required me to produce high quality work in all my endeavors, inspiring me to always pursue excellence in every piece of work I have ever done; Gaëlle Calvary, who is very passionate and enthusiastic about human-computer interaction. I am specially grateful to Gaëlle for giving me the first opportunity to teach, and for her feedbacks during the course of this thesis; finally, I am deeply thankful for the strong implication of Sophie Dupuy-Chessa in this thesis. For always being receptive to dialog, for her remarkable patience in guiding a young researcher like me, and for her valuable comments and suggestions in all my writings and rehearsals. Her ultimate concern for the welfare of her students is noteworthy.

I acknowledge the funding sources from the Connexion Project that made this work possible. Especially Catherine Devic (EDF), Danièle Dadolle (Atos Worldgrid), and François-Xavier Dormoy (Esterel Technologies), for providing the reading material and support needed to accomplish this work. My sincere thanks to Franck Etienne and Olivier Deschamps, for welcoming me several times at Atos Worldgrid.

I am thankful to colleagues of both IIHM and CONVECS teams who have helped me in one way or another: François Bérard, Éric Céret, Yann Laurillau, William Delamare, Maxime Guillon, Élisabeth Rousset, my good friend Miratul Mufida, and all members of the IIHM team; José Ignacio, Eric Léo, Hugues Evrard, Rim Abid, Jingyan, and all members of the CONVECS team. Special thanks to Fatma Jebali for being a wonderful and generous friend, for the time spent on weekends in the final stages of this thesis. I also thank the administration staff from both teams for kindly helping me with the French bureaucracy: Myriam Etienne, Helen Pouchot, Pascale Poulet, Antoine Alexandre, and Stéphanie Bellier. I appreciate working with Vanda Luengo, Vincent Lestideau, and Aurélien Faravelon, with whom I had the pleasure to teach.

I warmly thank my family for always supporting me: my grandmother, Cleonice, my father Valdeci, and my other relatives: Vanice, Valterli, Vilson, Anidia, Odorico, Regina, Rosangela, Roberto, and Rafael: your positive thoughts and cheering could be felt despite the miles of distance separating us. I also thank Jander, who motivated me to come to France in the first place; Kiev and Liviany, who helped me to get settled in France; and Thiago Mori, who shared a roof and his friendship with me when my Ph.D. journey started. I would like to express my warm thanks to Hélène Deladoire and Pascal Gaillard, for their cheering. I acknowledge the help of friends who offered me the comfort one needs, making my time in France pass smoothly: Hamid Mirisaee, Saman Noorzadeh, Laura Venuti, Divya Gupta, Vania, Zeina Wazani, Muhammad Sam, Kassiana Lima, Ana Maria, Pauline Nasatto, Andon Tchechmedjiev, and all those I have not mentioned, but who I'll always think about. I cannot thank enough my boyfriend, Nicolas Hili, for making the conclusion of this Ph.D. less burdensome. Your endless patience during the "writing nights and weekends" of a finishing Ph.D. candidate, your taking care of whatever was needed without complaining, just so I could focus on completing my dissertation, and constant presence during the tough moments were immeasurable helpful.

Contents

List of Figures **viii**

List of Tables **xii**

1 Introduction **1**
1.1 Human-Computer Interaction . . . 1
1.2 Interactive Systems . . . 2
1.3 Plastic User Interfaces . . . 3
1.4 Safety-Critical Systems . . . 6
1.5 Nuclear-Plant Systems . . . 7
1.6 Research Question . . . 8
1.7 Quality of Interactive Systems . . . 9
1.8 Context of the Thesis . . . 10
1.9 Outline of the Thesis . . . 10

2 State of the Art **13**
2.1 Goals . . . 13
2.2 Modeling . . . 14
2.3 Model-based Testing . . . 14
2.4 Formal Verification . . . 15
2.5 Criteria to Analyze the State of the Art . . . 17
2.6 Verification of Properties . . . 18
2.7 Assessing Consistency . . . 54
2.8 Summary . . . 68

3 The Nuclear Reactor Supervision Case Study **69**
3.1 Goals . . . 69
3.2 Nuclear-Plant Control Rooms . . . 69
3.3 The EDF Prototype . . . 72
3.4 The LIG Prototype . . . 76
3.5 The ADACS-N^{TM} Prototype . . . 80
3.6 Summary . . . 82

4 An Approach to Verifying Interactive Systems **83**
4.1 Goals . . . 83
4.2 Formal Techniques . . . 84

4.3 Global Approach . . . 85
4.4 Formal Models Based on the ARCH Architecture . . . 87
4.5 Languages and Tool Support . . . 88
4.6 Summary . . . 93

5 Verification of Industrial Interactive Systems 95
5.1 Goals . . . 95
5.2 Formal Model of the Case Study . . . 96
5.3 Propositions to Connect to Industrial Systems . . . 109
5.4 On the Connection to an Industrial System . . . 113
5.5 Conclusions of the Connection . . . 128
5.6 Summary . . . 129

6 Verification of Plastic Interactive Systems 131
6.1 Goals . . . 131
6.2 Improvements in the Formal Model . . . 132
6.3 Needs Raised by Plasticity . . . 136
6.4 Propositions to Verify Plasticity . . . 137
6.5 On the Comparison of User Interfaces . . . 144
6.6 Application of the Approach . . . 151
6.7 Summary . . . 159

7 Validation of the Use of Formal Verification for Interactive Systems 161
7.1 Goals . . . 161
7.2 The SRI Display System . . . 162
7.3 Formal Model . . . 164
7.4 Verification Approach . . . 167
7.5 Properties . . . 168
7.6 Results and Discussion . . . 169
7.7 Summary . . . 172

8 Conclusion and Perspectives 173
8.1 Summary of Contributions . . . 173
8.2 Perspectives . . . 175

Appendices 179

A Reactor Parameters 179

B An Excerpt of a LNT Specification 181

C MCL Properties 201

D SVL Scripts 207

Bibliography 211

List of Figures

1 An overview of the disciplines which contribute towards HCI. 2
2 Examples of bad user interfaces 3
3 Various platforms in which interactive systems can execute 4
4 The problem space of plastic user interfaces 5
5 A home heating control system 6

6 Model checking 17
7 Equivalence checking 17
8 A button modeled as an agent 21
9 Architecture of a CNUCE interactor 23
10 Scrollbar modeling with three CNUCE interactors: mouse, cursor and scrollbar . 23
11 The TLIM approach 24
12 The ADC interactor 26
13 A scrollable list as a composition of ADC interactors 26
14 The York interactor 27
15 An icon modeled with a York interactor 28
16 IVY - a tool for verifying interactive systems 30
17 The CERT interactor 31
18 Boolean flows of a CERT interactor representing a push button 31
19 A verification environment using CERT interactors 32
20 UI prototyping and verification 33
21 An example of a user interface with a dial and a slider 34
22 Example of an object in the ICO formalism – an ATM system 35
23 Example of a Petri net for the *Obtain_Cash* task 36
24 The PIE model 38
25 The Red-PIE model 38
26 *Presentation model* of a user interface 40
27 *Presentation and interaction model* of a home heating control system 40
28 The IFADIS framework 44
29 Property specification patterns in CTL 44
30 Examples of infusion pumps 46
31 Verification approach using PVS 47
32 The Gryphon translator framework 48
33 LUSTRE specification transformations 49
34 Automated UI testing process 54
35 Example of a image comparison 55

36 GUIDiff representation of the current state of a UI as a widget tree 56
37 Example of user interfaces for displaying shapes 57
38 PMs of the user interfaces 58
39 Behavioral sets of the user interfaces 58
40 PIMs for UI_A and UI_{C5} 59
41 Comparing systems with their user manuals 60
42 The Niki T34 Syringe pump 61
43 Mode-control panel 62
44 Degani *et al.*'s approach based on the composition of finite-state machines 63
45 Rushby *et al.*'s approach based on invariant verification 64
46 Classification of actions for human-machine interactions 65

47 Overall architecture of a control room system 70
48 A conventional nuclear power plant control room 70
49 A computerized nuclear power plant control room 71
50 A monitoring system of nuclear-plant control rooms 73
51 One reactor parameter zoomed out 73
52 Signals triggered on reactor functions 75
53 Example of the *zoom metaphor*: from one UI, the user can access other UIs that give more details about what is displayed on the previous UI 76
54 Two versions of a control room system 77
55 UI platform adaptation 79
56 UI after user adaptation 79
57 ADACS-N^{TM}: synoptic representations of the installation 80
58 ADACS-N^{TM}: examples of user interfaces 81
59 ADACS-N^{TM} objects 81
60 ADACS-N^{TM}: simulation mode and data logging 82

61 Model checking applied to interactive systems 85
62 Equivalence checking applied to interactive systems 86
63 Global approach to verifying interactive systems 87
64 ARCH architecture usage in the formal modeling 88
65 The Ocis (*Open/Caesar Interactive Simulator*) tool 90
66 A Labeled Transition System (LTS) 91
67 A UI fragment represented in an LTS 92
68 Tools used in the global approach 94

69 Main modules of the formal model 97
70 Formal model structure of the EDF system 97
71 The *plant status* module - an excerpt of Lnt code 99
72 A menu of the control room system 99
73 Extract of the reactor parameter modeling in Lnt 100
74 The *threshold overflow* ("dépassement haut") scenario simulated in a reactor parameter 101
75 In five cycles of anomalies all reactor parameters are affected by all anomaly scenarios 102

76 The *user module* - an excerpt of LNT code . . . 105
77 Models of the case study . . . 110
78 Analysis of traces proposition . . . 111
79 Test case generation proposition . . . 112
80 Co-simulation proposition . . . 113
81 New thresholds in the reactor parameters . . . 114
82 Analysis of traces in details . . . 117
83 Example of inclusion verification of traces . . . 118
84 The *super threshold underflow* anomaly scenario on a reactor parameter . . . 118
85 Example of an ADACS-N^{TM} input file . . . 119
86 An ADACS-N^{TM} log file (adapted) . . . 121
87 A sequence of transition labels . . . 122
88 A property containing a sequence of transition labels . . . 123
89 The `Parser` class diagram . . . 123
90 Correspondence between the ADACS-N^{TM} log files and the translated traces . . . 124
91 Property verification . . . 124
92 Validation coverage of ADACS-N^{TM} $\times$ the LNT formal model . . . 126
93 Intersection zone of ADACS-N^{TM} and the LNT formal model which is analyzed . . . 128

94 The LIG prototype of the control room system running on a PC . . . 132
95 New formal model structure . . . 133
96 The *parameters* module - an excerpt of LNT code . . . 134
97 The *signals module* - an excerpt of LNT code . . . 135
98 Common part to all propositions . . . 137
99 Modeling UI versions . . . 138
100 Excerpt of LNT code describing the UI modules . . . 139
101 Modeling a plasticity engine . . . 140
102 Excerpt of LNT code describing the adaptation rules . . . 140
103 Excerpt of LNT code of each UI version . . . 142
104 Comparison of user interfaces . . . 144
105 Equivalence checking of user interfaces . . . 145
106 An ISLTS (*Interactive System LTS*) . . . 145
107 UI appearance in an ISLTS . . . 146
108 Two strongly equivalent ISLTS . . . 147
109 Two branching equivalent ISLTS . . . 148
110 *Generalization* abstraction technique . . . 149
111 Example of an inclusion relation . . . 151
112 The five UI versions are compared by means of the ISLTS comparison . . . 152
113 ISLTS fragments of PC and Smartphone UIs . . . 153
114 *Generalization* in an ISLTS - case 1 . . . 153
115 *Generalization* in an ISLTS - case 2 . . . 154
116 *Elimination* in an ISLTS . . . 154
117 The PC version includes the Tablet version . . . 155
118 ISLTS in LNT code . . . 156
119 Example of a SVL script . . . 157
120 Summary of the abstraction techniques . . . 158

121 Different levels of equivalence between UI models 159
122 The abstraction problem 159

123 Displayer modes 162
124 Main UI of the SRI system 163
125 Organization of the UI zones and data 163
126 The *Settings* UI of the SRI system 164
127 Formal model structure of the SRI system 165
128 Functionalities of the displayer. In green, the ones covered by the formal model. In red, otherwise 167
129 Verification approach of the SRI system 168
130 Graph of UI navigation of the SRI system 169
131 A property of the SRI system in MCL 170
132 Counter-example of the non-satisfied property (in French) 170
133 The *Settings* UI does not provide direct access to the *SRI Status* UI 171

List of Tables

1 Summary of approaches to verifying system properties 53
2 Summary of approaches which assess consistency 67

3 Reactor parameters anomalies and their UI representation 74

4 Anomaly scenarios and the number of *instants* 102
5 Summary of the formal model of the EDF system 106
6 Size of the LTS of the EDF system model 106
7 Summary of properties of the EDF system 107
8 Summary of the new version of the formal model, to connect to ADACS-N™ . . 115
9 Size of the LTS of the new version of the formal model 115
10 Summary of the Parser 124

11 Summary of the new version of the formal model, to include plastic UIs 135
12 Size of the LTS of the new version of the formal model, to include plastic UIs . . 136
13 Summary of propositions to verify plastic user interfaces 143
14 Summary of the formal models in the different contexts of use 156
15 Summary of the comparisons 157

16 Summary of the formal model of the SRI system 166
17 Size of the LTS of the SRI formal model 166

18 Summary and comparison with the state of the art 175

19 Some reactor parameters of the EDF case study 179

Chapter 1

Introduction

Contents

1.1 Human-Computer Interaction 1
1.2 Interactive Systems 2
1.3 Plastic User Interfaces 3
1.4 Safety-Critical Systems 6
1.5 Nuclear-Plant Systems 7
1.6 Research Question 8
1.7 Quality of Interactive Systems 9
1.8 Context of the Thesis 10
1.9 Outline of the Thesis 10

1.1 Human-Computer Interaction

Human-Computer Interaction (HCI) is a discipline that studies the use of computer technology by humans and the means to improve this interaction. As noticed by [Bertino 1985, Mital & Pennathur 2004, Baecker & Buxton 2014], anger and frustration are the norm rather than the exception: "Users of advanced hardware machines are often disappointed by the cumbersome data entry procedures, obscure error messages, intolerant error handling and confusing sequences of cluttered screens. In particular, novice users feel frustrated, insecure and even frightened when they have to deal with a system whose behavior is incomprehensible, mysterious and intimidating." [Bertino 1985].

A lot of research has been conducted in the past years to mitigate such problems, making HCI a complex (nonetheless, rich) field: beyond computer science, several disciplines contribute to research in HCI. A list of such disciplines is illustrated in Figure 1. In particular, Software Engineering and Ergonomics are domains that have largely contributed to HCI in the past years [Abowd *et al.* 1992, Bastien & Scapin 1993, Vanderdonckt 1994, Long 1989, Bevan 2001, Imaz & Benyon 2007, Coutaz & Calvary 2012]. Human-Computer Interaction needs Software Engineering to address the design and development of useful and usable systems [Göransson *et al.* 2003]. They are both concerned with *requirements analysis*, *incremental and iterative design*, as well as *quality assurance* [Coutaz & Calvary 2012]. Ergonomics, on the other hand, gives principles of design and/or evaluation to be observed to obtain and/or to guarantee an ergonomic, usable, and useful human-computer interfaces [Vanderdonckt 1994]. All such improvements in human-computer interfaces improves the human-computer interaction.

Although computer systems had been largely used for commercial and industrial purposes long ago, HCI became an important issue only since the 80s. According to [Booth 1989], the

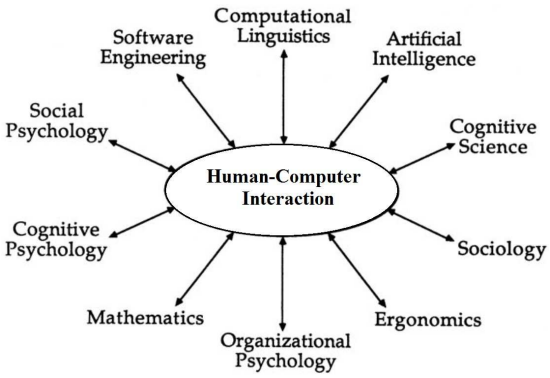


Figure 1: An overview of the disciplines which contribute towards HCI. Adapted from [Booth 1989]

main reason is that in previous decades early users were themselves programmers and designers of computer systems. There has been since then a substantial growth in the number of users who are neither programmers nor designers. This occurred with the emergence of personal computing in the late 70s. Personal computing, including both personal software (text editors, spreadsheets, and computer games) and personal computer platforms (operating systems, programming languages, and hardware), made everyone a potential computer user, and vividly highlighted the deficiencies of computer systems for those who wanted to use computers as tools [Carroll 2013]. This shifts the attention to one kind of computing systems: the *interactive systems*.

1.2 Interactive Systems

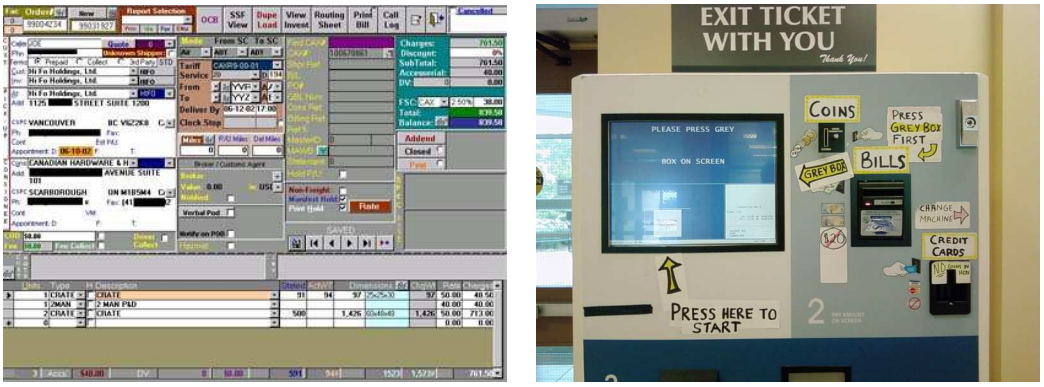
Interactive systems are computer systems characterized by significant amounts of interaction between humans and computers [Spring 2002]. Prime examples of interactive systems are Macintosh or Windows computer operating systems. Other examples are games, editors, web browsers, and Integrated Development Environments (IDE). All such systems involve a high degree of human-computer interaction [Spring 2002].

The first interactive systems were command-line based [Spring 2002]: in these applications, users were required to know the commands for each task they would like to perform. DOS (*Disk Operating System*) was a classic example. Gradually, this kind of systems was replaced by menu-form and dialog-based systems, in which data was entered through forms or dialog boxes, providing users with a limited set of choices. Automatic Teller Machines were an example of this generation of interactive systems. A third generation was introduced by Xerox Corporation in the 80s. The Xerox Star prototype made mouse, icons, desktop, windows and bitmap displays to work together, replicated in the Lisa and Macintosh in the mid-80s. Later in the 90s, Microsoft made the windows, icon, menu, and pointer (WIMP) approach universal. In this period, attention started to be paid to graphical user interfaces (GUI) [Spring 2002].

In this thesis, we call "user interface" (UI) a group of graphical components which are displayed at the same time on a screen, and permit users to interact with the system. Used in plural form, "user interfaces" can designate either the set of user interfaces of a specific system or the user interfaces of any system.

According to [Preece *et al.* 1994], the purpose of an interactive system is to help the user in accomplishing his goals for some application domain. User interfaces play a central role in the accomplishment of these goals. They are expected to provide users with all information and functions they need, and no more than that, to accomplish their goal in a correct manner.

However, one can observe nowadays that many user interfaces do not achieve this requirement [Nielsen & Landauer 1993, Olsen Jr 2007, Miñón *et al.* 2014]. For instance, overloaded UIs can disturb users and delay them from identifying useful information (Figure 2a), and confusing UIs can lead users to mistakes (Figure 2b). Nowadays, UIs are expected not only to provide means for users to accomplish a goal in a correct manner, but also to do so in an intuitive and non-ambiguous manner.



(a) website: “Rules to Better Interfaces General” (b) blog: “Recognizing a bad UI at first glance”

Figure 2: Examples of bad user interfaces

1.3 Plastic User Interfaces

With the advent of technology over the past years, new forms of interaction and new devices emerged: user interfaces are expected to cope with this innovation (Figure 3). The advance of ubiquitous computing and the increasing diversity of platforms and devices change user expectations. Systems should be able to adapt themselves to their *context of use* [Calvary *et al.* 2003], which consists of: the *platform* (e.g., a smartphone or a tablet), the *user* who interacts with the system (e.g., experts or novices), and the *environment* in which the system is executed (e.g., a dark room or outdoor). User interfaces are expected to be sensitive to this context. *Plasticity* is the capacity of a user interface to withstand variations in the context of use (platform, user, environment) while preserving usability [Thevenin & Coutaz 1999].

The dimensions of adaptation have been studied over the past years [Vanderdonckt *et al.* 2008, Calvary *et al.* 2011]. In [Vanderdonckt *et al.* 2008] the problem space of plastic UIs was defined, in which seven dimensions were identified (Figure 4):

1. *Adaptation means*: defines the means used for adaptation. UI *re-molding* denotes any UI reconfiguration that is perceivable by the user and that results from transformations on the UI. By contrast, UI *redistribution* denotes the re-allocation of the UI components to different interaction devices;

Figure 3: Various platforms in which interactive systems can execute

2. *UI component granularity*: denotes the smallest software UI unit that can be affected by re-molding and/or redistribution;
3. *State recovery granularity*: characterizes the granularity after adaptation has occurred (from the session level to the user's last action);
4. *UI deployment*: is a way to characterize how much adaptation has been pre-defined at design-time vs computed at runtime;
5. *Context of use*: is defined by the triplet $\langle platform, user, environment \rangle$ in which the UI is executed;
6. *Technological spaces coverage*: is a way to characterize the degree of technical heterogeneity supported by the system;
7. *Existence of a meta-UI*: a meta-UI allows end-users to program (configure, and control) their interactive spaces, to debug (evaluate) them, and to maintain and re-use programs.

Plasticity provides users with UIs that respond better to their needs. For example, a heating control system [Coutaz *et al.* 2000] can be controlled at home, through a dedicated wall-mounted device or through a handheld device connected to a wireless home-net; it can be used in the office, through the Web, using a standard work station, or anywhere using a mobile phone. Figure 5 illustrates two examples of a user interface of such heating control system, adapted according to the platform: the UI can be executed either in a large screen (Figure 5a) or in a small screen (Figure 5c). Users of this system are provided with the same functionality on both versions of the UI: the task of consulting and modifying the temperature of a particular room.

We discuss now the adaptation applied on the UI versions of Figures 5a and 5c, with respect to the plasticity problem space axes. The axes of which the characteristics are unknown are not

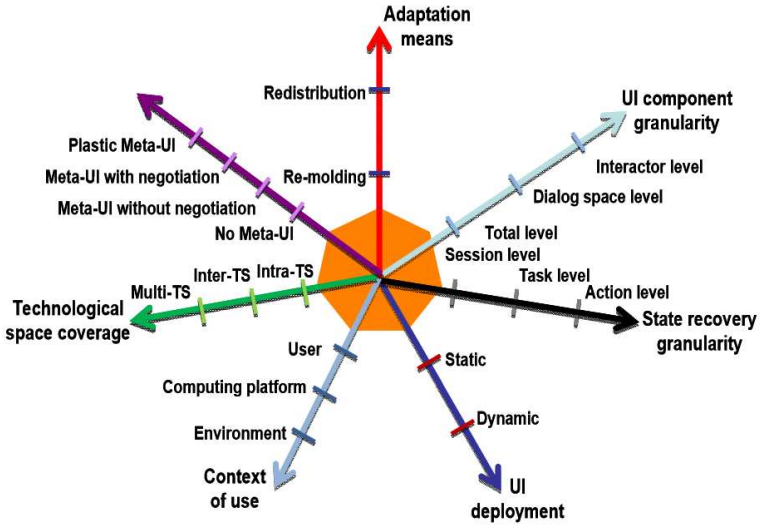
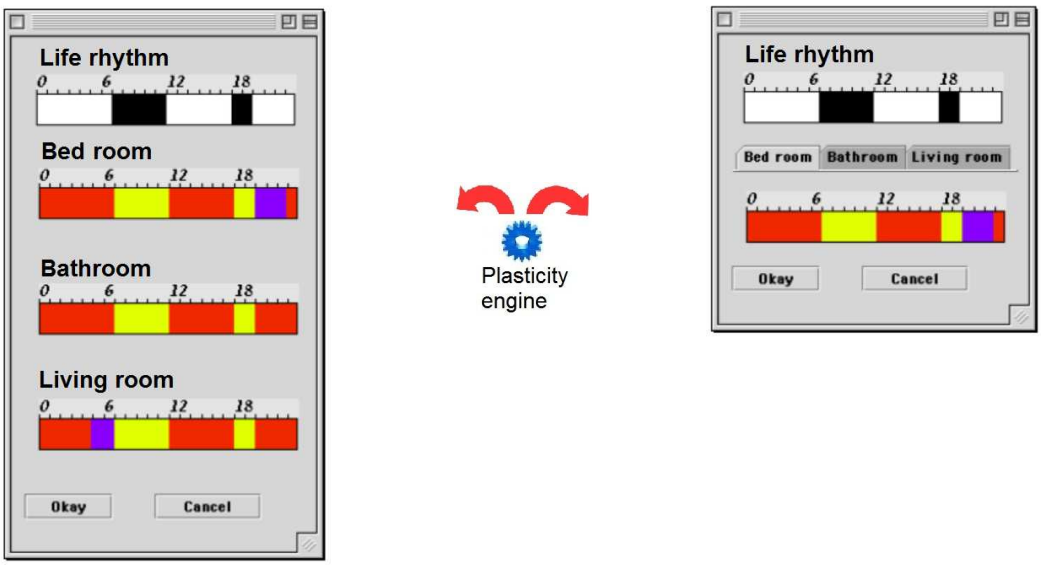


Figure 4: The problem space of plastic user interfaces [Vanderdonckt *et al.* 2008]

discussed (i.e., the *state recovery granularity*, the *UI deployment*, and the *technological space coverage*). Re-molding is the *adaptation means*, since the UI visual components are re-molded to adjust the size of the screen; the *UI component granularity* of the adaptation is defined at the interactor level, since the widget allowing a room temperature to be changed is re-molded; only changes in the platform are taken into account in the *context of use* of the UI: the system can be used, for instance, on large or small screens; and finally, this example does not contain *meta-UIs*.

In this adaptation, the appearance of the UI changed (i.e., in Figure 5a the temperature of all the rooms are available, in contrast to Figure 5c, in which one room at a time is available), as well as the UI behavior (i.e., in Figure 5c one additional action is required to the task of changing a room temperature: first to select the room). It can be the case that only the appearance of the UI changes, or only its behavior. Such decision can be made either at design-time or at runtime. At runtime, a *plasticity engine* (or *adaptation engine*) contains *transformation rules* (or *adaptation rules*) which can calculate a user interface adapted to its context of use. At design-time, several versions of the UI are created by the designer, one for each context of use. Then the plasticity engine chooses at runtime the most suitable one for the current context of use.

The adaptation may privilege some ergonomic criteria over others. The benefits plasticity brings to a UI justify to neglect some ergonomic criteria, as long as it does not deteriorate usability. For instance, one ergonomic criteria defined by [Bastien & Scapin 1993] is *minimal actions*, which is a matter of limiting as much as possible the steps users must go through to accomplish a task. The UI in Figure 5c undervalues this criterion by adding one more action to the user task, in order to privilege another ergonomic criteria called *information density* [Bastien & Scapin 1993], which concerns the users' workload from a perceptual and cognitive point of view with regard to the whole set of information presented to the users.



(a) Large screen: the temperature of the rooms are available at a glance

(b) The engine contains the UI adaptation rules

(c) Small screen: the temperature of a single room is displayed at a time

Figure 5: A home heating control system [Coutaz *et al.* 2000]

Plasticity provides users with different versions of a user interface. Since these versions diverge from each other at different levels, they may not be consistent with each other. Regarding the UI behavior, for instance, critical features that are expected to be provided on all UI versions may not be. And regarding the UI appearance, for instance, a given symbol may be displayed with different colors in different UI versions. The number of such inconsistencies may increase depending on the number (and complexity) of UIs provided by plasticity.

Although plasticity enhances UI capabilities, it adds complexity to the development of user interfaces. UIs are expected not only to provide correct, intuitive and non-ambiguous means for users to accomplish a goal in a reliable way, but also to be sensitive to changes in their context of use and to adapt themselves while preserving the aforementioned requirements. Furthermore, different versions of the UIs are expected to cope with all these requirements. This complexity is further increased when it comes to user interfaces of safety-critical systems.

1.4 Safety-Critical Systems

Safety-critical systems are systems in which a failure has severe consequences (e.g., death or injury to people, environmental harm, loss or damage to equipment). While in interactive systems in general bad user interfaces can be the source of frustrated, annoyed or intimidated users, causing waste of time, steeper learning curve, or even user rejection of new systems, bad UIs in safety-critical systems can be the source of deathly accidents/incidents. Examples of such systems are: transportation (e.g., avionics, trains), automotive (e.g., airbag systems, braking systems), health care (e.g., medication administration, radiation therapy machines), nuclear industry (i.e., nuclear reactor control systems, cooling systems), and even recreation

(e.g., amusement rides).

According to [Leveson 1995], risks have changed as society and the natural environment have changed. In the past, the greatest concerns were natural disasters. Today, industrialization has substituted nature-rooted disasters by man-made hazards. Some factors that may affect the emergence of risks are [Leveson 1995]:

- the appearance of new hazards due to advances in science and technology;
- the increasing complexity in the systems;
- the increasing exposure of society to hazard (e.g., passenger capacity in aircraft is increasing to satisfy economic concerns);
- the discovery and use of high-energy sources, which have increased the magnitude of the potential losses;
- the increasing automation of manual operations;
- the increasing centralization and scale of industrial production such as power plants; and
- the increasing pace of technological changes.

In particular, the increasing complexity in the systems is a reality in safety-critical systems. Complexity is a source of design errors: subtle faults can emerge during the integration phase. For instance, overly large redundancy-management software may lessen, rather than enhance, the overall reliability [Rushby & von Henke 1993].

This increasing complexity of systems is reflected in user interfaces. Several issues have been reported in the safety-critical domain due to bad user interfaces (e.g., in avionics [Degani & Heymann 2000], in radiation therapy machines [Turner 1993], in infusion pumps to deliver drugs in hospitals [Thimbleby 2010], etc.). UIs are now expected not only to provide correct, intuitive, non-ambiguous and adaptable means for users to accomplish a goal, but also to cope with safety requirements aiming to make sure that systems are reasonably safe before they enter into the market.

1.5 Nuclear-Plant Systems

Safety-critical systems are used, for instance, in nuclear plants. Nuclear power plants generally have large complex systems, which are redundant to recover whenever an emergency occurs. The environment of such systems is highly secure and protected, and only a restricted number of qualified users have access to the system functionalities. Therefore, the development and verification tools of these systems should be appropriately selected [Lutz 2000]. For instance, a simulated environment should be provided to each nuclear-plant unit, and systems should be tested in such environment before reaching the real plant [Yoshikawa 2005]. Besides, the nuclear-plant domain have specificities that distinguish it from other safety-critical domains:

1. Usually this kind of environment integrates several systems into sophisticated and complex equipments [Yoshikawa 2005]. Assessing the quality of such systems is hard, yet the verification of such systems is expected to be as exhaustive as possible.

2. The user interfaces of these systems are expected to provide high-quality support to operators in their daily activities and, more importantly, during incidents/accidents. In such conditions, operators must take action as instructed by the emergency operating procedures. It is known that the reasoning capabilities of humans deteriorate under stressful conditions [Niwa *et al.* 2001]: therefore, even though user interfaces of such systems are complex, they are expected to well support users in their activities. In addition, critical information should be constantly monitored and displayed: thus, critical user interfaces should always be visible, i.e., the display of another UI over these critical UIs is not authorized. A way to verify these constraints are needed.

3. Multidisciplinary users have access to system functionalities, i.e., chemists, physicians, managers, nuclear engineers, etc. Besides, operators pass through a significant number of hours of training before starting their activity [Commission *et al.* 1997]. Systems in this kind of environment should adapt to this variety of user profiles. In particular, the user interfaces of such systems should fulfill such variety while preserving usability.

4. Such systems should be conform to legal and regulatory requirements such as the IEC61513 international standard [IEC 2011], a standard for the nuclear industry under which a system is required to be developed. Therefore, a precise way to verify whether the system fulfill such requirements is needed.

These constraints should be taken into account by a methodology to assess the quality of such systems.

1.6 Research Question

Combining safety-critical systems with advances in human-computer interactive, such as Plasticity, leads to new problems. Given the large number of possible versions of UIs an adaptation engine can provide, it is time consuming and error prone to check the aforementioned requirements of plastic UIs by hand: some automation must be provided. In particular, plastic user interfaces in the context of safety-critical systems may have to preserve some critical functionalities in all contexts of use.

The severe consequences failures in safety-critical interactive systems call for a rigorous way to ensure the quality of such systems, a way which can handle in particular their complexity. In the context of nuclear plants, safety-critical systems are provided on conventional panels in the control rooms. Some production and monitoring systems are usually integrated into sophisticated and complex equipments. A rigorous way that is capable to explore the combination of these sources of complexity can help.

Besides, safety-critical systems can be used in industrial contexts, which make them large systems. A proposition to ensure quality of these systems should scale to large systems, in order to be used in industrial contexts.

Our research question is:

"How to improve quality of safety-critical interactive systems with plastic user interfaces, in a way which permits a rigorous verification of the system and is scalable for industrial applications?"

1.7 Quality of Interactive Systems

Several techniques to ensure quality of interactive systems in general exist, which can also be used to safety-critical systems: requirement validation, code review, static analysis, dynamic analysis (which can have four forms: simulation, testing, run-time analysis and log analysis), and formal verification [Garavel & Graf 2013]:

- *Requirement validation* aims to ensure that good-quality requirements are used to the next steps of the system development. This can be ensured in several ways, such as reviews by a panel of examiners or translation to other notations (e.g., informal requirements can be translated into semi-formal ones, which can reveal defects);
- *Code review* consist in submitting the code to a committee who searches for defects, enabling to catch flaws that are difficult to be caught for the design artifact author;
- *Static analysis* attempts at finding errors in design artifacts without executing them. It permits an automated review of the system artifact (i.e., the code, system diagrams, etc.) to ensure, for instance, that the artifacts are of sufficient quality to be reviewed;
- *Simulation* refers to the dynamic analysis of virtual design artifacts. It is commonly used to access the functional correctness of a system under design and to estimate its performance;
- *Testing* refers to the dynamic analysis of real design artifacts, after the system is developed, and before it is deployed. It is the most widely used analysis;
- *Run-time analysis* refers to the dynamic analysis of real design artifacts, after the system is developed and deployed, and during its execution. For instance, the memory of the design artifact is scrutinized to check for internal properties;
- *Log analysis* is a technique that seeks to make sense out of computer-generated records (also called log or audit trail records). It refers to the dynamic analysis of real design artifacts, after the system is developed and deployed, and after its execution; and finally,
- *Formal verification* relies, in whole or in part, on formal methods to the development of safe and secure systems. It refers to the dynamic analysis of virtual design artifacts, and before the system is developed and deployed.

Each one of these techniques have merits and shortcomings [Garavel & Graf 2013]. In this thesis, we mainly focus on two techniques: testing, since it is the most widely used one, even in safety-critical systems; and formal verification, since it is a suitable technique for safety-critical systems [Lutz 2000].

Testing a system consists in executing a runnable version of the system and observing whether the system is conform to the specifications or not. Testing can be either manually performed (by a human) or automatically performed (by testing softwares). In either way, the expected behavior of the system is described by means of test cases. Test cases are sequences of steps to be followed in order to check correctness of a given functionality of the system under test, and they can be created either manually or automatically, by model-based techniques.

Formal verification also relies on models of the system under analysis. It consists in the application of techniques that are strongly rooted in mathematics to reason over a model of the

system, providing a rigorous way to perform system verification. This permits, for instance, the simulation of the system, the early verification of properties, or the detection of inconsistencies in requirements.

However, relatively few case studies of formal methods to industrial systems were reported [Miller 2009]. Some causes are [Cofer 2012]:

1. *Usability*: formal notation and tools are unknown to developers;
2. *Cost*: a model that reproduces the behavior of the original system must be created, using a formal language. The creation / maintenance of this model is expensive;
3. *Fidelity*: there is no guarantees that the models really correspond to the system.

All these factors and others more have been studied, to reduce the gap between formal methods and industrial systems [Godefroid *et al.* 1998, Knight 1998, Hall 1999, Lutz 2000, Abrial 2006, Ameur *et al.* 2010, Newcombe *et al.* 2015]. New ways to reduce the gap between industrial systems and formal methods are also needed.

We present in this thesis an approach to assessing the quality of plastic interactive systems, in the context of nuclear power plants. The approach applies formal methods and is scalable to industrial systems. In order to reduce the gap between industrial systems and formal methods, the use of an intuitive formal specification language mitigates the *usability* and *cost* issues, and the connection of the formal specification with the modeled system mitigates the *fidelity* issue.

1.8 Context of the Thesis

This thesis was elaborated within the Connexion Project[1] (***Co**ntrôle Commande **N**ucléaire **N**umérique pour l'**ex**port et la rénovat**ion***), a R&D program to propose and validate an innovative architecture platform for control room systems of new generation nuclear power plants in France.

The project gathers the major actors in the nuclear power plant domain in France, including partners from academic research (CEA, INRIA, ENS Cachan, CNRS/CRAN, LIG, Telecom ParisTech), techno-providers of embedded software (Rolls-Royce Civil Nuclear, Corys TESS, Atos Worldgrid, Esterel Technologies, ALL4TEC, Predict), and AREVA, ALSTOM and EDF, groups specialists in energy, electricity, and electric power generation. The expertise brought by our laboratory (LIG) to the project is twofold: the experience in model driven engineering of user interfaces, specifically, plastic user interfaces (IIHM Research Group[2]), and the experience in the verification and validation (V&V) of systems (CONVECS Research project-team[3]). Based on this expertise, the main contributions of LIG to the project are the application of recent advances of plasticity in the context of the project (out of scope of this thesis), and a proposition of a V&V approach for plastic user interfaces in safety-critical systems (the scope of this thesis).

1.9 Outline of the Thesis

This thesis is structured in eight chapters, organized as follows:

[1] `http://www.cluster-connexion.fr/`
[2] `http://iihm.imag.fr/`
[3] `http://convecs.inria.fr/`

- Chapter 2 presents a representative list of propositions applying a variety of techniques and tools to improve the quality of systems (state of the art). Two main classes of approaches are identified to split the related work: approaches that verify whether the system follows a given specification and approaches that compare the system with another artifact to assess consistency. The related work is analyzed with respect to a set of criteria, and the limitations and needs for a new approach are highlighted.

- Chapter 3 describes the main nuclear power plant case study used to validate the approach proposed in this thesis. The improvements brought by plasticity to this case study are presented in the chapter. Finally, an industrial implementation to this case study is presented, to which our work is applied with different variants.

- Chapter 4 introduces our global approach to verifying interactive systems. With this goal, two formal techniques are integrated: model checking, for the verification of the system with respect to a given specification, and equivalence checking, to check for consistency of different versions of plastic user interfaces. The ideas common to both parts are presented in this chapter, as well as the rationale of the global approach unifying both parts.

- Chapter 5 details the first part of our approach: the verification of interactive systems with respect to a given specifications, and a connection to an industrial system. The case study presented in Chapter 3 is used to illustrate the approach, and a set of properties is verified over the system model. A connection with a real industrial system in the nuclear-plant domain is described.

- Chapter 6 details the second part of our approach: how interactive systems with plastic user interfaces are verified by our comparison method. User interfaces are classified according to their levels of similarity and several abstraction techniques are introduced. The proposed approach is also illustrated and validated in the case study presented in Chapter 3.

- Chapter 7 validates the approach by applying it to a second case study in the nuclear-plant domain, in which a system flaw is identified thanks to the verification of properties.

- Chapter 8 reviews the contributions of the thesis, gives concluding remarks and proposes possible directions for future work.

Chapter 2

State of the Art

Contents

2.1 Goals **13**
2.2 Modeling **14**
2.3 Model-based Testing **14**
2.4 Formal Verification **15**
2.5 Criteria to Analyze the State of the Art **17**
2.6 Verification of Properties **18**
2.6.1 Interactive Systems Modeled as a Composition of Parts 19
2.6.2 Interactive Systems Modeled Globally 37
2.6.3 Synthesis 51
2.7 Assessing Consistency **54**
2.7.1 System × System 54
2.7.2 System × User Manual 59
2.7.3 Synthesis 66
2.8 Summary **68**

We recall that the goal of this thesis is to propose an approach to verifying plastic interactive systems, in the context of safety-critical domains, which can be used in industrial systems.

2.1 Goals

This chapter gives an overview of several approaches to assessing the quality of interactive systems. We structure them in two categories:

- Approaches that verify properties over the system model, which guarantee that the system copes with a certain level of quality before it is deployed. We are specially interested in approaches that are applied to industrial systems in the safety-critical domain.

- Approaches that checks for consistency of the system with another artifact. These approaches are particularly important in the context of this thesis, because they could give ideas about ways of comparing different versions of plastic user interfaces.

We give an overview of these approaches and compare them according to a set of criteria. A table summarizing the presented approaches is provided at the end of each category. Model-based testing approaches are briefly described, since they are also largely used to assess the quality of safety-critical interactive systems. We start by describing the modeling technique.

2.2 Modeling

Modeling is a well-accepted engineering technique. It is a central part of all the activities that lead up to the deployment of good softwares [Booch *et al.* 2005]. A model is a representation, often in mathematical terms, of what is perceived as relevant characteristics of the object or system under study [Peterson 1981].

Models can be used for a variety of purposes [Booch *et al.* 2005, Turchin & skiĭ 2006]:

- to communicate the desired structure and behavior of the system;
- to visualize and control the system's architecture;
- to better understand the system under development;
- to help users to visualize the final product;
- to derivate specific predictions from theory that can be tested with data;
- to help the identification of relationships between various components of the system.

The use of models is an effective support for the development of interactive systems. There are limits to the human ability to understand complexity. Modeling helps designers to break complex applications into small manageable parts [Navarre *et al.* 2005]. Models of interactive systems describe the behavior of the input and output devices used by the systems and the interactions between the user and the systems. Model construction, thus, often requires making simplified assumptions. The extent to which a model helps in the development of human understanding is the basis for deciding how good the model is [Hallinger *et al.* 2000].

System models can be subject to a variety of techniques to assess the quality of interactive systems. For instance, model-based testing analyzes the real system, before its deployment [Garavel & Graf 2013].

2.3 Model-based Testing

Several authors use model-based testing to assess the quality of interactive systems. For instance, in [Tsai *et al.* 2000] UI navigation of interactive systems is modeled by a directed graph model called SNet, in which a node represents a UI and a link between two nodes represents a navigation path from one UI to another. Links have pre-conditions ans post-conditions. The pre-condition is the current status of the UI when a particular event is triggered. The post-condition is the expected status of the target UI after it is displayed. Based on this model, test cases can be generated systematically from various navigation paths and scenarios based on a given coverage criteria. While this work proposes a way to model UI navigation in order to generate test cases, user interface functionalities and appearance are not covered, which limits the approach.

Also based on system models, but this time on UML Statechart models, the authors of [Hjort *et al.* 2009] propose the usage of UPPAAL and UPPAAL-CORA to generate test cases from UI state machines. Alternatively, task trees are used in [Madani & Parissis 2009]; more precisely, task trees enriched with *operational profiles* (i.e., probabilities assigned to the user actions). Such enriched task trees are translated into finite-state machines (FSM), which are then used to generate the test data using the Lutess testing tool. The probabilities injected into the

FSM can be used to guide the test generation. Also using task models, the authors of [Bin & Anbao 2012] propose a method to formally integrate use case models and task models, so as to create a composite FSM that are used afterwards to generate test cases that capture more complete and detailed user interactions. The strength of these propositions is that their approaches are integrated to models currently used by developers (i.e., UML Statemachines, task models and use case models), which mitigate the barriers of adopting a new technique. However, supporting the communication from one specific model to another specific model (e.g., from UML Statemachines to UPPAAL) limits the approaches to those models.

In order to improve testing aspects such as effectiveness, coverage, etc., a number of techniques are proposed. For example, the authors of [Lu & Huang 2012, Huang & Lu 2012] propose the use of ant colony algorithm to dynamically generate feasible test cases from UIs modeled by event-flow graphs. However, in the UI models, only navigation and functionalities are covered, appearance is not modeled. Alternatively, the authors of [Bhasin *et al.* 2013] propose an *orthogonal testing* technique, which is based on genetic algorithms to efficiently reduce the number of generated test cases and improve effectiveness. However, only web-pages can be analyzed by this approach. Last but not least, *mutation testing* is used in [Alsmadi 2013] to improve UI test coverage. It consists in modifying the program source code in small ways. Each mutated version is called a mutant, and tests suites are measured by the capacity to detect and reject such mutants. However, the code source of the system under test should be available. While these approaches improve the effectiveness of testing, which by consequence improves the quality of interactive systems, their limitations avoid them to be used in a larger range of interactive systems.

The test-based work presented here is not an exhaustive list. Other approaches have been proposed in the literature to assess the quality of interactive systems by model-based testing (e.g., [Mariani *et al.* 2011, Memon *et al.* 2003, Nguyen *et al.* 2010, Nguyen *et al.* 2014, White & Almezen 2000]). Test cases are an intuitive way to express specifications: they do not require any additional knowledge in formal notations. However, they are usually used to asses the system functionalities, while in safety-critical systems there is also the need to assess the safety requirements and ergonomic properties, which is provided by approaches based on formal methods.

2.4 Formal Verification

Models are expressed in some modeling language, and depending on the nature of the language, can be *informal*, *semi-formal*, and *formal* [Garavel & Graf 2013]:

- *informal* models are expressed using natural language or loose diagrams, charts, tables, etc. They are genuinely ambiguous, heavily rely on human intuition, and no software tool can analyze them objectively;

- *semi-formal* models are expressed in a modeling language that has a precisely-defined syntax, conveys some intuitive meaning, but has no formal (i.e., mathematical, self-contained, unambiguous) semantics. Examples of semi-formal specification languages are class diagrams, data flow diagrams, decision trees, entity relationship models, object models, pseudocode, state diagrams, etc.;

- *formal* models are written in a language that has a precisely defined syntax and a formal semantics. Examples of formal specification languages are algebraic data types, synchronous languages, process calculi, input/output automata, etc.

Formal models are system descriptions translated into a very precise language which, unlike natural human languages, does not allow for any double meanings. Once we have framed a system in a formal model, we can deduce precisely what are the consequences of the assumptions we made – no more, no less [Turchin & skiĭ 2006].

A formal model of a system is useful in several aspects. A non-exhaustive list of usages of a formal model follows: the simulation of the system behavior, an overview of the system state space and behavior, the generation of test cases, the reasoning over such system representation in several ways, for instance, by performing formal verification of requirements written as properties, etc.

Formal verification applies techniques that are strongly rooted in mathematics. Errors detected by formal verification may indicate system defects. In this case, the system is modified to correct these. This modified system can then be modeled and analyzed again. The verification results provide a feedback to the whole approach, by revisiting the specifications, by refining the formal model, and by assessing the quality of the real system. This cycle is repeated until the analysis reveals no problems anymore. Examples of formal verification techniques are model checking, equivalence checking, and theorem proving.

Theorem proving is a deductive approach for the verification of systems [Boyer & Moore 1983]. Proofs are performed in the traditional mathematical style, using some formal deductive system. Both the system under verification and the specifications against which the systems are verified are modeled as logic formulas, and the satisfaction relation between them is proved as a theorem using the deductive proof calculus. Proofs progress by transforming a set of premises into a desired conclusion, using the axioms and the deduction rules. The process is not fully automated: user guidance is needed regarding the proof strategy to follow. Theorem proving techniques can handle infinite-state systems. They are applied not only to models of the system, but also to source code [Ameur *et al.* 2010].

Model checking (Figure 6) permits to verify whether a model satisfies a set of requirements, which are specified as properties. A property is a general statement expressing an expected behavior of the system. In model checking, a formal model of the system under analysis is needed to be created, which is afterwards represented as a finite-state machine (FSM). This FSM is then subject to exhaustive analysis of its entire state space to determine whether the properties hold or not. The analysis is fully automated and the validity of a property is always decidable [Clarke *et al.* 1983]. Even though it is easier for a human being to express properties in natural language, it can result in imprecise, unclear and ambiguous properties, which are undesired characteristics in formal methods. Expected properties should, thus, be also formalized by means of a temporal logic. The analysis is mainly supported by the generation of *counter-examples* when a property is not satisfied. A counter-example can be a set of steps that when followed, by interacting with the system, leads to a state in which the property is false. The results of the analysis permits a refinement of the modeled system.

Since the introduction of model checking in the early 80s, it has advanced significantly. The development of algorithmic techniques (e.g., partial-order reduction, compositional verification, etc.) and data structures (e.g., binary decision diagrams) allows for automatic and exhaustive analysis of finite-state models with several thousands of state variables [Ameur *et al.* 2010]. For

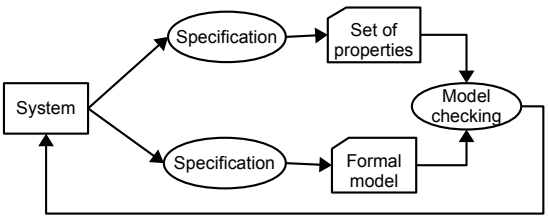


Figure 6: Model checking

this reason, model checking has been used in the past years to verify interactive systems in safety-critical systems of several domains, such as avionics [Degani & Heymann 2002], radiation therapy [Turner 1993], healthcare [Thimbleby 2010], etc.

Rather than verifying the satisfiability of properties, *equivalence checking* (Figure 7) permits to formally prove whether two representations of the system exhibit exactly the same behavior or not. In order to verify whether two systems are equivalent or not, a model of each system should also be created, and then both models are compared in the light of a given equivalence relation. Several equivalence relations are available in the literature (e.g., *strong bisimulation* [Park 1981] and *branching bisimulation* [van Glabbeek & Weijland 1996]). Which relation to choose depends on the level of details of the model and the verification goals. The results of the analysis also permits a refinement of the modeled systems.

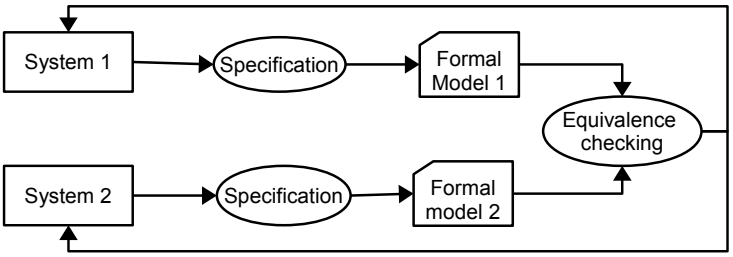


Figure 7: Equivalence checking

These are the main formal verification techniques applied in the following approaches to verifying interactive systems. In the sequel, we identify the criteria we will use to compare the proposed approaches in the state of the art.

2.5 Criteria to Analyze the State of the Art

In the sequel, we will focus on formal verification for interactive systems. Each approach is presented with respect to the following structure: after a brief *introduction* of the approach, we unfold it step by step, identifying which *language/formalism* is used to model the interactive system, followed by an *example* of a system modeled with the approach. Then, the *properties* the approach verifies are listed, together with the *language/formalism* used to formalize them, the

verification technique employed and whether the approach is *tool supported*. Then, an analysis is performed according to the following criteria:

1. *Modeling coverage*: the verification of the system relies on the system model. For this reason, the model coverage should be large enough for the verification to be useful. We analyze whether the studied approach covers aspects of the *functional core* and the *user interfaces* or not. The functional core of a system implements the domain-dependent concepts and functions, and the user interfaces implement the look and feel of the interactive system [Bass *et al.* 1991]. In this thesis, we call a "user interface" (UI) a group of graphical components which are displayed at the same time on a screen, and permit users to interact with the system. Used in plural form, "user interfaces" can designate either the set of user interfaces of a specific system or the user interfaces of any system. In addition, we also analyze if aspects of the *users* are included in the model, in order to take into account user behaviors.

2. *Verification of plastic user interfaces*: plasticity enhances regular user interfaces. A verification approach is expected to cope with such enhancements. In case the model covers user interfaces, we analyze whether the verification approach is applied to plastic user interfaces or not.

3. *Kinds of properties*: one kind of verification that can be performed over a system model is property verification. In the context of safety-critical systems, we believe that the focus should be directed both to safety requirements (to ensure that the system is correct) and to usability requirements (to ensure that the system prevents users from making mistakes) of such systems. For each author, we analyze the kinds of properties that can be verified using the approach.

4. *Application to the nuclear plant domain*: since the scope of this thesis is the nuclear plant domain, we analyze whether each approach is applied to this domain or not. This domain has its own specificities (cf. Section 1.5 on page 7) and applications of formal verification to this domain is needed.

5. *Scalability*: since we are interested in approaches that can be applied to industrial applications, we investigate the scalability of the studied approaches.

2.6 Verification of Properties

Interactive systems are described by a specification. A specification is a general statement about the behavior that is necessary to be satisfied by the system. It can be represented in several ways, such as test cases, safety requirements, desired properties, etc. When used for verification, the goal is to give insights about whether the system satisfies the specifications or not.

When the specifications are expressed as properties, the identified properties depend on the verification goals. For instance, properties can be extracted from the system requirements document, to guarantee that the system was developed according to the documentation. Alternatively, ergonomic frameworks aiming at guiding the development of good-quality interactive systems are available, such as [Abowd *et al.* 1992, Bastien & Scapin 1993, Vanderdonckt 1994]: ergonomic properties can be extracted from such frameworks. Last but not least, safety-critical

systems are constrained to several regulations to ensure their safety before they enter in the market. In this case, properties can be extracted from the safety requirement documents.

Several authors propose different categories of properties. For instance, three kinds of properties are identified in [Campos & Harrison 1997]: *visibility* properties, which concern the users' perception, i.e., what is shown on the user interface and how it is shown; *reachability* properties, which concern the user interfaces, and deal with what can be done at the user interface and how it can be done (in the users' perspective); and *reliability* properties, which concern the underlying system, i.e., the behavior of the interactive system. Reliability properties do not directly analyze the interaction between users and the user interfaces, but how the UIs and the underlying system collaborate. Alternatively, [Yamine *et al.* 2005] defines *validation* properties as the behavior of the UI expected, or desired, by the user (e.g., completeness, flexibility, task achievement, and so on); and *robustness* properties concern the successful achievement and assessment of the user goals when interacting with the system.

Since the scope of our work is safety-critical systems, we believe that the focus should be directed both to safety requirements (to ensure that the system follows the regulations) and to usability requirements (to ensure that the system prevents users from making mistakes). In this chapter, we propose to analyze whether existing approaches verify or not two kinds of properties:

- *usability* properties, which express whether the system follows ergonomic properties to ensure a good usability. These are generic properties that can be applied to any interactive system; They include *visibility* properties [Campos & Harrison 1997] and cover also other aspects, allowing more than the desirable visible effects on the UIs to be verified. They also include *robustness* properties [Yamine *et al.* 2005], i.e., the successful achievement and assessment of the user goals when interacting with the system may relate with usability aspects.

- *functional* properties, which express whether the system follows the requirements specifying its expected behavior, defined in requirement documents and/or safety requirement documents. These are specific properties that are identified and applied only to the modeled system. They include *reachability* properties as it is defined in [Campos & Harrison 1997]: what can be done at the user interface and how it can be done are defined in the requirements, and functional properties also aim at verifying that. They also include *reliability* properties [Campos & Harrison 1997]: they can verify how the UIs and the underlying system collaborate, but also only the underlying system. Finally, they also include *validation* properties [Yamine *et al.* 2005]: ensuring that the system follows the requirements also requires the verification of validation properties that characterize the UI behavior expected by a user.

The following subsections detail a representative list of approaches to verifying properties over a system model. They are structured in two main groups: (1) formal approaches that model interactive systems as a composition of smaller parts; and (2) formal approaches that model interactive systems globally.

2.6.1 Interactive Systems Modeled as a Composition of Parts

In this section we present several approaches that use formal notations to model interactive systems as a composition of smaller parts named agents, interactors, components, and objects,

in order to perform verification afterwards. Such approaches are suitable to model large and complex systems, since systems are subdivided in smaller and manageable parts.

a) Abowd *et al.* (USA, 1991–1995)

In [Abowd 1991], a framework for the formal description of users, systems and user interfaces is proposed. The interactive system is modeled as a collection of *agents*, which closely relates this model to other multi-agent architecture models, such as PAC [Coutaz 1987]. But, contrary to PAC, in [Abowd 1991] the agents are formally described.

The language to describe the agents borrows notations from several formal languages. For the internal description of the agent, the language is similar to a model-oriented notation such as Z or VDM and for the external description, it is similar to process calculus, such as CSP or CCS. The verification of specifications written in these notations can be tool supported (for instance, ProZ for Z specifications). However, such a verification is not described in [Abowd 1991]. An example of a button modeled in this language is illustrated in Figure 8.

This formalization of agents was the basis for the definition of a framework of desired properties to be verified. A catalog of *usability* and *functional* properties is presented, each discussed in terms of the interaction framework and/or the agent model. For instance, the informal definition of *predictability* is that "a predictable system is one in which it would be possible for the user to internalize a model that would be of benefit to future interactions". In this agent model framework, the properties are formally defined by means of mathematical templates which can be instantiated to the studied interactive system.

Alternatively, another approach to property verification is proposed in [Wang & Abowd 1994, Abowd *et al.* 1995], in which interactive systems are described by means of a tabular interface using the Action Simulator tool. Action Simulator is a tool for describing PPS (*Propositional Production System*) specifications. A propositional production system makes it easier to enumerate states and state transitions by factoring the state space into fields of mutually exclusive conditions [Abowd *et al.* 1995].

The dialog model is a specification of the constraints on the behavior of a user interface [Abowd *et al.* 1995]. In PPS, the dialog is specified as a number of production rules using pre- and post-conditions. Action Simulator permits such PPS specification to be represented in a tabular format, in which the columns are the system states, and the production rules are expressed at the crossings of lines and columns.

A translation from such a tabular specification of the interactive system to SMV input language is described in [Wang & Abowd 1994]. The CTL temporal language is used to formalize the properties, allowing the following *usability* properties to be verified:

- *reversibility* [Abowd *et al.* 1995]: can the effect of a given action be reversed in a single action?
- *deadlock freedom* [Abowd *et al.* 1995]: from an initial state, is it true that the dialog will never get into a state in which no actions can be taken?
- *livelock freedom* [Wang & Abowd 1994]: from any state of a given state set, can the user escape from that state set?
- *undo within N steps* [Wang & Abowd 1994]: from any state of a given state set, if the next step leads the system out of the state set, can a user go back to the given state set

```
agent button
internal
    types
        BStatus ::= up | down
    attributes
        bstatus : BStatus
    initially
        bstatus = up
    operations
    press()
        changes (bstatus)
        pre bstatus = up
        post bstatus' = down
    release()
        changes (bstatus)
        pre bstatus = down
        post bstatus' = up
    inform(bs : BStatus)
        changes ()
        pre bs = bstatus
communication
    inputs buttin : press(), release()
    outputs buttout : inform(bs : BStatus)
external
    μX • buttin?x → buttout!inform(b) → X
endagent button
```

Figure 8: A button modeled as an agent [Abowd 1991]

within N steps?

In addition, the following *functional* properties can be verified:

- *rule set connectedness* [Abowd *et al.* 1995]: from an initial state, can an action be enabled?
- *state avoidability* [Wang & Abowd 1994]: can a user go from one state to another without entering some undesired state?
- *accessibility* [Wang & Abowd 1994]: from any reachable state, can the user find some way to reach some critical state set (such as the help system)?
- *accessibility within N steps* [Wang & Abowd 1994]: from any reachable state, no matter which enabled action a user is going to take, can the user find some way to reach some critical state set (such as the help system) within N steps?
- *event constraint* [Wang & Abowd 1994]: does the dialog model ensure/prohibit a particular user action for a given state set?
- *feature assurance* [Wang & Abowd 1994]: does the dialog model guarantee a desired feature in a given state set?

- *weak task completeness* [Abowd *et al.* 1995]: can a user find some way to accomplish a goal from initialization?
- *strong task completeness* [Abowd *et al.* 1995]: does the dialog model ensure that a user can always accomplish a goal?
- *state inevitability* [Abowd *et al.* 1995]: from any state in the dialog, will the model always allow the user to get to some critical state?
- *weak task connectedness* [Abowd *et al.* 1995]: from any state, can the user find some way of getting to a goal state?
- *strong task connectedness* [Abowd *et al.* 1995]: from any state, can the user find some way to get to a goal state via a particular action?

The automatic translation of the tabular format of the system states into the SMV input language is an advantage of the approach, since it allows model checking of properties to be performed. The tabular format of the system states and the actions that trigger state changes provides a reasonable compact representation in a comprehensible form. However, it looks like the approach does not scales well to larger specifications, unless an alternative way to store a large sparse matrix is provided. The approach covers the modeling of the functional core and the UIs, and to some extent the modeling of the users, by describing the user actions that "fire" the system state changes. However, no application to safety-critical systems is reported.

b) Paterno *et al.* (Italy, 1990–2003)

Interactive systems can be formally described as a composition of *interactors* [Faconti & Paternó 1990]. Interactors are more concrete than the agent model previously described, in that they introduce more structure to the specification by describing an interactive system as a composition of independent entities [Markopoulos 1997]. Interactors have a state, they receive input events, send output events, and can be connected one to another. Interactors have a reactive behavior and can operate in parallel [Yamine *et al.* 2005]. The external communication of interactors are formally defined, as well as their internal behavior. Notations to describe interactors cover the specification of the state, the behavior and the communication capabilities of the interactor [Campos & Harrison 1998].

The interactors of CNUCE [Paternò & Faconti 1992] provide a communication means between the user and the system (Figure 9). Data manipulated by the interactors can be sent and received through events in both directions: towards the system and towards the user [Paternó 1994], which are both abstracted in the model by a description of the possible system and user actions.

A CNUCE interactor is described by four sub-components:

- *collection*: represents the external appearance of the interactor in an abstract way;
- *feedback*: outputs to the user the data generated inside the interactor;
- *measure*: receives data from the user and manipulates them before sending to the *feedback* and *control* sub-components;
- *control*: sends the data produced inside the interactor to other interactors or to the system.

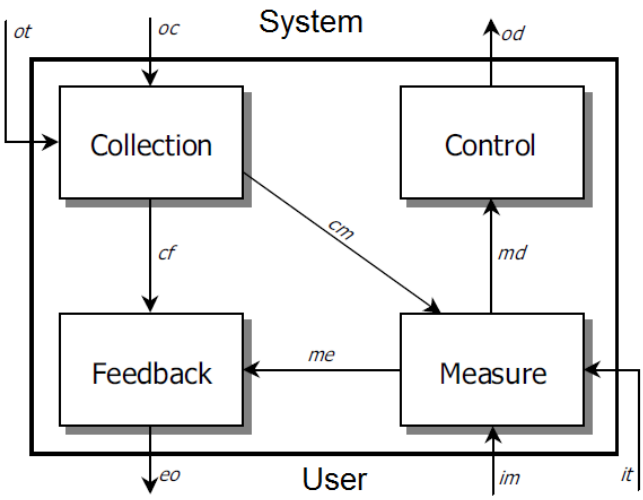


Figure 9: Architecture of a CNUCE interactor, adapted from [Paternò & Faconti 1992]

Interactors can be composed in the following way: the *feedback* sub-component of one interactor is connected to the *collection* sub-component of another interactor, and the *control* sub-component of one interactor to the *measure* sub-component of the other one.

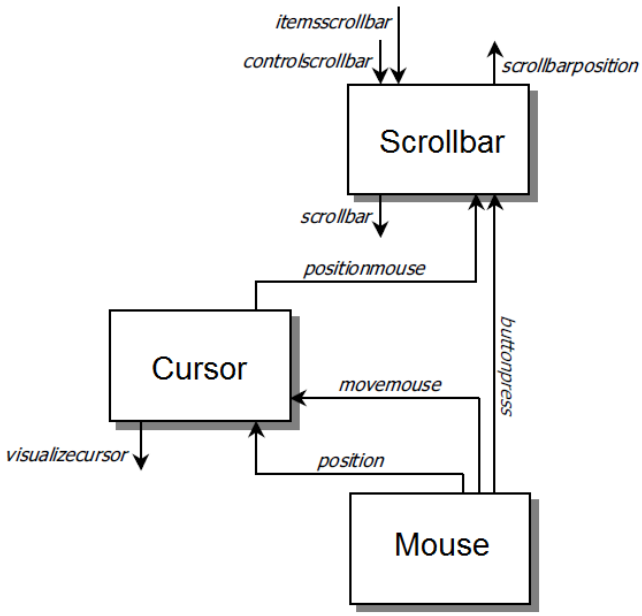


Figure 10: Scrollbar modeling with three CNUCE interactors: mouse, cursor and scrollbar. Adapted from [Paternò & Faconti 1992]

Figure 10 illustrates an example of a scrollbar modeled with three CNUCE interactors: *mouse*, *cursor* and *scrollbar*. In this example, the interactors are seen as black boxes, i.e., the four sub-components of each interactor are hidden. A flow of data passes through the interactors: the *mouse* interactor sends to the *cursor* its current position and if it has moved. The *cursor* handles the display of the cursor (`visualizecursor` arrow). When a mouse button is pressed (`buttonpress` arrow), the *scrollbar* interactor recovers the cursor position (through `positionmouse` arrow) and displays the scrollbar new position (through `scrollbarposition` arrow). The CNUCE interactors are specified using LOTOS [ISO/IEC 1989]. This formal model of the interactive system can be afterwards used for verification.

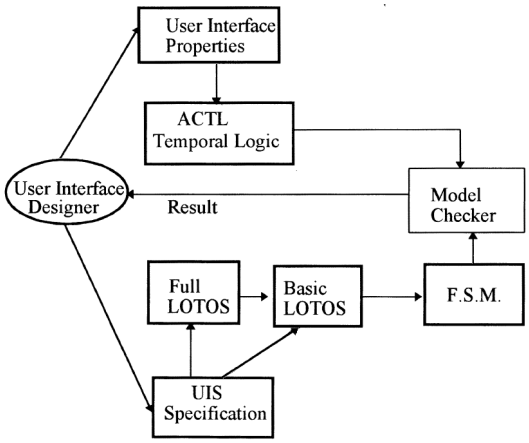


Figure 11: The TLIM approach [Paternó 1997]

The verification approach proposed in [Paternó 1997] is called TLIM (*Tasks,* LOTOS*, Interactors Modeling*), and is depicted in Figure 11. In this approach, the user interfaces of the interactive system are represented by a CTT (*Concur Task Trees*) task model [Paternò *et al.* 1997] (i.e., "UIS Specification" in Figure 11), built using the CTTE tool [Mori *et al.* 2002]. This task model is later used to automatically generate [Paternò & Santoro 2003] the Full LOTOS specifications of the interactors. Such LOTOS models are then transformed into a Basic LOTOS specification using the LITE tool, in order to remove information on the data types. This new UI LOTOS model is then translated into a finite-state machine (FSM) of the model, which are afterwards used to verify properties. A set of properties is identified from the user interfaces and formalized using an action-based temporal logic called ACTL, a kind of branching-time temporal logic which allows one to reason about the actions a system can perform [Paternó 1997]. The properties are verified by model checking, using the CADP model checker [Garavel *et al.* 2013], and the verification results are used to refine the user interfaces.

This approach has been used to verify the following *usability* properties:

- *visibility* [Paternó & Mezzanotte 1994]: each user action is associated with a modification of the presentation of the user interface to give feedback on the user input;
- *continuous feedback* [Paternó & Mezzanotte 1994]: this property is stronger than *visibility*: besides requiring a feedback associated with all possible user actions, this has to occur before any new user action is performed;

- *reversibility* [Paternó & Mezzanotte 1994]: this property is a generalization of the *undo* concept. It means that users can perform part of the actions needed to fulfill a task and then perform them again, if necessary, before the task is completed in order to modify its result;

- *existence of messages explaining user errors* [Paternó & Mezzanotte 1994]: whenever there is a specific error event, a help window will appear.

In addition, the following *functional* property can be verified:

- *reachability* [Paternó & Mezzanotte 1994]: this property verifies that a user interaction can generate an effect on a specific part of the user interface.

Besides, the approach also permits other *functional* properties expressing the expected behavior of specific case studies to be verified [Paterno 1993, Paternò & Mezzanotte 1996, Paternò & Santoro 2001].

Despite the fact that the approach covers mainly the modeling of user interfaces, a mathematical framework is provided to illustrate how to model the user and the functional core too [Paternó 1994].

The approach has been applied to several case studies of safety-critical systems in the avionics domain [Paternó & Mezzanotte 1994, Paternò & Mezzanotte 1996, Paternó 1997, Paternò & Santoro 2001, Navarre *et al.* 2001, Paternò & Santoro 2003]. These examples show that the approach scales well to real-life applications. Large formal specifications are obtained, which describe the behavior of the system, permitting meaningful properties to be verified. However, no case study applied this approach to the nuclear power plant domain. For instance, the multidisciplinary users of nuclear-plant systems are not considered by this approach.

A practical limitation of this approach is that the designer needs an in-depth understanding of both the LOTOS specification of the interactive system and the temporal logic formulation of the properties [Markopoulos *et al.* 1998]. Besides, the approach does not cover the modeling and verification of plastic user interfaces.

c) Markopoulos *et al.* (England, 1995–1998)

ADC (*Abstraction-Display-Controller*) [Markopoulos 1995] is an interactor model that also uses LOTOS to specify the interactive system (specifically, the user interfaces). In addition, the expression of properties is facilitated by templates.

An ADC interactor is composed of two units (Figure 12): an *abstraction and display unit* (ADU) and a *controller unit* (CU). The ADU is a component that receives, modifies, and sends the interactor data through *gates* named `ainp`, `aout`, `dout` and `dinp` (Figure 12). The CU component enables/disables the interaction on these gates.

The interactor handles two types of data: *display* data, which come (and are sent to) either directly from the UI or indirectly through other interactors, and *abstraction* data, which are sustained by the interactor to provide input to the application or to other interactors [Markopoulos *et al.* 1998].

A UI is modeled as a composition of ADC interactors. For example, Figure 13 shows the modeling of a scrollable list. Two slider interactors allow the displayed portion of the list to be scrolled in two dimensions. Each slider receives interactions and displays coordinates from

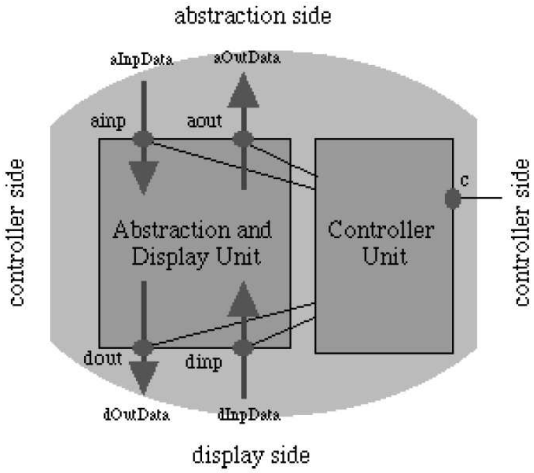


Figure 12: The ADC interactor [Markopoulos *et al.* 1998]

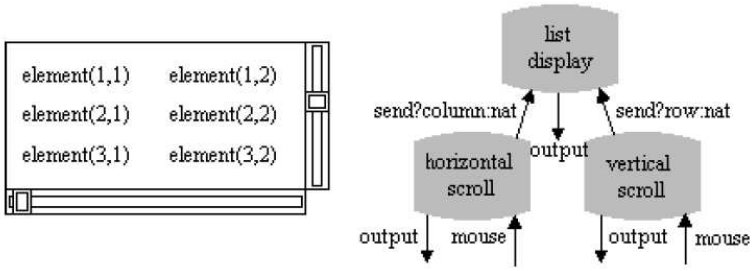


Figure 13: A scrollable list as a composition of ADC interactors [Markopoulos *et al.* 1998]

its display side, and interprets its input to instruct the list interactor to scroll [Markopoulos *et al.* 1998].

Once formalized in LOTOS, the ADC interactors can be used to perform formal verification of *usability* properties using model checking. The properties to be verified over the formal model are specified in the ACTL temporal logic. For example, the following properties can be verified:

- *determinism* [Markopoulos 1997]: a user action, in a given context, has only one possible outcome;
- *restartability* [Markopoulos 1995]: a command sequence is restartable if it is possible to extend it so that it returns to the initial state;
- *undoability* [Markopoulos 1995]: any command followed by *undo* should leave the system in the same state as before the command (single step *undo*);
- *eventual feedback* [Markopoulos *et al.* 1998]: a user-input action shall eventually generate a feedback.

In addition, the following *functional* properties can be verified:

- *completeness* [Markopoulos 1997]: the specification has specified all intended and plausible interactions of the user with the interface;
- *reachability* [Markopoulos 1997]: it qualifies the possibility and ease of reaching a target state, or a set of states, from an initial state, or a set of states.

In this approach, the CADP [Garavel *et al.* 2013] toolbox is used to verify properties by model checking [Markopoulos *et al.* 1996]. Specific tools to support the formal specification of ADC interactors are not provided [Markopoulos *et al.* 1998].

The ADC approach concerns mostly the formal representation of the interactor model. Regarding the coverage of the model, the focus is to provide an architectural model for user interface software. The functional core and the user modeling are not covered. Besides, no case study applying the approach to the verification of critical systems is reported. In fact, the approach is applied to several small scale examples [Markopoulos 1995] and to a case study on a graphical interface of Simple Player for playing movies [Markopoulos *et al.* 1996], which makes it difficult to measure whether it can scale up to realistic applications or not.

d) Duke and Harrison *et al.* (England, 1993–1995)

Another interactor model is proposed by the University of York [Duke & Harrison 1995] to represent interactive systems. Compared to the CNUCE interactor model, the main enhancement brought by the interactors of York is an explicit representation of the state of the interactor.

The York interactor (Figure 14) has an internal state and a rendering function (i.e., `rho` in Figure 14) that provides the environment with a perceivable representation (`P`) of the interactor internal state. The interactor communicates with the environment by means of *events*. Two kinds of events are modeled:

- *stimuli* events: they come from either the user or the environment, and modify the internal state of the interactor. Such state changes are then reflected to the external presentation through the rendering function;
- *response* events: they are events generated by the interactor and sent to the user or to the environment.

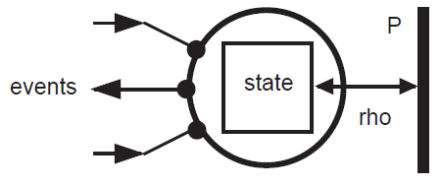


Figure 14: The York interactor [Harrison & Duke 1995]

Figure 15 illustrates an example of an icon modeled with a York interactor. It has two possible internal states: active or inactive, which are modified according to two stimuli events ON and OFF. Each interactor state change generates a response event called ALERT, and triggers the rendering function `rho`, which returns the corresponding perceivable representation among the space of possible renderings `P`.

The York interactor model defines an interactive system by the composition of a set of interactors acting in parallel, each of which representing a part of the system, some entity in the operating environment, or some aspect of the behavior of the user [Fields *et al.* 1995b].

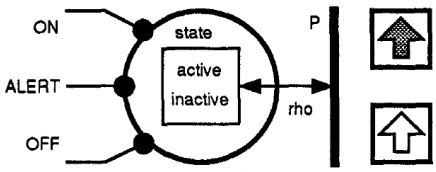


Figure 15: An icon modeled with a York interactor [Duke & Harrison 1993]

York interactors are described using the Z notation [Spivey 1989]. This notation facilitates the modeling of the state and operations of a system, by specifying it as a partially ordered sets of events in first-order logic [Duke & Harrison 1993].

This approach permits *usability* properties to be verified. Properties are also specified in first-order logic formulas. Unlike the previous approaches, the York interactor model uses *theorem proving* as formal verification technique. Examples of properties that can be verified are [Duke & Harrison 1995]:

- *honesty*: the effects of a command are intermediately made visible to the user;
- *weak reachability*: it is possible to reach any state through some interaction;
- *strong reachability*: each state can be reached after any interaction `p`;
- *restartability*: any interaction `p` is a prefix of another `q` such that `q` can achieve any of the states that `p` initially achieves.

The York interactor model served as a basis to other interaction frameworks [d'Ausbourg *et al.* 1998, Campos & Harrison 2001], some of them used to reason over safety-critical systems [Sousa *et al.* 2014]. Besides Z, this interactor model is also modeled with other formalisms, such as modal action logic (MAL) [Duke *et al.* 1995, Duke *et al.* 1999], VDM [Fields *et al.* 1995a, Harrison *et al.* 1996], and structured MAL and PVS (*Prototype Verification System*) [Campos 1999].

The York interactor model provides an abstract framework for structuring the description of interactive systems in terms of layers. It encapsulates two specific system layers: the state and the display [Harrison & Duke 1995], thus covering both the functional core and the UIs in the modeling. The approach is applied to a case study in a safety-critical system, an aircraft's fuel system [Fields *et al.* 1995b] in which the pilot's behavior is modeled, thus showing that the approach also covers the modeling of users. No further case studies applying the approach were found in the literature, which makes it difficult to tell whether the approach scales up to larger interactive systems or not. Besides, no application was found in the nuclear plant domain.

e) Campos *et al.* (Portugal, 1997–2015)

The York interactor model is the basis of the work proposed in [Campos 1999]. Here, Campos chooses MAL (*Modal Action Logic*) language to implement the York interactor model, since MAL's structure facilitates the modeling of the interactor behavior.

In this work, Campos conducts a deep investigation of two different techniques to verify interactive systems, namely model checking and theorem proving, the goal being to increase the capabilities of reasoning about specifications of interactive systems. Campos describes how to specify interactors using the PVS language and applies theorem proving to verify properties that mainly express the system side of the design (*functional* properties).

The approach is applied to several case studies. An application of both model checking and theorem proving to a common case study is described. Campos gives the following conclusions regarding theorem proving [Campos 1999]:

- Theorem provers are usually not so well suited for temporal reasoning as model checkers are;
- Theorem provers wins in the expressiveness of the logics they use, when the problem involves reasoning about the system state;
- But such expressive power comes at the cost of decidability, which requires human intervention in the proof process;
- Both model checkers and theorem provers have pros and cons. In model checking, the effort goes mainly into developing the model and tuning the checking process, while in theorem proving, the effort goes mainly into guiding the theorem prover through the proofs.

Further, deeper investigations are performed (and tools developed) into the usage of model checking (only), in order to verify interactive systems. In [Campos & Harrison 2001], the authors propose the MAL interactor language to describe interactors that are based on MAL, and a tool called `i2smv` is proposed to translate MAL specifications into the input language of the SMV model checker.

To support the whole process, a toolbox called IVY is developed [Campos & Harrison 2009]. In this toolbox, the `XtrmSwing` tool performs reverse engineering of user interfaces written in Java/Swing (Figure 16), generating specifications in the MAL interactor language. Such specifications can be modified in the Model Editor provided by the toolbox, and the translation into an SMV specification is provided by the `i2smv` tool. Besides, an editor of properties is provided, in which the designer can select from a number of patterns of properties that best suit the analysis needs, and instantiate them with actions and attributes from the model [Campos & Harrison 2009].

In this framework, the properties are specified using the CTL (*Computational Tree Logic*) temporal logic, allowing the verification of *usability* and *functional* properties [Campos & Harrison 2008]. Particularly, the following *usability* properties can be expressed:

- *feedback* [Campos & Harrison 2008]: a given action provides a response;
- *behavioral consistency* [Campos & Harrison 2008]: a given action causes consistent effect;
- *reversibility* [Campos & Harrison 2008]: the effect of an action can be eventually reversed/undone;
- *completeness* [Campos & Harrison 2009]: one can reach all possible states with one action.

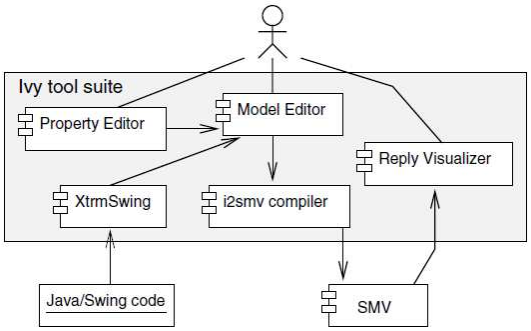


Figure 16: IVY - a tool for verifying interactive systems [Silva *et al.* 2007]

The approach covers the three aspects we are considering in this thesis: in [Campos & Harrison 2011], the approach is used to model the functional core and user interfaces of an infusion pump; and assumptions about user behaviors are covered in [Campos & Harrison 2007], by strengthening the pre-conditions on the actions the user might execute.

The approach is applied to several case studies [Campos & Harrison 2001, Harrison *et al.* 2013], specifically, in safety-critical systems (e.g., healthcare systems [Campos & Harrison 2009, Campos & Harrison 2011, Campos *et al.* 2014, Harrison *et al.* 2015] and avionics systems [Campos & Harrison 2007, Doherty *et al.* 1998, Sousa *et al.* 2014]), showing that the approach scales well to real-life applications. However, no case study applies this approach to plastic user interfaces.

f) D'Ausbourg *et al.* (France, 1996–2002)

Another approach based on the York interactor model is proposed in [d'Ausbourg *et al.* 1998, d'Ausbourg 1998]. These authors push further the modeling of an interactive system by events and states initially proposed by the York approach.

Their interactor model is called CERT (Figure 17). It also contains an internal state, and the interface between an interactor and its environment consists of a set of input and output events. Both internal state and events are described as flows. Informally, a flow is a sequence of values [d'Ausbourg *et al.* 1998], and the Boolean flows used to define a CERT interactor are:

- F_s: flow describing the internal state of the interactor;
- F_u: flow describing events associated with user actions;
- F_e: flow describing events sent by the environment;
- F_p: flow describing the presentations displayed to the user;
- F_a: flow describing actions triggered by the interactor.

The relation between events and states found in the CERT interactor model is illustrated in Figure 18, in which the behavior of a push button is described by the following Boolean flows: $F_u = \{on, press\}$, $F_e = \{\emptyset\}$, $F_s = \{selected\}$, $F_p = \{highlighted\}$ and $F_a = \{trigger\}$. Boolean flows are represented by leading or trailing edges – a leading edge indicates a `true` value and an trailing edge a `false` value. For instance, a leading edge of the `press` flow expresses

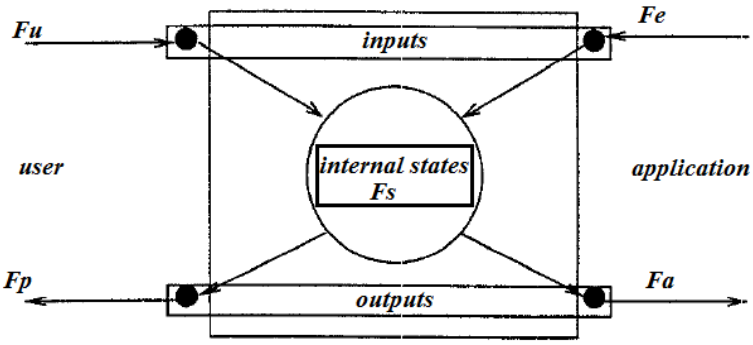


Figure 17: The CERT interactor, adapted from [d'Ausbourg *et al.* 1998]

than the user pressed the mouse button (`push` event), while a trailing edge indicates that the user released it.

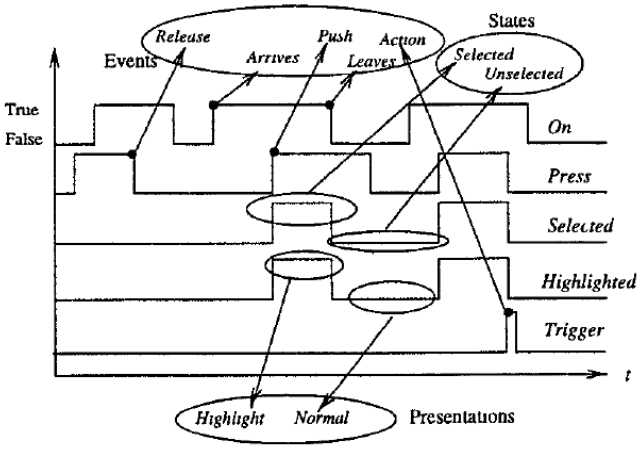


Figure 18: Boolean flows of a CERT interactor representing a push button [d'Ausbourg *et al.* 1998]

In this example [d'Ausbourg *et al.* 1998], the `on` and `press` flows take a *true* value, respectively, when the user places the mouse on the button and while the user presses the button. When both events take place, the internal state of the interactor changes to *selected* (represented in the image by the third horizontal flow), and the presentation changes to *highlighted*. Finally, the `trigger` flow takes a *true* value when an action is triggered by the interactor. The configuration of these flows shows that this push button is selected only if the mouse button is pressed while the pointing device is already on the interactor. So, the sequence of events [*Arrives, Push*] leads to the *Selected* state of this interactor.

Such representation of interactors by flows allows their specification using the LUSTRE data flow language. A system described in LUSTRE is represented as a network of nodes acting in parallel. Each node transforms input flows into output flows at each clock tick.

The proposed verification approach is depicted in Figure 19. The UIs are first described in a user interface generator called UIM/X. This tool generates a UIL file (a text file containing

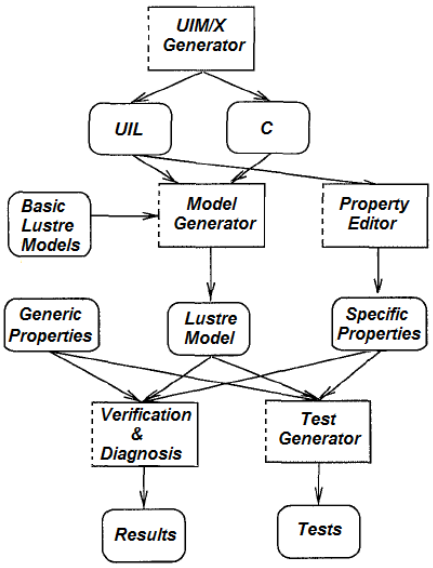


Figure 19: A verification environment using CERT interactors, adapted from [d'Ausbourg *et al.* 1998]

the structure and hierarchy of widgets of the UI) and a C file (which contains the instructions of the callback procedures). Both the UIL and the C files are used to automatically generate the CERT interactors in LUSTRE [d'Ausbourg *et al.* 1996]. The analyzers that automatically generate these LUSTRE models have been developed using the system Centaur.

The LUSTRE formal model is then verified by model checking. Verification is achieved by augmenting the system model with LUSTRE nodes describing the intended properties, and using the Lesar tool to traverse the state space generated from this new system. The properties can be either specific or generic properties. Specific properties deal with how presentations, states, and events are dynamically linked into the UIs, and they are automatically generated from the UIL file (they correspond to *functional* properties). Generic properties might be checked on any user interface system, and they are manually specified (they correspond to *usability* properties). The verification process allows the generation of test cases, using the behavior traces that lead to particular configurations of the UI where the properties are satisfied.

In particular, the following *usability* properties are verified [d'Ausbourg *et al.* 1998]:

- *reactivity*: the UI emits a feedback on each user action;
- *conformity*: the presentation of an interactor is modified when its internal state changes;
- *deadlock freedom*: the impossibility for a user to get into a state where no actions can be taken;
- *unavoidable interactor*: the user must interact with the interactor at least once in any interactive session of the UIs.

As well as the following *functional* property:

- *rule set connectedness*: an interactor is reachable from any initial state.

A drawback of the approach is that it does not handle sophisticated data types in the system modeling. The representation of the internal system state and events by Boolean flows considerably limits the modeling capabilities of the approach.

The approach is applied to critical systems, specifically, to the avionics field [d'Ausbourg 2002]. In this case study, the interactions of the pilot with the system and the behavior of the functional core are modeled, which shows that the approach scales well to real-life applications. Besides, the previous push button example (Figure 18) demonstrates that the approach also covers the modeling of the UIs. No application was found, however, to plastic user interfaces.

g) Bumbulis *et al.* (Canada, 1995–1996)

Similar to interactor models, user interfaces can be described by a set of interconnected primitive components [Bumbulis *et al.* 1995a, Bumbulis *et al.* 1995b, Bumbulis *et al.* 1996]. The notion of component is similar to that of interactor, but a component is more closely related to the widgets of the UI. Such component-based approach allows both rapid prototyping and formal verification of user interfaces from a single UI specification (Figure 20).

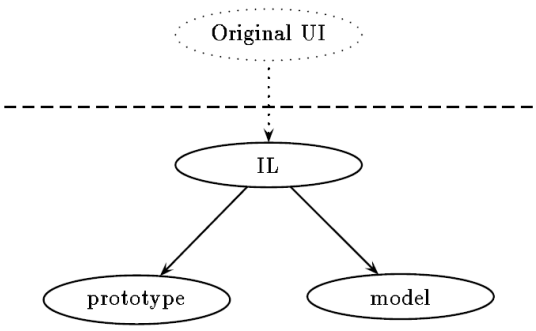
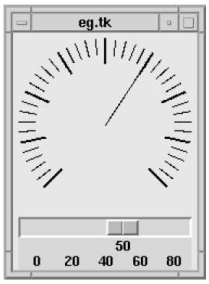


Figure 20: UI prototyping and verification [Bumbulis *et al.* 1995a]

In the Bumbulis *et al.*'s approach, user interfaces are described as a hierarchy of interconnected component instances using the Interconnection Language (IL). Investigations have been conducted into the automatic generation of IL specifications by re-engineering the UIs [Bumbulis *et al.* 1995a]. However, such automatic generation is not described in the paper. From such component-based IL specification of the UI, a Tcl/Tk code is mechanically generated, in order to provide a UI prototype for experimentation (Figure 20), as well as a HOL (*Higher-Order Logic*) specification for formal reasoning using theorem proving [Bumbulis *et al.* 1995a]. Higher-order logic extends first-order logic by allowing higher-order variables (i.e., variables whose values are functions) and higher-order functions (i.e., functions whose arguments and/or results are other functions).

Properties are specified as predicates in Hoare logic, a formal system with a set of logical rules for reasoning about the correctness of computer programs. Proofs are constructed manually, even though investigations to mechanize the process have been conducted [Bumbulis *et al.* 1995a]. No *usability* properties are verified in this approach. Instead, the approach permits *functional* properties to be verified, which are directly related to the expected behavior of the modeled UI.

Figure 21a illustrates a user interface to which the approach is applied [Bumbulis *et al.* 1995b]. It consists of a dial and a slider related to each other so that the slider will track the motion of the dial and vice-versa. The corresponding IL specification (Figure 21b) consists of two primitive components, `Dial` and `Slider`, which are connected by means of the *changed* and *set* ports. Each port has a polarity specifying whether the port *requires* (<) or *provides* (>) values. Ports are then bound in order to exchange values. Values sent to a *set* port update the value of the component; values are sent to a *changed* port when the component's value changes (either as a result of the user's actions, or as a result of a value being sent to the set port [Bumbulis *et al.* 1995a]).



(a) User interface

```
Frame primitive
Dial changed>int set<int primitive
Slider changed>int set<int primitive

Main {
    f:Frame f.d:Dial f.s:Slider

    f.d.changed --> f.s.set
    f.s.changed --> f.d.set
}
```

(b) The IL description of the UI

Figure 21: An example of a user interface with a dial and a slider [Bumbulis *et al.* 1995a]

The approach covers only the modeling and verification of user interfaces. The user and the functional core are not modeled. No application to safety-critical systems was found in the literature. The slider user interface (Figure 21a) is quite simple: neither bigger nor more realistic examples are provided. In the IL language, UIs are described by means of bound components, thus, it is not clear how to model more complex UIs in this approach, since UI components are not always bound to each other. In addition, it is not clear how multiple UIs could be modeled, neither the navigation modeling between such UIs. All these aspects indicate that the approach does not scale well for larger applications.

h) Palanque *et al.* (France, 1990–2015)

In [Palanque & Bastide 1995], another approach is proposed to modeling and verifying interactive systems with a different formalism: Petri nets [Carl 1962]. Being a graphical model, Petri nets can be easier to understand than textual descriptions.

Originally designed for the modeling and implementation of event-driven interfaces [Bastide & Palanque 1990] and nowadays covering the modeling of a full interactive system (not only the user interfaces), the ICO formalism (*Interactive Cooperative Objects*) permits applications to be prototyped and tested before they are fully implemented, by means of cooperative objects.

A system described via ICO is modeled as a set of objects that cooperate to perform the system tasks. ICO uses concepts borrowed from the object-oriented formalism (such as inheritance, polymorphism, encapsulation, and dynamic instantiation) to describe the structural or static aspects of systems, such as its attributes and the operations it provides to its environment. In addition, ICO uses high-level Petri nets to describe the dynamics and behavioral aspects of the system, such as the spontaneous activity of the object, the activation (on the UI) of the

operations the object provides according to its internal state, and the effect of these operations on the internal state of the object [Navarre *et al.* 2009].

An object in the ICO formalism is characterized by five components [Navarre *et al.* 2005], partially illustrated in Figure 22 with an example of an ATM (*Automated teller machine*):

- *Cooperative object*: (item 1 in Figure 22) models the behavior of the interactive system. This behavior is described by a high-level Petri net called ObCS (*Object Control Structure*), where the tokens that circulate in the network can carry data;
- *Presentation*: (item 2 in Figure 22) describes the UI appearance, i.e., a set of widgets;
- *Activation function*: (item 3 in Figure 22) is responsible for linking user actions on the UI with the services offered by the objects;
- *Rendering function*: provides consistency between the internal state of the system and its external appearance;
- *Availability function*: guarantees that the services provided by an object will only be available if the corresponding Petri net transition is available.

Class ATM
Attributes
a, b : Real; p : Integer; c : Credit_Card; r : {ABORT, CANCEL, RETRY};
Methods
Okpin <p : Integer, c : Credit_Card> : Boolean;
Ok <a,b:real> : Boolean; Avail<c:Card> : Real;
Services
InsertCard <c:Card>; Select <a: Real>; EnterPin <p : Integer>;
WithdrawCash; WithdrawCard; Balance;
ObCS (Figure containing the ObCS of the ATM Class) **(1)**
Presentation **(2)**
Layout -- Not presented
Activation function: **(3)**

Widget	User's actions	Service
Pushbutton Balance	Click	Balance
Pushbutton Withdrawal	Click	Withdrawal
Pushbutton InsertCard	Click	InsertCard
Pushbutton EnterPin	Click	EnterPin
Pushbutton Select	Click	Select
Pushbutton Cash	Click	WithdrawCash
Pushbutton Card	Click	WithdrawCard

End

Figure 22: Example of an object in the ICO formalism – an ATM systemadapted from [Palanque *et al.* 1996]

An ATM enables users to manage their bank account. In the case study of this example, the bank's clients can perform tasks such as obtaining cash, getting a statement about their account, ordering a check book, etc. The example shown in Figure 23 describes the *Obtain_Cash* task, which is included in the Petri net describing the ATM class (item 1 in Figure 22). The system states are modeled at each moment by tokens placed in the Petri net places (e.g., the initial

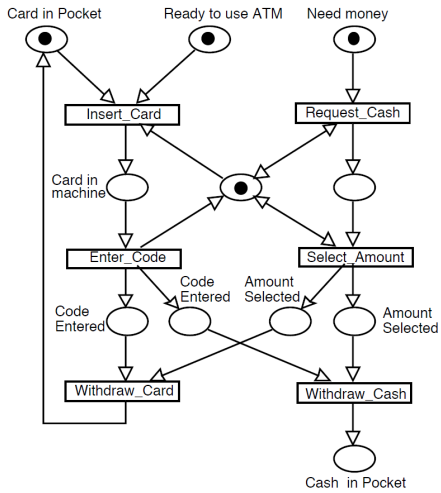


Figure 23: Example of a Petri net for the *Obtain_Cash* task [Palanque *et al.* 1996]

state of the system modeled in Figure 23, where the tokens are in *card in pocket*, *ready to use ATM*, etc., places).

Once the system is modeled, it is possible to apply model checking to verify *usability* and *functional* properties. For instance, the following *usability* properties can be verified [Palanque & Bastide 1995]:

- *predictability*: the user is able to foresee the effects of a command;
- *deadlock freedom*: the impossibility for a user to get into a state where no actions can be taken;
- *reinitiability*: the ability for the user to reach the initial state of the system.

As well as the following *functional* properties:

- *exclusion of commands*: commands which must never be offered at the same time (or, on the contrary, must always be offered simultaneously);
- *succession of commands*: the proper order in which commands may be issued; for instance a given command must or must not be followed by another one, immediately after or with some other commands in between;
- *availability*: a command is offered all the time, regardless of the state of the system (e.g., a help command).

The specification is verified using Petri net property analysis tools [Palanque *et al.* 1996]. In order to automate the process of property verification, the ACTL temporal logic can be used to express the properties, which are then proved by model checking the Petri net *marking graph* [Palanque *et al.* 1999]. A *marking* is a certain distribution of tokens in the Petri net places. A *marking graph* of a Petri net is a finite-state machine that illustrates all the possible

configurations of the tokens in the Petri net, in accordance with the transitions between places that are defined by the Petri net.

The ICO approach also permits user's cognitive behavior to be modeled by a common Petri net for system, device and user [Moher *et al.* 1996]. An environment called PetShop (for *Petri net Workshop*) [Bastide & Palanque 1995] is developed for supporting the design of interactive systems according to the ICO methodology. Alternatively, the Java PathFinder model checker is used to verify a set of properties on a safety-critical application in the interactive cockpit systems modeled with ICO [Brat *et al.* 2013].

The approach has been applied to other case studies in safety-critical systems in the space domain (for instance: [Palanque *et al.* 1997, Bastide *et al.* 2003, Bastide *et al.* 2004, Brat *et al.* 2013]). These case studies and the maturity of the tools show that the approach scales well to real-life applications. However, no case study applies this approach to the nuclear power plant domain. It is not clear, for instance, how the multidisciplinary users would be considered by this approach. Generation of contextual user interfaces are proposed in [Martinie *et al.* 2014], but there is no change in the context of use: UIs are generated and distributed across another display, to support operators in executing specific procedures. Finally, dynamic reconfiguration of user interfaces are proposed in [Navarre *et al.* 2008], allowing operators to continue interacting with the interactive system even though part of the hardware side of the user interface is failing. This could be seen as an application of plasticity: UI adaptation in order to recover from faults in the hardware such as displays. However, this is more related to fault-tolerant issues than to the capacity of the UI to adapt to changes in the user, platform, and/or environment.

Besides, the verification based on Petri net properties has limitations exposed in [Navarre *et al.* 2009]. The analysis is usually performed on the underlying Petri net (a simplified version of the original Petri net). A drawback is that properties verified on the underlying Petri net are not necessarily true on the original Petri net. Thus, the results of the analysis are essentially indicators of potential problems in the original Petri net.

2.6.2 Interactive Systems Modeled Globally

Some approaches do not use compositions of smaller parts (i.e., agents, interactors, components, or objects) to model interactive systems. Yet, realist and complex case studies can be modeled and verified using these approaches. In this section we present several of such approaches.

a) Dix *et al.* (England, 1985–1995)

The PIE model [Dix *et al.* 1987] considers interactive systems as a "black-box" entity that receives a sequence of inputs (keystrokes, clicks, etc.) and produces a sequence of perceivable effects (displays, LEDs, printed documents, etc.). The main idea is to describe the user interfaces in terms of the possible inputs and their effects [Dix 1991]. Such practice is called *surface philosophy* [Dix 1988] and aims at omitting parts of the system that are not apparent to the user (the internal details of systems, such as hardware characteristics, languages used, or specification notations). The domain of input sequences is called `P` (standing from *programs*), the domain of effects is called `E` and both are related by an interpretation function `I` that determines the effects of every possible command sequence (Figure 24).

The effects `E` can be divided into permanent *results* (e.g., print-out) and ephemeral *displays* (the actual UI image). Such specialization of the effects constitutes another version of the PIE

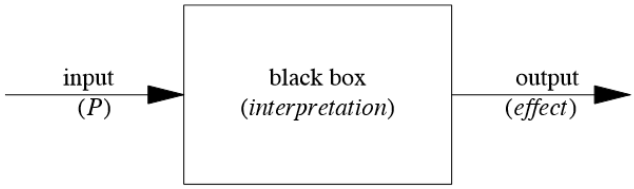


Figure 24: The PIE model [Dix 1991]

model, called *Red*-PIE model [Dix 1991] (Figure 25).

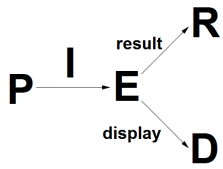


Figure 25: The Red-PIE model, adapted from [Dix 1991]

In such models, user inputs are called commands (`C`), and can be used at various levels of granularity (e.g., individual keystrokes, or operations on a spreadsheet) [Dix 1995]. The command history is called `P`, and defined as a sequence of commands (`P`=**seq** `C`). For instance, a calculator that adds up single elements is formalized as follows [Dix 1991]:

$C = 0, ..., 9$
$P = \mathbf{seq}\ C$
$E = \mathbb{N}\ -\ \textit{the natural numbers}$
$I(null) = 0$
$I(pc) = I(p) + c$

The interpretation function basically says: one starts off with a running sum of zero; if at any stage one enters a new number (`c`) it gets added to the current running sum (`I(p)`).

The PIE model provides a generic way of modeling interactive systems and permits the following *usability* properties to be formalized:

- *predictability* [Dix 1995]: the UI shall be predictable, i.e., from the current effect it should be possible to predict the effect of future commands. Such property is formalized as follows:

  $predict : D > R$
  $\forall s \in E : predict(display(s)) = result(s)$

  The first line says that *predict* is a function that from a display produces a result. Using the $\forall$ (the universal quantifier) operator, the second line states that, considering any state s, the display of that state applied to the *predict* function results in exactly the same as if applied to the *result* function directly;

- *simple reachability* [Dix 1991]: all system effects can be obtained by applying some sequences of commands;
- *strong reachability* [Dix 1988]: one can get anywhere **from anywhere**;
- *undoability* [Dix *et al.* 1987]: for every command sequence there is a function "undo" which reverses the effect of any command sequence;
- *result commutativity* [Dix *et al.* 1987]: irrespective of the order in which different UIs are used, the result is the same.

The PIE and Red-PIE models are ones of the first approaches that used formal notations for the modeling of interactive systems and desired properties. However, their mathematical notations are very abstract, and no tool support is provided neither for modeling nor for property verification. For this reason, the scalability of the approach cannot be measured. Considering the modeling coverage, the framework describes how to formalize interactive systems and the user interfaces in an abstract way. Users are not described, though. Besides, no application to safety-critical systems is reported.

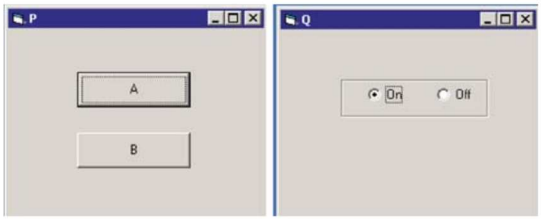
b) Bowen and Reeves (New Zealand, 2005–2015)

In [Bowen & Reeves 2007a, Bowen & Reeves 2008a], an approach is proposed to bridge the gap between user interface (UI) design and formal methods. In this approach, UIs are formally described using a *presentation model* (PM) and a *presentation and interaction model* (PIM). In order to support the use of formal methods for non-expert designers, an approach is described in [Bowen 2015] for reverse-engineering Java applications to automatically generate these formal models. The *presentation model* is specified using Z and μCharts (a Statechart-like language), and the *presentation and interaction model* is specified using finite-state machines.

The *presentation model* is used to formally capture the meaning of an informal design artifact such as a scenario, a storyboard, or a UI prototype [Bowen & Reeves 2007a]. Since it describes such informal design artifact in terms of the widgets of the design, the presentation model permits static properties of a UI to be expressed.

Figure 26 illustrates an example of a presentation model, in which UIs are described by two components, `p` and `q` (these may be different windows, or different states of the UIs). In this example, `r` describes the entire UI (i.e., the combination of `p` and `q`), and `p` has two widgets, `aCtrol` and `bCtrl`. The behaviors associated with `aCtrl` (resp. `bCtrl`) are `eAction` and `fAction` (resp. `dAction`). Here, `q` has one widget, `cSel`, which is a `SValSelector` with the `eAction` and `fAction` behaviors. Therefore, the presentation model `r` is the combination of all the widgets of `p` and `q` (the ":" operator acts as a composition), and describes the total possible behaviors of the UI [Bowen & Reeves 2007a].

The use of the triplet $\langle p, q, r \rangle$ for each widget in the presentation model does not hold enough information to accurately represent the UI dynamic behavior. In order to address this issue, the authors use finite-state machines, which in conjunction with the presentation model, forms the *presentation and interaction model* (PIM) [Bowen & Reeves 2007a]. The PIM model consists in a finite-state machine decorated with parts of the presentation model. A software application with four distinct windows/dialogs have four PMs, each describing the widgets and behaviors of one of the windows. The PIM then have four states, each state corresponding to one of the PMs.



(a) User interfaces

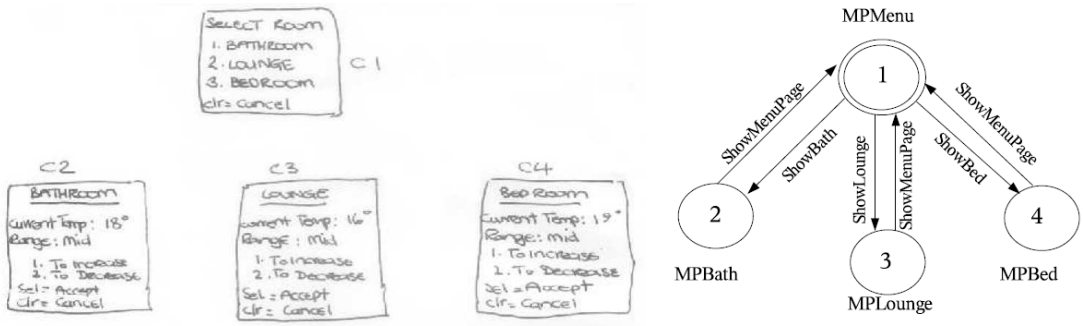
PModel	*p q r*
Widgetname	*aCtrl bCtrl cSel*
Category	*ActionControl SValSelector*
Behaviour	*dAction eAction fAction*

p *is* $(aCtrl, ActionControl, (eAction\ fAction))$
$(bCtrl, ActionControl, (dAction))$
q *is* $(cSel, SValSelector, (eAction\ fAction))$
r *is* $p : q$

(b) Presentation model

Figure 26: *Presentation model* of a user interface [Bowen & Reeves 2008a]

An example of a *presentation and interaction model* is given in Figure 27. It consists of a home heating control system accessible via a mobile phone, which supports the monitoring and control of temperatures in a number of different rooms [Bowen & Reeves 2007a]. The application has four UIs (`C1`,`C2`,`C3` and `C4`). The corresponding PIM captures the way in which one moves from one part of the interface to another.



(a) User interface prototype (b) PIM

Figure 27: *Presentation and interaction model* of a home heating control system [Bowen & Reeves 2007a]

The approach is applied to several case studies. In [Bowen & Reeves 2007b], the authors use the models in the design process of UIs for the PIMed tool, a software editor for the presentation models and PIMs. However, in this work, no automated verification of the presentation model and the PIMs of the editor is proposed. The formal models are manually inspected. For example, in order to verify the deadlock freedom property, the authors use a manual walk-through procedure in the PIMs, which is an error-prone and time-consuming procedure.

In [Bowen & Reeves 2013a], the verification is automatized by the ProZ tool, allowing *usability* properties to be verified using model checking. The toll is used to analyze the models, to animate them during design and to model check properties expressed in LTL. The kinds of properties that can be verified are:

- *total reachability* [Bowen & Reeves 2007b]: one can get to any state from any other state;
- *deadlock freedom* [Bowen & Reeves 2007b]: a user cannot get into a state where no action

can be taken;

- *behavioral consistency* [Bowen & Reeves 2008a]: controls with the same behavior have the same name;
- *minimum memory load on user* [Bowen & Reeves 2008a]: users do not have to remember long sequences of actions to navigate through the UI.

The presentation model and PIMs can be used to derive tests [Bowen & Reeves 2013b, Bowen & Reeves 2011]. Abstract tests are generated from the formal models. These tests are then manually instantiated and executed against the application under test.

Another case study to which these formal models have been applied relates to a safety-critical system in the healthcare domain [Bowen & Reeves 2013a]: this time, the verification is tool supported. The authors model a syringe pump, a device commonly used to deliver pain-relief medication in hospitals and respite care homes. The device has ten widgets, which include the display screen, eight soft keys and an audible alarm. Temporal safety properties and invariants (to check boundary values) are verified against the formal models using ProZ and Z/EVES [Bowen & Reeves 2005, Bowen & Reeves 2013a].

Although the approach mainly focuses on modeling the user interfaces, the presentation model can also be used to model the user operations. The main goal is to ensure that all user operations described in the formal specification have been described in the UI design [Bowen & Reeves 2007a]. In addition, the case study described in [Bowen & Reeves 2013a] also models the functional core of the infusion pump system, showing that the approach covers all three aspects: users, UIs and functional core.

Finally, despite the fact that the approach is tool supported and has been applied to real case studies in healthcare systems, there is no evidence that the approach scales well. The models are always manually written and relatively small, especially the PIM. In the given examples, the number of states and transitions of the finite-state machine ranges from 2 to 14 states and from 2 to 40 transitions. For the healthcare case study described in [Bowen & Reeves 2012] the authors indicate that the PIMs of the manual and pump are small enough to perform a manual inspection (10 states and 14 states respectively). To be more scalable, the approach should be evaluated on larger formal models, in which case some automations should be provided. Investigations have started in this direction [Bowen 2015], by reverse-engineering interactive systems to automatically generate formal models. But so far the tool was only tested on four small applications, only the presentation model has been generated, and only Java applications have been covered.

c) Aït-Ameur *et al.* (France, 1998–2014)

Another approach that also relies on theorem proving, but this time using the B method [Abrial 1996] to specify the interactive system, is proposed in [Aït-Ameur *et al.* 1998b, Aït-Ameur *et al.* 1999, Aït-Ameur *et al.* 2003a]. The approach permits task models to be validated. Task models can be used to describe a system in terms of tasks, subtasks, and their temporal relationships. A task has an initial state and a final state, and is decomposed in a sequence of several subtasks.

The approach uses the B method for representing, verifying and refining specifications. A B model is composed of state variables and a set of *atomic* events. State variables are described

by guarded commands and generalized substitutions (assignment `, ANY, BEGIN` and `SELECT`). For example, in `Evt = SELECT P THEN S END;` the event `Evt` is fired and the statement `S` is executed when the Boolean condition `P` is true. Events modify the state of the specified system. The authors use the set of events to define a transition system that permits the dialog controller of the interactive system to be represented.

The CTT (*Concur Task Trees*) notation [Paternò *et al.* 1997] is used to represent task models. In [Aït-Ameur *et al.* 2003a], only the CTT operator called "sequence" between tasks is covered. In further work [Aït-Ameur *et al.* 2005, Yamine *et al.* 2005, Aït-Ameur *et al.* 2009] the authors describe how the semantics of CTT can be formally described in Event B allowing to translate, with generic translation rules, every CTT construction (interruption and disabling included) in Event B, which is the event-based definition of B method.

This usage of Event B to encode CTT task models is described in several case studies [Aït-Ameur & Baron 2004, Aït-Ameur & Baron 2006, Aït-Ameur *et al.* 2006, Cortier *et al.* 2007]. In particular, the approach is used to verify Java/Swing user interfaces [Cortier *et al.* 2007], from which Event B models are obtained. Such Event B models encapsulate the UI behavior of the application. Validation is achieved with respect to a task model that can be viewed as a specification. Following the approach previously explained, this task model is encoded in Event B, and assertions ensure that suitable interaction scenarii are accepted by the CTT task model. Demonstrating that the Event B formal model behaves as intended comes to demonstrate that it is a correct refinement of the CTT task model.

Moreover, the following *usability* properties can be verified:

- *robustness* [Aït-Ameur *et al.* 2003b]: these properties are related to system dependability;
- *visibility* [Aït-Ameur *et al.* 1999]: relates to feedback and information delivered to the user.

As well as the following *functional* properties:

- *reachability* [Aït-Ameur *et al.* 1999]: these properties express what can be done at the user interface and how can it be done;
- *reliability* [Aït-Ameur *et al.* 1999]: concerns the way the interface works with the underlying system;
- *behavioral properties* [Aït-Ameur & Baron 2006]: characterize the behavior of the UI suited by the user.

The proof of these properties is done using the invariants and assertions clauses of the B method, together with the validation of specific aspects of the task model (i.e., *functional* properties), thus permitting a full system task model to be validated. The Atelier B tool is used for an automatic proof obligation generation and proof obligation checking [Aït-Ameur 2000].

In order to compare this theorem proving-based approach to model checking-based approaches, the authors show how the same case study is tackled using both theorem proving (with Event B) and model checking (with Promela/SPIN) [Aït-Ameur *et al.* 2003b]. The case study consists in a franc/euro exchange application that converts from French francs to Euros and vice-versa. The authors conclude that both techniques permit the case study to be fully described, and that both permit robustness and reachability properties to be verified. The proof process of the Event B-based is not fully automatic, but it does not suffer from the state-space explosion of

model-checking techniques. The Promela-SPIN-based technique is fully automatic, but limited to finite-state systems on which exhaustive exploration can be performed. The authors conclude that a combined usage of both techniques would strengthen the verification of interactive systems.

An integration of the approach with testing is also presented in [Aït-Ameur *et al.* 2004]. Here, the informal requirements are expressed using the semi-formal notation UAN [Hix & Hartson 1993] (instead of CTT), and the B specifications are manually derived from this notation. To validate the formal specification, the authors use a data-oriented modeling language, named EXPRESS, to represent validation scenarios. The B specifications are translated into EXPRESS code (the B2EXPRESS tool [Aït-Sadoune & Aït-Ameur 2008]). This translation gives data models that represent specification tests and permits Event B models to be animated.

The approach is applied to several case studies in the avionics domain [Jambon *et al.* 2001, Aït-Ameur *et al.* 2014]. Specifically, the authors illustrate how to explicitly introduce the context of the systems in the formal modeling [Aït-Ameur *et al.* 2014]. The approach is also applied to the design and validation of multi-modal interactive systems [Aït-Ameur *et al.* 2010, Aït-Ameur *et al.* 2006, Aït-Ameur & Kamel 2004].

The case study described in [Aït-Ameur *et al.* 1998a] shows that the approach covers the modeling of users, UIs and the functional core. The numerous case studies and the maturity of the approach suggests that it scales well to real-life applications. However, no application was found to plastic user interfaces.

d) Loer and Harrison *et al.* (Germany, 2000–2006)

Another approach to verifying interactive systems is proposed in [Loer & Harrison 2002, Loer & Harrison 2006], also with the goal of making model checking more accessible to software engineers. The authors claim that in the avionics and automotive domains requirements are often expressed as Statechart models [Loer & Harrison 2002]. To introduce formal verification in the process, they propose an automatic translation from Statechart models (created with the Statemate toolkit) to the input language of the SMV model checker, which is relatively robust and well supported [Loer & Harrison 2006].

Such translation is part of the IFADIS toolbox (Figure 28), which also provides guided process of property specifications and a trace visualization to facilitate the result analysis of the model checker. The properties can be verified using Cadence SMV or NuSMV model-checking tools. Depending on the type of property, the model checker can output traces that demonstrate why a property holds or not [Loer & Harrison 2006]. The TraceVis tool (Figure 28) displays such traces using an enhanced tabular view.

The property editor helps designers to construct temporal-logic properties by making patterns available (Figure 29) and helping the process of instantiation [Loer & Harrison 2006]. Temporal-logic properties can be specified either in LTL (*Linear Temporal Logic*) or CTL (*Computational Tree Logic*).

The following *usability* properties can be verified:

- *reachability* [Loer & Harrison 2000]: are all the states reachable or not?
- *robustness* [Loer & Harrison 2000]: does the system provide fall-back alternatives in the case of a failure? or, alternatively, are the guards for unsafe states foolproof?
- *recoverability* [Loer & Harrison 2000]: does the system support undo and redo?

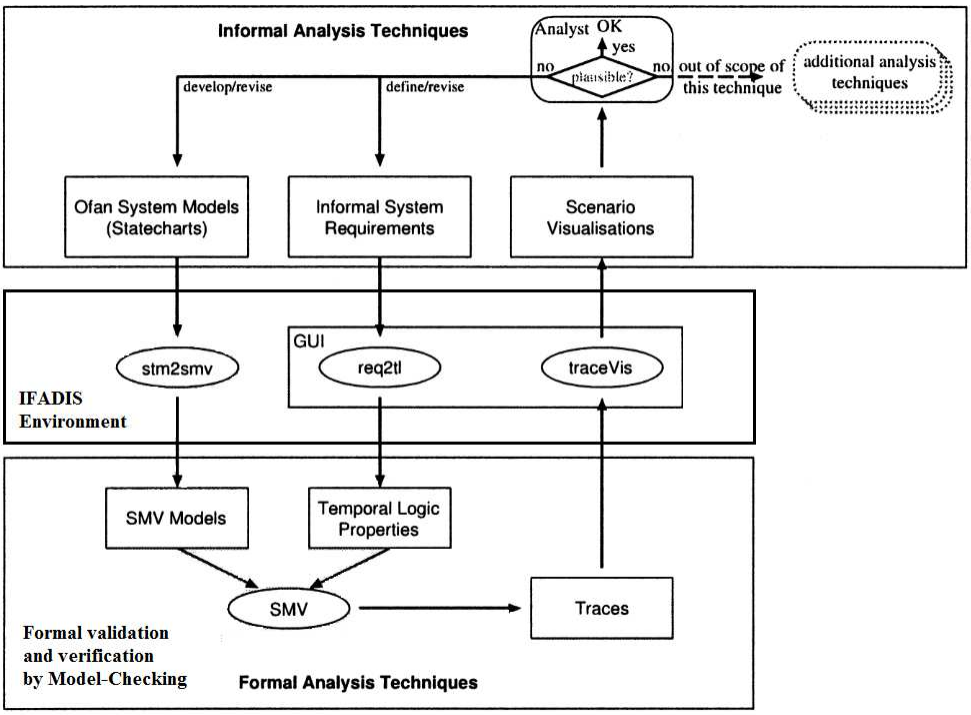


Figure 28: The IFADIS framework, adapted from [Loer & Harrison 2006]

Property	CTL template for property
reachability	AG EF(⟨*target_configuration*⟩)
mutual exclusion	AG !(⟨*configuration1*⟩ & ⟨*configuration2*⟩)
robustness	INVAR !⟨*triggering_condition*⟩; AG (⟨*starting_configuration*⟩ -> EF(⟨*target_configuration*⟩))
effect visibility	
(appropriateness)	AG(⟨*starting_configuration*⟩) -> AF(⟨*display_configuration*⟩)
(timeliness)	AG(⟨*starting_configuration*⟩ & ⟨*input_signal*⟩) -> AF(⟨*target_configuration*⟩ & *event_counter* = *n*)
recoverability	AG(⟨*starting_configuration*⟩ & ⟨*input_signal*⟩) -> EX EF(⟨*starting_configuration*⟩)
consistency	AG (⟨*input_signal*⟩) -> AX(⟨*intended target configuration*(*s*)⟩)
flexibility	AG(⟨*starting_configuration*⟩ & ⟨*reset_counter*⟩) -> EF(⟨*target_configuration*⟩ & ⟨*input_counter* = *m*⟩);

Figure 29: Property specification patterns in CTL [Loer & Harrison 2002]

- *visibility of system status* [Loer & Harrison 2000]: does the system always keep the users informed about what is going on, through appropriate feedback within reasonable time?
- *recognition rather than recall* [Loer & Harrison 2000]: is the user forced to remember information from one part of the dialog to another?
- *behavioral consistency* [Loer & Harrison 2006]: does the same input always yield the same

effect?

In particular, the *reachability* property here is classified as a *usability* property because here it is defined as generic property, which can be applied to any interactive system (i.e., "are all the states reachable or not?"), in contrast to the classification of the *reachability* property in Subsection c) on page 41, for instance, where it is classified as a *functional* property because over there it expresses what can be done at the UI, and how can it be done, which is something that is usually defined in the system requirements. See Subsection 2.6 on page 18 to recall how we define the *usability* and *functional* properties in this thesis.

Concerning the modeling coverage of the approach, the authors describe five pre-defined elements in which the formal model is structured [Loer & Harrison 2000]:

1. *control elements*: description of the widgets of the UIs;
2. *control mechanism*: description of the system functionalities;
3. *displays*: description of the output elements;
4. *environment*: description of relevant environmental properties;
5. *user tasks*: sequence of user actions that are required to accomplish a certain task.

Therefore, their model covers the three aspects we consider in this thesis: the user, UIs, and the functional core.

Although the approach is not applied to many case studies (i.e., only to the avionics domain [Loer & Harrison 2006]), several reasons indicate that the approach scales well to real-life applications. The approach is supported by a tool that provides: a translation from engineering models (Statecharts) to formal models (SMV specifications); a set of property patterns to facilitate the specification of properties; and a trace visualizer to interpret the counter examples generated by the model checker. It is used in the case study described in [Loer & Harrison 2006], and an evaluation shows that the tool improves the usability of model checking for non-experts [Loer & Harrison 2006]. However, no case study applies the approach to plastic user interfaces.

e) Thimbleby *et al.* (England, 1987–2015)

In the healthcare domain, several investigations of medical device user interfaces have been conducted in Swansea University and Queen Mary University of London. Specifically, investigations are conducted of interactive hospital beds [Acharya *et al.* 2010], for user interfaces of drug infusion pumps [Masci *et al.* 2014a, Cauchi *et al.* 2012a, Masci *et al.* 2015, Thimbleby & Gow 2008], and interaction issues that can lead to serious clinical consequences.

Infusion pumps (Figure 30) are medical devices used to deliver drugs to patients. Deep investigation has been done of the data entry systems of such devices [Thimbleby 2010, Oladimeji *et al.* 2011, Masci *et al.* 2011, Thimbleby & Gimblett 2011, Cauchi *et al.* 2012b, Gimblett & Thimbleby 2013, Oladimeji *et al.* 2013, Cauchi *et al.* 2014, Tu *et al.* 2014, Li *et al.* 2015]. If a nurse makes an error in setting up an infusion (for instance, a number ten times larger than the necessary for the patient's therapy), the patient may die. Under-dosing is also a problem: if a patient receives too little of a drug, recovery may be delayed or the patient may suffer unnecessary pain [Masci *et al.* 2011]. Figure 30 illustrates various entry systems for infusion

pumps. The number entry system of the user interface on the Alaris GP infusion pump has four buttons (Figure 30a). A pair of buttons is used to increase the value, and a second pair is used to decrease the value displayed [Masci *et al.* 2015]. The number entry system of B-Braun Infusomat Space (Figure 30b) is an example of "5-keys" user interfaces: four arrow keys and a confirmation button [Masci *et al.* 2015]. Figure 30c illustrates an infusion pump (which identity is concealed for confidentiality reasons [Masci *et al.* 2013a]) with a numeric keypad and a confirmation button.

(a) Alaris GP (b) B-Braun Infusomat Space (c) Pump's identity is concealed

Figure 30: Examples of infusion pumps

The authors report several issues with the data entry system of such pumps [Masci *et al.* 2014a]. For instance, the infusion pump of Figure 30c mistakenly discards the decimal point in input key sequences for fractional numbers between $[100.1, 1200)$. For example, the input key sequence 100.1 is registered as 1001 without any warning or error message. This issue arises because of a constraint imposed in a routine of the pump's software: numbers above or equal to 100 cannot have a fractional part. Due to this constraint, the pump erroneously ignores the decimal point in the key sequence 100.1, and registers it as 1001, a value ten times larger than the intended one.

Such issue is detected among with several others [Masci *et al.* 2014a] using the approach depicted in Figure 31. In this approach, the C++ source code of the infusion pump of Figure 30c is manually translated into a specification in the PVS formal language ([a] in Figure 31). *Usability* properties such as *consistency of actions* and *feedback* are formalized ([b] in Figure 31) as *invariants* to be established using theorem proving:

- *consistency of actions*: the same user actions should produce the same results in logically equivalent situations;
- *feedback*: it ensures that the user is provided with sufficient information on what actions have been done and what result has been achieved.

A behavioral model is then extracted ([c] in Figure 31), in a mechanized manner, from the PVS formal specification. This model captures the control structure and behavior of the software related to handling user interactions. Theorem proving is used to verify that the behavioral model satisfies the *usability* properties. Lastly, the behavioral model is exhaustively explored to generate a suite of test sequences ([d] in Figure 31) [Masci *et al.* 2014a].

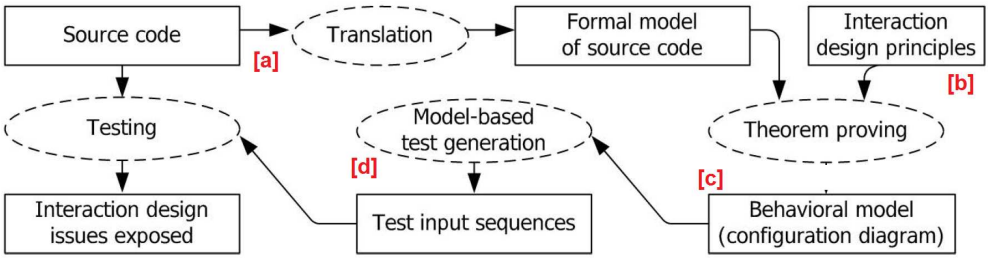


Figure 31: Verification approach using PVS, adapted from [Masci *et al.* 2014a]

A similar approach is described in [Masci *et al.* 2013a], in which the PVS specification is automatically discovered [Thimbleby 2007a, Gimblett & Thimbleby 2010] from reversely engineering the infusion pump software. Besides, *functional* properties are extracted from the safety requirements provided by the US medical device regulator FDA (*Food and Drug Administration*), to make sure that the medical device is reasonably safe before entering the market [Masci *et al.* 2013a].

The same FDA safety requirements are used to verify a PVS formal model of another device, the Generic Patient Controlled Analgesia (GPCA) infusion pump [Masci *et al.* 2013b]. In this work, the authors propose the usage of formal methods for rapid prototyping of user interfaces. Once verified, the formal model of the infusion pump is automatically translated into executable code through the PVS code generator, providing a prototype of the GPCA user interface from a verified model of the infusion pump.

An approach to integrating PVS executable specifications and Stateflow models is proposed in [Masci *et al.* 2014b], aiming at reducing the barriers that prevent non-experts from using formal methods. It permits Stateflow models to be verified, avoiding the hazards of translating design models created in different tools.

All the work mentioned in this subsection is based on the PVS theorem prover. Nevertheless, model checking can also be used in order to formally verify medical devices [Thimbleby 2007b, Masci *et al.* 2011, Masci *et al.* 2015]. For example, the authors model the Alaris GP (Figure 30a) in [Masci *et al.* 2015], and the B-Braun Infusomat Space (Figure 30b) infusion pumps in the higher-order logic specification language SAL (*Symbolic Analysis Laboratory*) [De Moura *et al.* 2004]. Afterwards, model checking is applied to verify the predictability of user interfaces, a *usability* property expressed in the LTL temporal logic. *Predictability* is defined in [Masci *et al.* 2011] as "if users look at the device and see that it is in a particular display state, then they can predict the next display state of the device after a user interaction".

The maturity of the approach described here, and its applications to numerous case studies are evidences that the approach scales well to real-life applications. No applications are reported, however, to plastic user interfaces.

Concerning the modeling coverage, the aforementioned case studies deal with the display and functionalities of the devices, but do not cover the modeling of the users interacting with such devices.

f) Miller *et al.* (USA, 1995–2013)

Also in the safety-critical domain, but in avionics, deep investigation has been conducted at Rockwell Collins of the usage of formal methods for industrial realistic case studies. Preliminary usage of formal methods aimed at creating consistent and verifiable system specifications [Hamilton *et al.* 1995], paving the way to the usage of formal methods at Rockwell Collins. Another preliminary use of formal methods was the usage of a synchronous language called RSML (*Requirements State Machine Language*) to specify requirements of a Flight Guidance System. Algorithms to translate specifications from this language to the input languages of the NuSMV model checker and the PVS theorem prover have been proposed [Miller *et al.* 2006], enabling one to perform verification of safety properties and functional requirements expressed in the CTL temporal logic (i.e., *functional* properties). Afterwards, deeper investigations are conducted to further facilitate the usage of formal methods.

According to [Miller 2009], relatively few case studies of model checking to industrial problems outside the field of engineering equipment are reported. One of the causes is the gap between the descriptive notations most widely used by software developers and the notations required by formal methods [Lutz 2000]. To alleviate the difficulties, as part of NASA's Aviation Safety Program (AvSP), Rockwell Collins and the research group on critical systems of the University of Minnesota (USA) develop the Rockwell Collins Gryphon Translator Framework [Hardin *et al.* 2009], providing a bridge between some commercial modeling languages and various model checkers and theorem provers [Miller *et al.* 2010]. The translation framework is illustrated in Figure 32. It supports Simulink, Stateflow, and SCADE models, and it generates specifications for the NuSMV, Prover, and SAL model checkers, the ACL2 and PVS theorem provers, and generates C and Ada code [Miller *et al.* 2010] (BAT and Kind are also included as target model checkers in [Cofer 2012]). Alternatively, Z specifications are also covered by the approach as an input language, since Simulink and Stateflow models can be derived from Z specifications [Hardin *et al.* 2009].

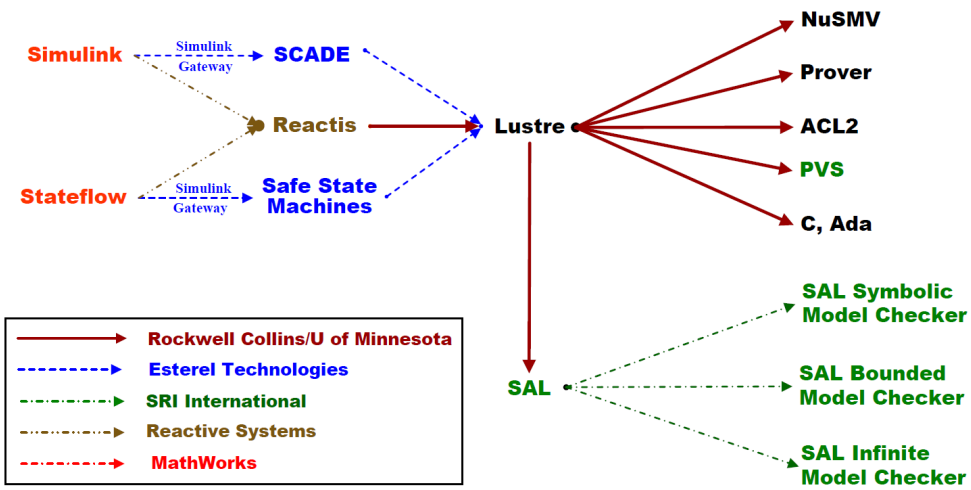


Figure 32: The Gryphon translator framework [Miller *et al.* 2010]

The Rockwell Collins translators use the LUSTRE language as an intermediate representation,

but LUSTRE is hidden from users [Miller *et al.* 2010]. Once in LUSTRE, the specification is loaded into an abstract syntax tree (AST) and several transformations are applied to it (Figure 33). Each transformation produces a new AST syntactically closer to the target specification language while preserving the semantics of the original LUSTRE specification. The number of transformations depends on the degree of similarity between source and target languages [Miller *et al.* 2010]. Tools are also developed to translate the counter-examples produced by the model checkers back to Simulink and Stateflow models [Cofer 2010], since for large systems it can be difficult to determine the cause of the violation of the property only by examining counter-examples [Whalen *et al.* 2008].

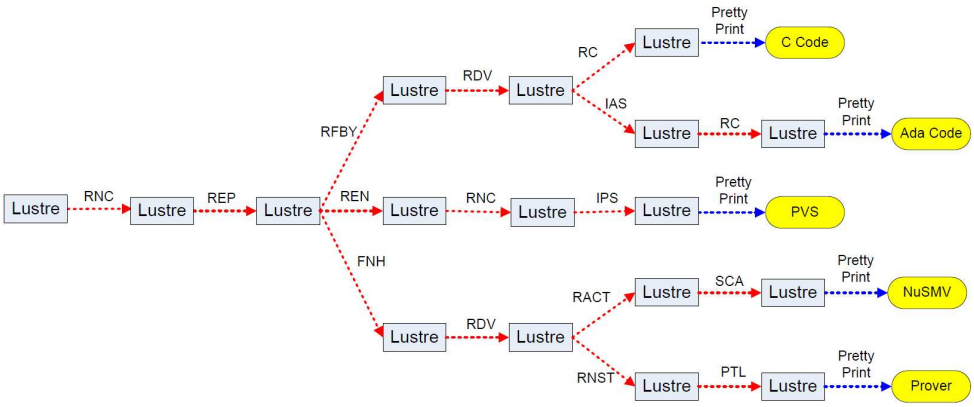


Figure 33: LUSTRE specification transformations [Cofer 2010]

The technique is validated in several case studies in the avionics [Cofer *et al.* 2008, Whalen *et al.* 2008, Miller 2009, Miller *et al.* 2010, Cofer 2010]. The first application of the NuSMV model checker to an actual product at Rockwell Collins is the mode logic of the FCS 5000 Flight Control System [Miller 2009]: 26 errors are found in the mode logic.

The largest and most successful application is the Rockwell Collins ADGS-2100 (*Adaptive Display and Guidance System Window Manager*), a cockpit system that provides displays and display management software for commercial aircrafts [Miller *et al.* 2010]. The Window Manager (WM) ensures that data from different applications are displayed correctly on the display panel. A set of properties that formally expresses the WM requirements (i.e., *functional* properties) is developed in the CTL and LTL temporal logics: 563 properties are developed and verified, and 98 design errors are found and corrected.

The approach is also applied to an adaptive flight control system prototype for unmanned aircraft modeled in Simulink [Whalen *et al.* 2008, Cofer 2010]. During the analysis, over 60 *functional* properties are verified, and 10 model errors and 2 requirement errors are found in relatively mature models.

Concerning the modeling coverage, the approach covers only the functional core of the avionics interactive systems [Cofer *et al.* 2008, Miller 2009, Miller *et al.* 2010], but not the user interfaces nor the user behavior.

These applications to the avionics domain demonstrates that the approach scales well. Even if the approach does not take user interfaces into account, it is a good example of formal methods applied to safety-critical systems. In addition, further investigations of the usage of

compositional verification are conducted [Cofer *et al.* 2012, Murugesan *et al.* 2013], to enhance the proposed techniques.

g) Knight *et al.* (USA, 1992–2010)

Another application of formal methods to safety-critical system, specifically, to the nuclear power plant domain, is described in [Knight & Brilliant 1997]. The authors propose the modeling of user interfaces in three levels: *lexical*, *syntactic*, and *semantic* levels. Different formalisms are used to describe each level. The notion of user interfaces as a dialog between the operator and the computer system consisting of three components (lexical, syntactic, and semantic levels) is proposed by [Foley & Wallace 1974]. Each level is specified separately:

- The *semantic* level contains the functionalities provided by the UI, more precisely, the meaning of the actions the UI performs or provides, not the form or sequence of those actions [Elder & Knight 1995]. This level is specified using Z;
- The *syntactic* level of the UI is the structure of the human-computer dialog. This structure is defined as a language with a grammar specifying the valid sequences of user inputs and computer outputs [Elder & Knight 1995];
- Finally, the *lexical* level specifies how the UI effects its dialog, i.e., which instruments and controls the user sees and employs to pilot the system [Knight & Brilliant 1997]. The graphical elements of the lexical level are defined using a predefined library of C++ classes.

Following this view of user interface structure, the authors develop a formal specification of a research reactor used in the University of Virginia Reactor (UVAR) for training nuclear engineering students, radiation damage studies, and other studies [Knight & Brilliant 1997]. In order to illustrate the specification layers, the authors focus on the safety control rod system, one of the reactor subsystems. They give in the paper the three specifications for this subsystem.

The approach is also applied to other safety-critical systems, such as the Magnetic Stereotaxis System (MSS), a healthcare application for performing human neurosurgery [Knight & Kienzle 1992, Elder & Knight 1995].

However, the formal specification is not used to perform formal verification. According to the authors, the main goal is to develop a formal specification approach for user interfaces of safety-critical systems. The authors evaluate the proposed framework according to seven criteria: expressiveness, usability, changeability, implementability, analyzability, verifiability, and accuracy. They rate each criterion subjectively, according to their own experience. Concerning verifiability, the authors claim that the verification of a UI specification using this approach is simplified by the use of an executable specification for the lexical level, and by the use of a notation from which an implementation can be synthesized for the syntactic level. For the semantic level, they argue that all the tools and techniques developed for Z can be applied [Knight & Brilliant 1997].

Later, a toolset called Zeus is proposed to support the Z notation [Knight *et al.* 1999]. The tool permits the creation and analysis of Z documents, including syntax and type checking, schema expansion, precondition calculation, domain checking, and general theorem proving. The tool is evaluated in a development of a relatively large specification of an international maritime software standard, showing that Zeus meets the expected requirements [Knight *et al.* 1999].

Following such a separation of concerns in three levels, the authors propose another approach called Echo [Strunk *et al.* 2005], this time applied to a case study in the avionics domain. In order to decrease complexity with traditional correctness proofs, the Echo approach is based on the refactoring of the formal specification [Yin *et al.* 2009a, Yin *et al.* 2009b], reducing the verification burden by distributing it over separate tools and techniques. The system model to be verified (written in PVS) is mechanically re-factored. It is refined into an implementable specification in Spark Ada by removing any unimplementable semantics. After refactoring, the model is documented with low-level annotations, and a specification in PVS is extracted mechanically [Yin *et al.* 2008]. Proofs that the semantics of the re-factored model is equivalent to that of the original system model, that the code conforms to the annotations, and that the extracted specification implies the original system model constitute the verification argument [Yin *et al.* 2009a].

An extension of the approach is proposed in [Yin & Knight 2010], aiming at facilitating formal verification of large software systems by a technique called *proof by parts*, which improve the scalability of the approach for larger case studies.

The authors did not clearly define the kinds of properties they can verify over interactive systems with their approach. The case studies to which the approach is applied mainly focused on the benefits of modeling UIs in three layers using formal notation.

UIs, users and the functional core of systems are covered by this approach. The UI *syntactic* level in their approach defines valid sequences of user inputs on the UIs, which is to some extent the modeling of the users, and the cypher system case study described in [Yin *et al.* 2008] verifies the correctness of the functional core.

2.6.3 Synthesis

This section presents a representative list of approaches to verifying interactive systems with respect to the specifications, i.e., general statements about the behavior of the system, which are represented here as desired properties, and analyzed afterwards using formal methods. The approaches diverge on the formalisms they use for the description of interactive systems and for the specification of properties. The propositions that model interactive systems globally provide a simple way to handle systems that are manageable as a whole. By contrast, the propositions that model interactive systems as a composition of parts provide a way to handle complex systems by breaking them into smaller manageable parts, which is more suitable for the complexity of safety-critical systems.

Some authors use theorem proving to perform verification, which is a technique that can handle infinite-state systems. Even though a proof done by a theorem prover is ensured to be correct, it can quickly become a hard process [Campos 1999]: the process is not fully automated, user guidance is needed regarding the proof strategy to follow. Simulation can also be used to assess the quality of interactive systems. Simulation provides an environment for training the staff before starting their daily activities. However, simulated environments are limited in terms of training, since it is impossible to drive operators into severe and stressful conditions even using a full-scale simulator [Niwa *et al.* 2001]. Simulation explores a part of the system state space and can be used for disproving certain properties by showing examples of incorrect behaviors. To the contrary, formal techniques such as model checking, equivalence checking, etc., consider the entire state space and can thus prove or disprove properties for all possible behaviors [Garavel & Graf 2013].

The presented approaches allow either *usability* or *functional* properties to be verified over the system models. We believe that in case of safety-critical systems, the verification approach should cover both such properties, due to the ergonomic aspects covered by the former and the safety aspects covered by the latter. Some approaches cover the modeling of the users, the user interfaces, and the functional core. None of them cover, however, plastic user interfaces. Besides, none of them was applied to the verification of nuclear-plant systems. Table 1 summarizes these approaches.

Table 1: Summary of approaches to verifying system properties

	Authors (*et al.*)	Modeling		Verification		Criteria					
		model language	property language	technique	tool support	modeling coverage			kind of properties	application	scalability
						users	UIs	core			
1	Abowd	SMV	CTL	model checking	SMV, Action Simulator	√	√	√	usability, functional	non-critical	no
2	Paterno	CTT, LOTOS	ACTL	model checking	CTTE, LITE, CADP	√	√	√	usability, functional	avionics	yes
3	Markopoulos	LOTOS	ACTL	model checking	CADP		√		usability, functional	non-critical	no evidence
4	Duke & Harrison	Z	first-order logic	theorem proving	Z	√	√	√	usability	avionics	no evidence
5	Campos	MAL, SMV, PVS	CTL	model checking, theorem proving	i2smv, IVY, SMV, PVS	√	√	√	usability, functional	avionic, healthcare	yes
6	D'Ausbourg	LUSTRE	LUSTRE	model checking	UIM/X, Centaur, Lesar	√	√	√	usability, functional	avionics	yes
7	Bumbulis	IL, HOL	Hoare logic	theorem proving	HOL system		√		functional	non-critical	no
8	Palanque	Petri nets	ACTL	model checking	PetShop, Java PathFinder	√	√	√	usability, functional	avionics	yes
9	Dix	mathematical notation	mathematical notation	-	none		√	√	usability	non-critical	no evidence
10	Bowen & Reeves	Z, μCharts, FSM	LTL/ invariants	model checking	PIMed, ProZ, Z/EVES	√	√	√	usability	healthcare	no evidence
11	Aït-Ameur	B method, Event B, EXPRESS	B	theorem proving	Atelier B, B2EXPRESS, Promela-SPIN	√	√	√	usability, functional	avionics	yes
12	Loer & Harrison	SMV	CTL, LTL	model checking	Statemate, IFADIS, Cadence SMV, NuSMV	√	√	√	usability	avionics	yes
13	Thimbleby	PVS, SAL	invariants, LTL	model checking, theorem proving	PVS, SAL, Stateflow		√	√	usability, functional	healthcare	yes
14	Miller	RSML, LUSTRE	CTL, LTL	model checking, theorem proving	NuSMV, PVS, Reactis, Gryphon			√	functional	avionics	yes
15	Knight	Z, PVS, SPARK Ada	-	theorem proving	Z, Zeus, Echo	√	√	√	-	avionics, nuclear, healthcare	yes

2.7 Assessing Consistency

Plasticity provides users with different versions of a UI which can vary at different levels, raising the need to verify consistency between such UI versions. We analyze different approaches to verify consistency of interactive system with another artifact, which can inspire us to propose an approach to verifying plastic UIs.

Assessing consistency consists in comparing an interactive system with another system artifact to check to which extent they are consistent. For instance, one can compare a previous version of the system with the most recent version, or a system against its user manual, or different versions of a system user interface. In such approaches, a referential does not exist a priori, contrary to the verification of properties over the system model, in which the properties are the referential. When analyzing consistency, we mainly searches for differences between both system artifacts. Once differences are found, analyses are needed to define whether such divergences indicate a flaw in the modeled systems or not.

In human-computer interaction, several approaches have been proposed to assess consistency. The following subsections describe a list of them. Interactive systems are compared either with another version of the system, or with their user manual.

2.7.1 System × System

In this subsection we present a number of approaches that use various techniques to compare different versions of interactive systems with each other. Specifically, when a user interface of these systems evolves, different techniques based on testing or formal methods are used to compare new versions of the UIs with their previous versions.

a) Jung *et al.* (Korea, 2012)

An image comparison approach is proposed in [Jung *et al.* 2012], to support regression testing of user interfaces. Their approach automates the detection of divergences between two versions of a user interface when the system evolves. This approach consists of two methods (Figure 34): an *event-driven testing* method, and a *capture and replay* method. In the former, events are sent directly to the target program, using test cases written in scripting languages. In the latter, test scripts are recorded and repeated when needed.

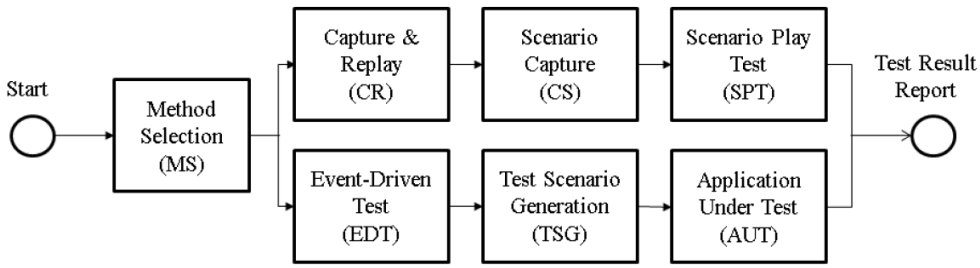


Figure 34: Automated UI testing process [Jung *et al.* 2012]

In the event-driven testing method, the tester extracts information by parsing an XML

script test. The tester must specify in the XML script the expected results of the interactions with the UI. The event-driven testing communicates directly with the target program. This is useful when the goal is to test the target program functionalities. When the focus is mainly to test the target program appearance, the capture and replay method is more useful. It does not communicate directly with the target program. The target program is analyzed based on the coordinates of graphical components and a comparison of the output images is the basis of the verification [Jung *et al.* 2012].

In the capture and replay method, a *capture start* button is used to record the actions of the tester. When the capture is finished, a *stop* button is pressed to create a script that can be further used to test new versions of the UI automatically (i.e., the *replay* step). To detect deviations between the old UI and the new UI, the tool provides image comparing-based validation using charts. When the test scenario requires output values on the UI, print screens of the old UI and the new UI are recorded before and after the execution of the scenario. Based on the coordinates of the UI visual components, comparison of the output images is used to verify divergences on the old UI and the new UI [Jung *et al.* 2012]. For each UI (i.e., the old UI and the new UI), a chart is generated with the coordinates of the UI visual components, representing the UI before (Figure 35a) and after (Figure 35b) the execution of the feature. The charts are then composed (Figure 35c), to highlight the differences on the UI before and after the execution of the feature. Such composed chart is generated for the old and the new UI, and both are compared to check for divergences between the UIs.

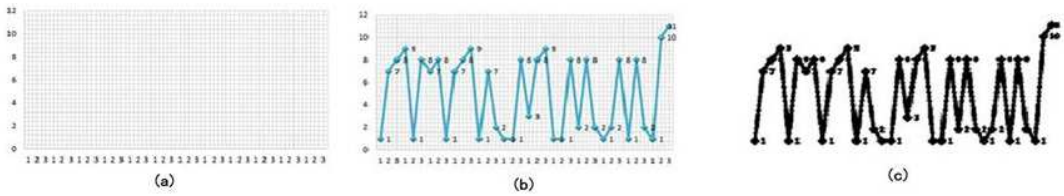


Figure 35: Example of a image comparison [Jung *et al.* 2012]

The approach covers mainly the analysis of the user interfaces of interactive systems. Nonetheless, the scenarios saved in the *capture* part of the approach can be seen as a representation of the user behavior. No attention is paid, however, to the functional core. There is no evidence that the approach scales well for larger case studies, since no other papers using the approach was found. Besides, no application of the approach to safety-critical systems was found.

Although these methods permit both functionality and appearance of the UIs to be verified, capture-and-replay tools require a great amount of manual effort from the tester, who needs to record and maintain input sequences [Bauersfeld 2013]. Especially in the context of frequently changing UIs, the scripts require high maintenance, since input sequences are significantly fragile to UI layout change. Such fragility can render entire automated test suites inept [Borjesson & Feldt 2012]. Instead of capture and replay, *Visual UI Testing* technique is used in the sequel. It also uses image recognition, the main difference being that it is less hardcoded than capture and replay to the UI elements positioning. Another approach to comparing UIs is proposed in [Bauersfeld 2013], in which the position of UI controls does not matter.

b) Bauersfeld *et al.* (Spain, 2011–2014)

The author of [Bauersfeld 2013] presents a regression testing library named GUIDiff to compare two versions of a UI and to produce a list of detected deviations. A tree representation is obtained from an operating system's accessibility API, and used to compare both UI versions (Figure 36). Each UI version is represented by a widget tree containing the visible UI controls (the tree nodes) and annotations with their property values.

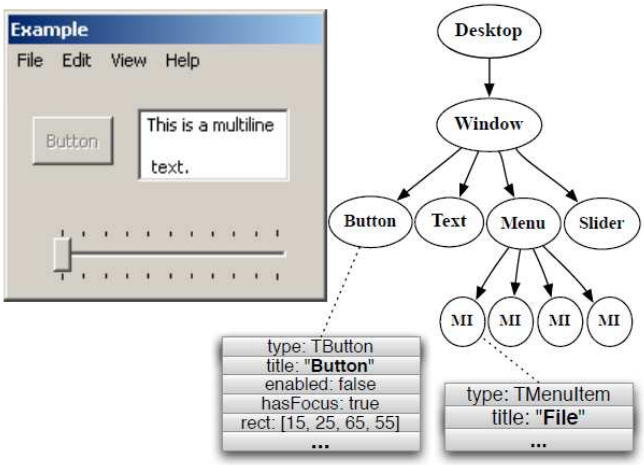


Figure 36: GUIDiff representation of the current state of a UI as a widget tree [Bauersfeld 2013]

GUIDiff can execute two different versions of a system in parallel and walk through the UIs, by iteratively selecting and executing particular actions derived by the widget trees. GUIDiff uses GUITest [Bauersfeld & Vos 2012] to obtain and to execute the possible actions on the UI versions. GUITest is a Java library that generates test sequences automatically by randomly selecting possible UI actions in order to drive the tests. The idea is to execute the same actions in both UI versions, and observe the differences in the widget trees of their states [Bauersfeld 2013].

The tool is semi-automatic in the sense that it automatically finds differences and the tester should label them as actual deviations or false positives [Bauersfeld 2013]. In order to reduce false positives, the author recommends to take into account only the same UI controls, ignoring controls introduced in the most recent version of the UI.

In order to improve the selection of possible actions on the UIs, the authors apply an ant-colony algorithm [Bauersfeld *et al.* 2011a, Bauersfeld *et al.* 2011b]. At present, this approach is implemented for Java SWT applications only. An alternative to improve the selection of possible actions on the UIs is proposed in [Bauersfeld & Vos 2014], in which a tool called Rogue/TESTAR (*TEST Automation at the useR interface level*) [Bauersfeld *et al.* 2014a] uses a Prolog specification to derive sensible and sophisticated action sequences.

The approach is applied to several case studies [Bauersfeld *et al.* 2014a, Bauersfeld *et al.* 2014b] and it is well supported by tools that have been themselves positively evaluated by users [Bauersfeld *et al.* 2014b]. However, there is no evidence that this work scales well to larger applications. Moreover, no application of the approach to safety-critical systems have been reported.

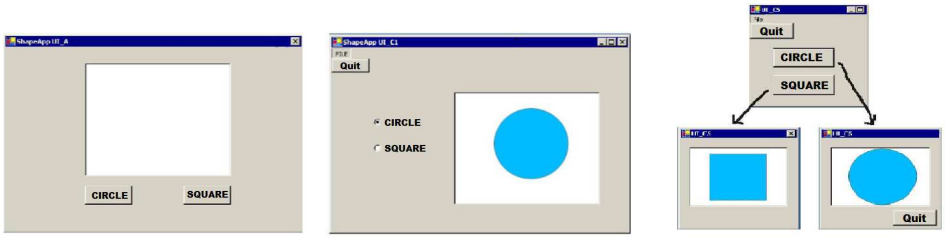
Besides, this approach only covers the user interfaces, specifically, their appearance. No

analysis has been carried out in which the functional core and/or users would have been modeled. This approach allows the comparison of different versions of a UI, and the identification of the UI divergences in a fully automatic way. Even though, leaving the newly introduced UI controls out of the analysis (to reduce false positives) limits the UI components that are covered by the approach. Besides, the approach is not adapted to plastic user interfaces: only UIs relatively similar to each other can be compared.

c) Bowen and Reeves (New Zealand, 2005–2008)

A formal approach to compare different user interfaces is proposed in [Bowen & Reeves 2006, Bowen & Reeves 2008b]. Specifically, the authors propose an approach to defining whether a user interface is a *refinement* of another user interface. Refinement relates to the ability to move between different implementations of a system without having any negative impact on a user's view of the system in terms of functionality or usability [Bowen & Reeves 2006].

The authors of [Bowen & Reeves 2005] discuss practical ways to include design guidelines in the formal modeling of user interfaces in order to ensure that one UI is a good refinement of another one. The approach requires the UIs to be represented by formal models, which in this case are the *presentation model (PM)* and *presentation and interaction model (PIM)* (the same models introduced in Subsection 2.6.2 - b), page 39). To grasp how the authors verify refinement, consider the UIs illustrated in Figure 37, which allow a user to display two different shapes.



(a) Design UI_A (b) Design UI_C (c) Design UI_{C5}

Figure 37: Example of user interfaces for displaying shapes [Bowen & Reeves 2008b]

Four criteria should be satisfied for the UI_C to refine the UI_A [Bowen & Reeves 2008b]:

1. One can substitute UI_C for UI_A and maintain *contractual utility*, i.e., when UI_C is substituted for UI_A, the new UI needs to provide at least all of the functionality of the previous UI. For the example illustrated in Figure 37, the presentation model of the UI_A and UI_C are illustrated in Figure 38.

   For each UI, the following behavior sets are derived from the presentation models (Figure 39): the set S_Beh represents the system functionality made available via the UI, and the set I_Beh represents the UI functionalities. From these sets, we observe that the sets $S_Beh[UI_A]$ and $S_Beh[UI_C]$ are equal and that $I_Beh[UI_A] \subseteq I_Beh[UI_C]$. Therefore, in that respect, UI_C might be considered a correct refinement of UI_A.

2. The widgets of UI_C are not more abstract than those of UI_A. In order to verify that, the authors rely on a widget category hierarchy that describes widgets in terms of the

UI_A is	(CircleButt, ActCtrl, (S_ShowCircle)), (SquareButt, ActCtrl, (S_ShowSquare)), (ShapeFrame, SValRspndr, (S_DisplayShape)), (QuitButt, ActCtrl, (I_QuitApp))
UI_{C1} is	(CircleRB, RadioButton, (S_ShowCircle)), (SquareRB, RadioButton, (S_ShowSquare)), (ShapeFrame, SValRspndr, (S_DisplayShape)), (FileMenu, Container, ()), (QuitMenuItem, ActCtrl, (I_QuitApp)), (QuitBox, ActCtrl, (I_QuitApp)), (MinBox, ActCtrl, (I_MinWindow)), (MaxBox, ActCtrl, (I_MaxWindow))

Figure 38: PMs of the user interfaces [Bowen & Reeves 2008b]

$$S_Beh[UI_A] = \{S_ShowCircle, S_ShowSquare, S_DisplayShape\}$$
$$I_Beh[UI_A] = \{I_QuitApp\}$$
$$S_Beh[UI_C] = \{S_ShowCircle, S_ShowSquare, S_DisplayShape\}$$
$$I_Beh[UI_C] = \{I_QuitApp, I_MinWindow, I_MaxWindow\}$$

Figure 39: Behavioral sets of the user interfaces, adapted from [Bowen & Reeves 2008b]

kind of behavior they exhibit. For a widget description, becoming less abstract means moving down the relevant hierarchy tree from the current position. In the example given in Figure 37, UI_A has standard buttons whereas UI_C has radio buttons. In the widget category tree, as both of these are examples of `ActionControls`, UI_C might be considered a correct refinement of UI_A [Bowen & Reeves 2008b].

3. The layout and appearance of UI_C is not less defined than those of UI_A. Unfortunately, the PM and PIM formal models cannot be used to check that at this stage of the work [Bowen & Reeves 2008b].

4. The usability of UI_C is not less than that of UI_A. In order to ensure that, the authors examine some of the conditions on the PIM models, such as reachability and absence of deadlocks. If one has a UI that produces a PIM strongly connected (i.e., any state can be reached from any other state) and no deadlock, then it is expected that these properties are preserved in the PIM of the new UI [Bowen & Reeves 2008b]. The PIMs for both UI_A and UI_{C5} from Figure 37, are given in Figure 40. The PIM for UI_A consists of a single state (since Figure 37a contains a single UI) and so it has strong reachability and no deadlock. The PIM for UI_{C5} has three states, one for each UI of Figure 37c. This PIM has no deadlock either, however, it no longer has strong reachability: it is not possible to reach state 2 from state 3. UI_{C5} does not maintain the usability of UI_A and is, therefore, not a suitable refinement [Bowen & Reeves 2008b].

Such notion of refinement enables a formal-based reasoning to compare different versions of a UI. Refinement as defined here, differs from plasticity in the sense that the user is not

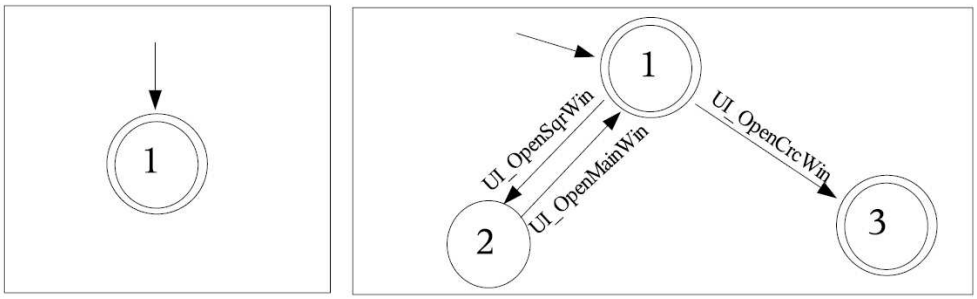


Figure 40: PIMs for UI_A and UI_{C5} [Bowen & Reeves 2008b]

supposed to observe that a user interface has changed [Bowen & Reeves 2006], whereas this is permitted in plastic user interfaces. Besides, in refinement, the adaptation and the choice of the most adaptable UI occurs before the execution of the system. By contrast, in plasticity the adaptation can either occur at design time (and the most suitable UI is chosen at runtime), or the UI is adapted according to changes in the context of use at runtime.

Concerning the modeling coverage, although the approach mainly focuses on modeling the user interfaces, the presentation model can also be used to model the user operations, the main goal being to ensure that all user operations described in the specification have been described in the UI design [Bowen & Reeves 2007a]. Besides, as described before, the S_Beh behavior set derived from the presentation models represents the system functionality made available via the UI. So, the approach covers the modeling of the user, UIs, and functional core.

However, the verification of UI refinement is not automated and requires manual inspection of the UI models. This gives no evidence that the approach scales well to larger applications. Finally, no applications to safety-critical systems have been reported.

2.7.2 System × User Manual

Consistency verification approaches can also be used to verify if users correctly understand interactive systems. For instance, if users are aware of all system functionalities, and if the system behaves how the user expects. The expected behavior of a system can be analyzed though its user manual, since users can learn how to interact with the system by reading the user manual or through targeted training sessions [Chinnapongse *et al.* 2009]. In either case, users develop a *mental model* of the system. Analyzing this mental model helps to avoid usability problems in the system, and may help to evaluate necessary user training and instruction materials that accompany a given device [Chinnapongse *et al.* 2009].

a) Chinnapongse *et al.* (USA, 2009)

The authors of [Chinnapongse *et al.* 2009] model the expected behavior of a system by analyzing the specifications/user manual of the system (Figure 41). The mental model is manually created using the NModel framework [Jacky *et al.* 2007], a Microsoft model-based testing tool for C♯ programs. Using the *model program viewer* (mpv) tool from the NModel framework (Figure 41), the mental model is used to generate a finite-state machine, in which states are identified with

UIs, and transitions represent changing UIs in response to invoking UI elements [Chinnapongse *et al.* 2009].

From this mental model, a model-based approach is used to ascertain compliance of the system to the mental model. The mental model can be used to generate a test suite (using the *off-line test generator* tool from the NModel framework in Figure 41). This test suit is then applied to the system implementation (using the *conformance tester* tool from the NModel framework in Figure 41). The application to the system implementation must be coupled to the implementation by means of a test harness (*stepper*) which invokes an instance of the implementation to be tested and causes the appropriate actions to be executed when invoked by the *conformance tester*. Finally, the *conformance tester* is executed with the test suite and the implementation coupled with the stepper to check for consistency between the implementation and the mental model [Chinnapongse *et al.* 2009].

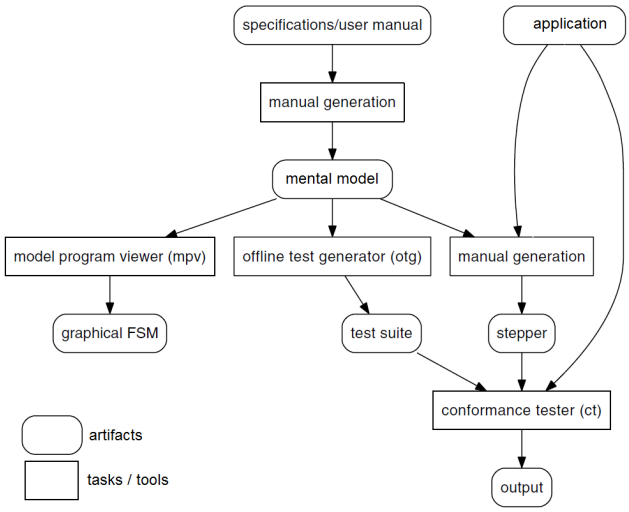


Figure 41: Comparing systems with their user manuals, adapted from [Chinnapongse *et al.* 2009]

The approach is applied to a UI-driven handheld device in the healthcare domain used to assist personnel with diagnosis and treatment of patients. An inconsistency between the observed behavior and the behavior described in the user manual is demonstrated. However, no further application to larger case studies is reported, which makes it hard to conclude whether it scales well or not to larger real-life applications.

The authors suggest an alternative to improve the approach, by extracting also a state-machine model from the application source code and compare it to the mental model using the notions of state machine equivalence or pre-order [Chinnapongse *et al.* 2009]. This improvement is interesting in the context of plastic user interfaces: one could use a state-machine representation for each version of the UI and verify the equivalence between them.

One limitation of this work is that it focuses on UI navigation (transitions between the UIs). The interaction capabilities of the user interfaces (i.e., the actions the user can perform within each user interface that are not related to the display of another UI) are not analyzed. Concerning the modeling coverage, the approach covers the modeling of the user behaviors and the UIs: the finite-state machine representing the user manual, which is a mental model of the

users. The functional core is not modeled. Instead, the real system is tested using the test cases generated by the model.

b) Bowen and Reeves (New Zealand, 2012)

In [Bowen & Reeves 2012], formal methods are applied to compare a medical device against its respective user manual. The device under study, a syringe pump, is designed to deliver the contents of a syringe to a patient in a controlled manner at a specified rate. Among its functionalities, the paper focuses on the delivery of a prescribed dosage of medication of the pump. The syringe pump has nine widgets: eight soft-keys and a display (Figure 42), all of each are fixed and always present in all modes of the pump. The formal models used in this case study, called *presentation model (PM)* and *presentation and interaction model (PIM)* have been already presented in Subsection 2.6.2 - b), page 39. For each mode of the pump there is a single presentation model, and each presentation model contains descriptions of the nine widgets, but with different behaviors [Bowen & Reeves 2012].

Figure 42: The Niki T34 Syringe pump [Bowen & Reeves 2012]

Both the pump and the manual are modeled in the same manner, by identifying the modes and widget behaviors within those modes and describing them in different models [Bowen & Reeves 2012]. Concerning the modeling coverage, in this case study the operations of the functional core are included in the formal model; however, no references are made to the UIs or the users, since the authors do not argue that the user manual should serve as a user model.

The PIMed tool [Bowen & Reeves 2007b] is used to validate the models, as well as the ProB and ZOOM tools. PIMed can automatically generate a set of abstract tests from a model (either the model built from the manual, or the one built from inspecting the pump). These tests are then manually instantiated, and one checks that the manual or the pump (depending on which model the test relates to) passes the tests. If a test does not pass, a possible error in the associated model is revealed [Bowen & Reeves 2012]. This technique detected several divergences between the models.

This work illustrates an example of usage of formal methods to compare two artifacts of a system: the system and its user manual. Contrary to [Chinnapongse *et al.* 2009], which formalizes only the user manual, this work shows that consistency verification approaches can

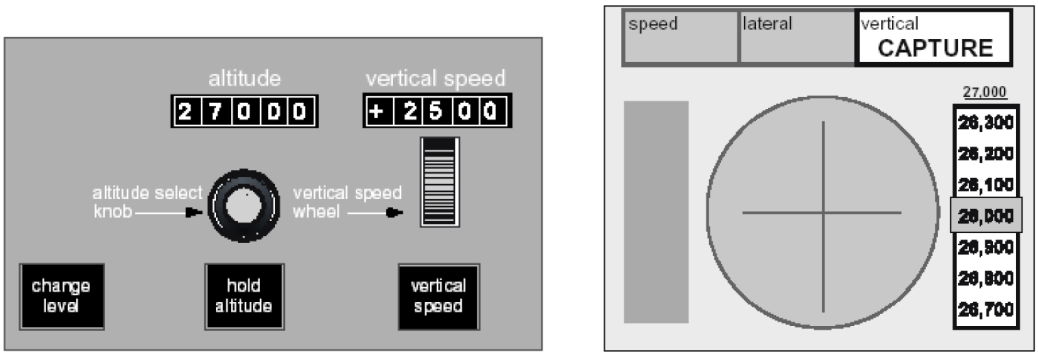
be supported by formal methods.

However, the comparisons described in the paper are not automated. The authors argue that they did not need more detailed and mathematical techniques to discover the model inconsistencies, as manual inspection already revealed problems [Bowen & Reeves 2012]. To scale to larger case studies, such manual inspection would not be enough.

c) Degani *et al.* (USA, 1996–2013)

In the avionics domain, Degani *et al.* propose another approach to comparing systems with their user manual. In flight control systems, a variety of different behaviors, or modes, are employed. In this context, mode confusion is the difference between the actual machine behavior and what it is expected by the pilot [Degani *et al.* 1996]. Several incidents (some fatal) involving mode confusion are observed and documented [Degani *et al.* 1996, Degani & Heymann 2000, Degani *et al.* 2000, Degani & Heymann 2002, Degani *et al.* 2013].

One example of mode confusion implicating a specific autopilot and the changes in its altitude settings is described and modeled in Figure 43a. It illustrates the control panel through which the pilot interacts with the autopilot. Figure 43b illustrates the display through which the pilot obtains information about several system's behavior [Degani & Heymann 2002].



(a) Guidance and Control Panel (b) Attitude Display Indicator

Figure 43: Mode-control panel [Degani *et al.* 2000]

This autopilot presents an unexpected behavior when engaging in the "capture" mode. This is a mode to which the autopilot automatically transits (with no explicit command from the pilot), to maneuver the aircraft to a gentle transition from climb or descent to the target altitude. The issue is that the altitude at which the autopilot engages the "capture" mode varies, and the pilot has no pre-knowledge of this altitude.

This is an example of mode confusion: the actual system behavior diverges from the user expectation of the system. An approach to comparing both a system model and a user model is proposed in [Degani *et al.* 2013], in which both the system model and the user model are finite-state machines (Figure 44). The system model is created by analyzing the actual system, and the user model is created by analyzing the user manuals. A composite model containing the synchronous product of both state machines is computed, and an analysis (detailed in the following) aiming at identifying *error states* in this composite model is conducted. An *error state* is a state-pair (i.e., a user-model state and a system state) composed by a legal and an illegal

state, and it indicates that the user model does not correctly reflect the system model [Degani & Heymann 2002].

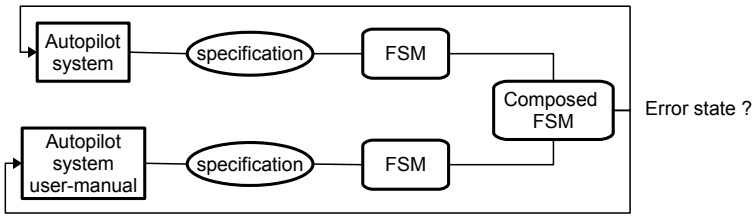


Figure 44: Degani *et al.*'s approach based on the composition of finite-state machines

The authors of [Degani *et al.* 2013] propose a methodology for generating correct user models for a modeled system, e.g., free of error states. Such user model is further used to generate simple (yet adequate) user interfaces for such systems. A web-based tool has been developed (UIverify [Shiffman *et al.* 2004]) to support the approach. It accepts as input a model of the system and a mental model, checks whether the mental model is adequate, and generates a correct mental model, which can be used to generate correct user interfaces. Note that the input models of the tool (the finite-state machines) have to be manually created, which is an error prone task for real-life applications.

Several papers provide a well-structured theoretical work about how to use finite-sate machines to model the system's behavior, task specification, the required user interface, and the user-model [Degani *et al.* 1996, Degani *et al.* 2000, Degani & Heymann 2000, Degani & Heymann 2002, Heymann & Degani 2002, Degani & Heymann 2007, Heymann & Degani 2007, Degani *et al.* 2013]. The goal is to ensure that a correct and unambiguous interaction between the user and the machine is possible.

Although the approach revealed a severe flow in the design of a complex safety-critical system for avionics, it is not automated. The authors points out that for larger systems an automatic tool is desirable [Degani *et al.* 2013], which indicates a scalability issue.

Concerning the modeling coverage of the approach, the finite-state machine (modeled using the Statecharts language) contains modules describing: the environment, the pilot functions/-tasks, the controls of the user interfaces, the displays, and the modes of the autopilot [Degani *et al.* 1996], covering the three aspects we consider: user interfaces, functional core, and users.

d) Rushby *et al.* (USA, 1999–2002)

Rushby *et al.* have been investigating the usage of formal methods in industrial safety-critical systems, specially avionics systems, since the 80s. However, only in the 90s the focus starts to be shifted to the analysis of the system behavior and the operators *mental model* [Rushby *et al.* 1999, Crow *et al.* 2000]. In [Rushby 2001], an approach based on model checking is applied to the same case study described in [Degani *et al.* 2000], to analyze the autopilot UI model and the pilots' mental model. Both were analyzed with the Murϕ model checker, the expected behavior of the system being described using invariants (Figure 45).

The approach is applied to several autopilot case studies [Rushby *et al.* 1999, Crow *et al.* 2000, Rushby 2001, Rushby 2002], in which issues on real-life systems are identified. Such evidences indicate that it scales well.

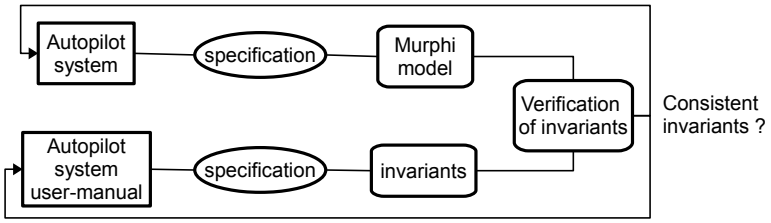


Figure 45: Rushby *et al.*'s approach based on invariant verification

Concerning the modeling coverage, the approach covers all the three aspects we are analyzing: the model specifies actions by the pilot (i.e., the *user*) on the *user interface* (e.g., pressing a button, or changing the value set by a dial), actions corresponding to the dynamics of the aircraft (i.e., the *functional core*, such as a change in the current altitude of the aircraft), and finally those performed by the autopilot system in response to certain events.

Both Degani's and Rushby's approaches have shown that formal methods can be used to compare two artifacts of the system, helping to detect divergences between them, which is interesting in the context of plastic user interfaces.

e) Combéfis *et al.* (Belgium, 2009–2013)

[Combéfis 2013] also proposes a formal framework for reasoning over system and user models, and the user models can be also extracted from user manuals. Furthermore, his work also proposes the automatic generation of user models. Using his technique, "adequate" user models can be generated from a given initial user model. Adequate user models capture the knowledge that the user must have about the system, i.e., the knowledge needed to control the system, using all its functionalities and avoiding surprises. This generated user model can be used, for instance, to improve training manuals and courses [Combéfis & Pecheur 2009].

In order to compare the system and the user model, and to verify whether the user model is adequate to the system model, both models should be provided. With this goal, in this approach system and user are modeled with enriched labeled transition systems called HMI LTS [Combéfis *et al.* 2011a]. An LTS represents a system by a graph composed of states and transitions between states. Transitions between states are triggered by actions, which are represented in LTS transitions as labels. Intuitively, an LTS represents all possible evolutions of a system modeled by a formal model.

The enhancements brought by HMI LTS is that three kinds of actions are defined [Combéfis *et al.* 2011a] (Figure 46):

1. *Commands*: actions triggered by the user on the system;
2. *Observations*: actions triggered by the system, but that the user can observe;
3. *Internal actions*: actions that are neither controlled nor observed by the user.

To be considered "adequate", user models are expected to follow two specific properties: *full-control* and *mode-preserving*. Intuitively, a user model allows *full control* of a system if at any time, when using the system according to the user model [Combéfis & Pecheur 2009]: the

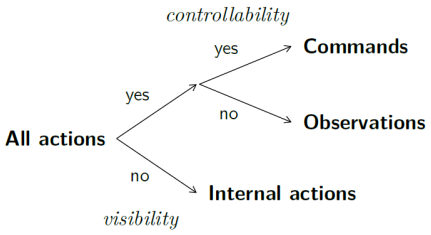


Figure 46: Classification of actions for human-machine interactions [Combéfis 2013]

commands that the user model allows are exactly those available on the system; and the user model allows at least all the observations that can be produced by the system [Combéfis & Pecheur 2009]. A user model is said to be *mode-preserving* according to a system, if and only if, for all possible executions of the system the users can perform with their user model, given the observation they make, the mode predicted by the user model is the same as the mode of the system [Combéfis & Pecheur 2009]. Model-checking is used to verify both properties over the user model.

In order to automatically generate adequate user models, the authors propose a technique based on a derivative of weak bisimulation, in which equivalence checking is used [Milner 1980]. This is called "minimization of a model modulo a equivalence relation". Intuitively, using equivalence checking, they generate a user model U_2 from the initial user model U_1, i.e., U_2 is equivalent to U_1 with respect to specific equivalence relations introduced by the authors.

Two equivalence relations are proposed: *full-control equivalence* and *mode-preserving equivalence*. Full-control equivalence distinguishes commands and observations: two equivalent states must allow the same set of commands, but may permit different sets of observations. Minimization modulo this equivalence produces a minimal user model that permits full-control of the system. A mode-preserving equivalence is then derived from the full-control equivalence, by adding an additional constraint that the modes of two equivalent states must be the same [Combéfis & Pecheur 2009]. Using these equivalence relations, the authors can generate mode-preserving-fully-controlled user models, which can then be used to design user interfaces and/or training manuals. Both properties (i.e., *mode-preserving* and *full-control*) and their combination are interesting because they propose that different levels of equivalence can be shown between system models.

A tool named `jpf-hmi` has been implemented in Java and uses the JavaPathfinder model checker [Combéfis *et al.* 2011a], to analyze and generate user models. The tool takes as input a system and a user model, and can verify full-control and mode-preserving properties. The tool produces an LTS corresponding to one minimal fully-controlled mental model, or it reports that no such model exists by providing a problematic sequence from the system [Combéfis *et al.* 2011a].

The approach is applied to several examples that are relatively large [Combéfis *et al.* 2011b]. In the healthcare domain, a machine that treats patients by administering X-ray or electron beams is analyzed with the approach, which detects several issues in the system. In the avionics domain, the approach is applied to an autopilot system of a Boeing airplane [Combéfis 2013], and a potential mode confusion is identified. These are evidences that the approach scales well to real-life applications.

Concerning the coverage modeling of the approach, the users, the user interfaces and the functional core are modeled and compared to each other, the user model being extracted from the user manual describing the system.

2.7.3 Synthesis

This section provides a representative list of approaches to assessing the consistency of interactive systems with another artifact. Interactive system are compared either with another version of the system or with its user manual. Table 2 summarizes these approaches. The comparison between the system and its user manual ensures the consistency between both. Since the user manual is one of the tools used in the staff training, it is crucial that it represents correctly the system, specially in safety-critical systems. From these work, we retain one of the formal technique used: equivalence checking, which is very suitable for the comparison of user interfaces.

However, the approaches that compare two versions of a system apply either model-based testing or manual inspection as analysis techniques. Manual inspection does not scale for large real-life systems. Model-based testing is more promising than manual inspections. Testing activities have significantly increased over the years, specially in safety-critical systems [Lex & Powej 1997]. However, it has been shown that testing is not a sufficient method for such systems [Butler & Finelli 1993]. The efficiency of testing highly relies on its coverage, and test coverage is never exhaustive. Test cases often emphasize boundary conditions (e.g., startup, shutdown) or anomalous conditions (failure detection and recovery), since hazards can result from improper handling of these vulnerable states [Lutz 2000]. Besides, test-based techniques require a runnable version of the system under test, which pushes testing to begin often late in the system development cycle, when it is costly to find flaws in the system. Alternatively, approaches based formal techniques bypass the need of a runnable version of the system, they can be performed earlier in the development cycle, and they allow an exhaustive analysis of the system model.

Table 2: Summary of approaches which assess consistency

	Authors (*et al.*)	Modeling			Verification		Criteria				
		artifacts	language 1st artifact	language 2nd artifact	technique	tool support	modeling coverage			application	scalability
							users	UIs	core		
1	Jung	system×system	image	image	model-based testing	a prototype	√	√		non-critical	no evidence
2	Bauersfeld	system×system	tree	tree	model-based testing	GUITest, GUIDiff, Rogue/ TESTAR		√		non-critical	no evidence
3	Bowen and Reeves	system×system	Z, μCharts, FSM	Z, μCharts, FSM	manual model inspection	-	√	√	√	non-critical	no evidence
4	Chinnapongse	system×user manual	-	FSM	model-based testing	NModel	√	√		healthcare	no evidence
5	Bowen and Reeves	system×user manual	Z, μCharts, FSM	Z, μCharts, FSM	manual model inspection	PIMed, ProB, ZOOM			√	healthcare	no
6	Degani	system×user manual	FSM, Statecharts	FSM	manual model inspection	UIVerify	√	√	√	avionics	no
7	Rushby	system×user manual	Murϕ	Murϕ	model checking	Murϕ	√	√	√	avionics	yes
8	Combéfis	system×user manual	LTS	LTS	model checking, equivalence checking	jpf-hmi, Java Pathfinder	√	√	√	avionics, healthcare	yes

2.8 Summary

A representative list of approaches to assessing the quality of interactive systems is presented in this chapter, divided in two main classes: property verification approaches, in which a set of properties are verified over the system model, and approaches to assessing consistency, which compare the system with another artifact.

For the first class, we present several approaches that represent systems as a composition of parts, which inspire us on how to manage complex systems, and several approaches that do not model systems as compositions, which inspire us as an alternative to model systems that are manageable without the need to break them in smaller parts.

For the second class, i.e., approaches to assessing consistency, we present approaches that compare different versions of the system, or the user interfaces, which inspire us on how to compare different versions of plastic UIs, followed by approaches that compare the system with its user manual, which inspire us on how to model the users and their mental model of the system.

Different formalisms are used in the system modeling (and property modeling, when applied). Numerous case studies have shown that each formalism has its strengths. The criteria to choose one over another would be more related with the knowledge and experience of the designers in the formalisms. Different formal techniques are employed, such as model checking, equivalence checking and theorem proving. Most of the work presented here are tool supported, even though some authors still use manual inspection of the models to perform verification.

This study of the related work shows that:

- No approach so far verifies or compares plastic user interfaces. Investigations of both plasticity and formal verification of interactive systems started in late 80s, thus, plastic UIs were not mature enough to be used as case study in the aforementioned approaches.

- No approach considers the verification of nuclear power plant systems with users, user interfaces and functional core aspects integrated, while considering usability and functional properties. Specifically, only one approach [Knight & Brilliant 1997] is applied to interactive systems in the nuclear-plant domain (Subsection 2.6.2 - g), page 50), but the formal specification is not used to perform formal verification.

Chapter 3

The Nuclear Reactor Supervision Case Study

Contents

3.1 Goals **69**
3.2 Nuclear-Plant Control Rooms **69**
3.3 The EDF Prototype **72**
3.3.1 *Global Synthesis* User Interface 72
3.3.2 Reactor Parameters 72
3.3.3 Failure Signals 74
3.4 The LIG Prototype **76**
3.4.1 The Control Room Version 76
3.4.2 Plastic Versions 78
3.5 The ADACS-N^{TM} Prototype **80**
3.6 Summary **82**

3.1 Goals

In the context of the Connexion Project[1], several case studies were proposed to validate the technical propositions of the project, some case studies being shared among different partners. The case study to which the work of this thesis was applied is a system designed to support monitoring activities in a nuclear-plant control room. This chapter describes this case study.

3.2 Nuclear-Plant Control Rooms

In the overall architecture of a nuclear unit plant, a control room system is integrated with several other systems, which are layered in several levels (Figure 47): level 0 consists of the instrumentation, in which actuators and sensors perform measurements (e.g., pressure, temperature, flow) and transform physical data into electrical signals that can be handled by PLCs (*Programmable Logic Controllers*). At level 1, the PLCs interact with the physical devices to ensure that the input data coming from level 0 remain within the limits allowed by safety; the PLCs trigger actions in case of disturbances. Level 1 interfaces with level 2, in which the control room is placed. In the control room, operators are provided with several terminals, control panels, and backup panels, to manage the plant. Finally, level 2 integrates with level 3, in which systems external to the nuclear unit process data provided by level 2 [Worldgrid 2011b].

A control room permits a centralized operation of the nuclear unit, in contrast to previous generations of nuclear power plants, in which operators were distributed all over the nuclear

[1] `http://www.cluster-connexion.fr/`; see Section 1.8 on page 10

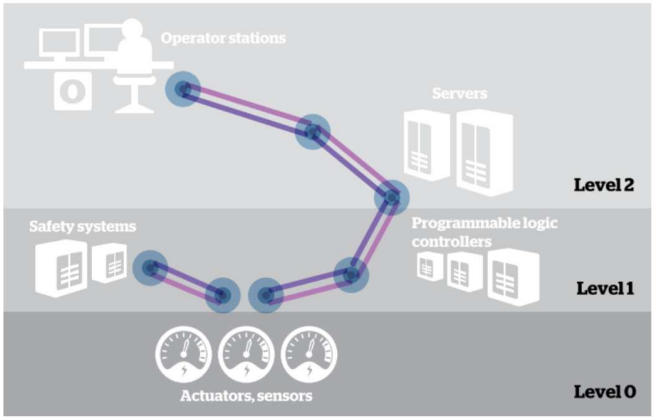


Figure 47: Overall architecture of a control room system [Worldgrid 2011b]

unit [Connexion 2012]. The RCC-E book [AFCEN 2012] defines a control room as "the whole of electrical systems allowing to assure the monitoring and control of a nuclear unit". *Conventional* control rooms are mainly composed of large panels as the one illustrated in Figure 48, where synoptics represent the installation, and where analog devices such as gauges, valves, pumps, and dials, etc., permit operations on the nuclear unit.

Figure 48: A conventional nuclear power plant control room

Recent nuclear plants have *computerized* control rooms, which allow a fully operation (e.g., display of information and anomalies, execution of commands and procedures, etc.) of the

nuclear unit by computer systems. Figure 49 illustrates an example of computerized control room with four workstations, in which:

- each operator is provided with seven monitors: three monitors for operation, and four monitors for displaying alerts and alarms;
- on the walls, large-size conventional displays are used to show measurements of physical functions. Besides the mimics, some information displayed on the UIs on the operators' workstations can also be displayed on such wall-size conventional displays, permitting relevant information to be shared between all operators in the control room [Connexion 2012];
- a conventional panel (behind the wall-size conventional displays) is used when the computerized systems fail. Information coming from the nuclear unit are also sent to the conventional panel.

Figure 49: A computerized nuclear power plant control room

In such environment, several actors play different roles in the nuclear unit:

- **Control-room operators**: They are constantly in operation, being responsible for monitoring and operating the nuclear reactor units. The monitoring task consists in verifying the state of units and comparing them to referential information, in order to identify the reason of eventual gaps. After the diagnosis of such gaps, immediate interventions can be made from the control room;
- **Operation supervisor**: He is responsible for the nuclear unit. Disposing of significant mobility, the operation supervisor moves between two control rooms, participates to management meetings, and has an office outside the control room for the management tasks;
- **Safety engineer**: He is responsible for guaranteeing the safety of the nuclear unit. He has a workstation inside the control room. In case of an accident, the safety engineer uses the conventional panel to monitor the safety of the plant.

The case study of this thesis concerns one of these actors: *control room operators*. From now on, we refer to control-room operators as "users".

3.3 The EDF Prototype

Numerous interactive systems assist users in their daily activities of monitoring the nuclear unit. In the context of the Connexion Project, common case studies were proposed to several partners, allowing them a multi-faceted exploration of real-life settings to demonstrate their contributions within the project. Our research laboratory, together with Atos Worldgrid[2] and EDF[3] (*Électricité de France*), conducted in-depth investigation of one EDF system that implements several control room activities. The main goal of the system is to provide a general overview of the plant status, warning the user about anomalies regarding the plant before an alarm occurs (to prevent in the monitoring) [Chériaux *et al.* 2012]. The main user interface (UI) of the system is called *Global Synthesis* (Figure 50). The system contains other UIs with different features not covered in our thesis, such as UIs that display synoptics of the nuclear unit.

3.3.1 *Global Synthesis* User Interface

The Global Synthesis UI is illustrated in Figure 50 (in French). On the top line of the user interface, six tabs indicate the current status of the plant. These status ranges from RP (working at full capacity) to RCD (completely stopped):

1. RP (*Reactor in power* – "Réacteur en production");
2. AN/GV (*Normal stop on steam generator* – "Arrêt normal sur générateur de vapeur");
3. AN/RRA (*Normal stop on cooling system* – "Arrêt normal sur circuit de refroidissement");
4. API (*Stop by intervention* – "Arrêt pour intervention");
5. APR (*Stop for reloading* – "Arrêt pour rechargement"); and
6. RCD (*Reactor discharged* – "Réacteur complètement déchargé").

These status indicate the current operating mode of the reactor. The transitions between these six status take place either in the physical unit first (and the system detects this change and updates the user interface), or if the user changes it on the user interface (affecting the physical status of the plant).

3.3.2 Reactor Parameters

Depending on the plant status, different reactor parameters are displayed in the middle part of the UI (for instance, the reactor average temperature, the average thermal power, etc. Appendix A gives a full list of these parameters). The Global Synthesis UI (Figure 50) groups the reactor parameters that should be constantly monitored. Other parameters are dispersed in other UIs. Each parameter is represented by a widget like the one of Figure 51.

[2] `http://fr.atos.net`
[3] `https://www.edf.fr/`

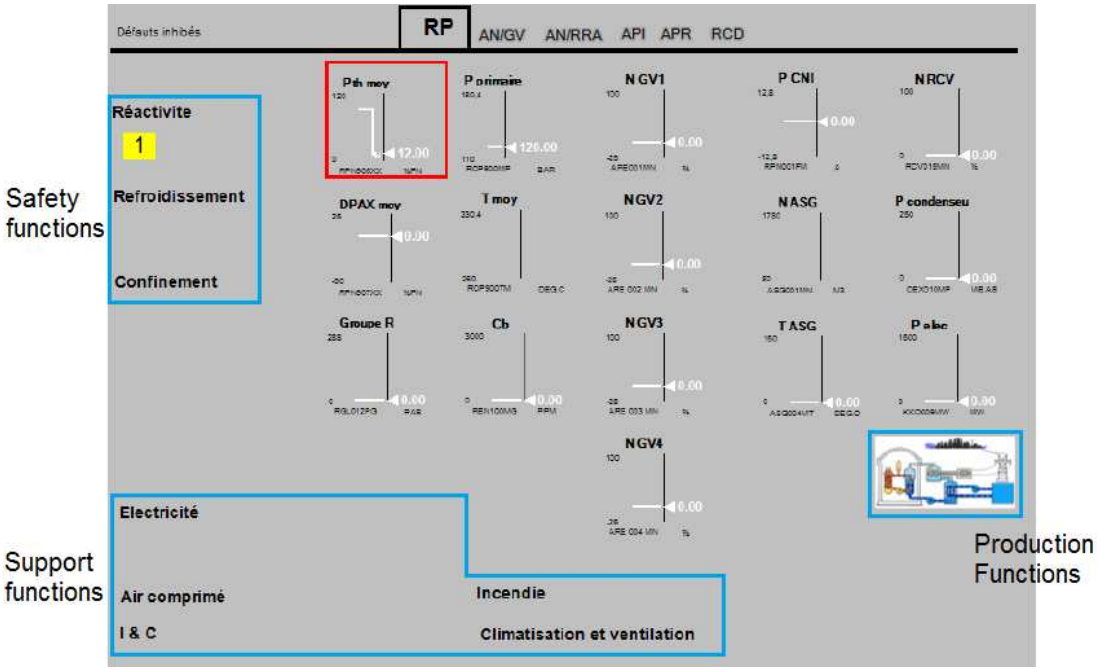


Figure 50: A monitoring system of nuclear-plant control rooms

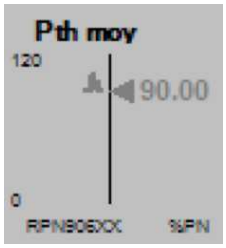
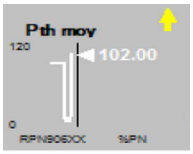
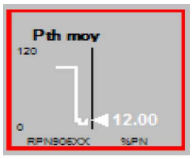
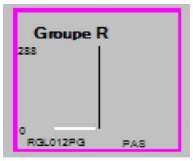
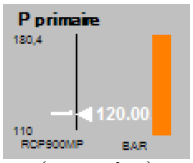


Figure 51: One reactor parameter zoomed out

The top of the parameter widget displays the parameter name (for instance, `Pth_moy`, standing for *average thermal power*). The middle displays the current value of this parameter (90.00) and a line displaying the past values of the parameter. These values vary between a minimum and a maximum value (*thresholds*, also displayed in the widget, e.g., 0 – 120). Finally, the bottom of the widget displays the name of the sensor that monitors the parameter (`RPN906XX`) and its measurement unit (`%PN`).

The system monitors the evolution of all reactor parameters over time. If something unusual occurs in a parameter displayed in the current UI, the parameter is highlighted in different ways. Table 3 lists some kinds of parameter anomalies we considered in this case study, and how such anomalies are highlighted on the UIs.

Table 3: Reactor parameters anomalies and their UI representation

#	Anomaly type	Description	UI representation
1	*threshold overflow* (“dépassement haut”)	when a parameter achieves a value higher than its maximum threshold	 (yellow top arrow)
2	*threshold underflow* (“dépassement bas”)	when a parameter achieves a value lower than its minimum threshold	(yellow bottom arrow)
3	*gradient excess* (“dépassement gradient”)	a calculation based on the variation of the parameter value in one second, with respect to a maximum threshold of variation	 (red square)
4	*invalid measurement* (“invalidité de mesure”)	when a parameter used in the calculation is invalid	 (magenta square)
5	*loss of redundancy* (“perte de redondance”)	if the value measured by a particular sensor diverges by more than 5% from the average of all sensors, this parameter is excluded from the computation of the average	 (orange bar)

3.3.3 Failure Signals

Besides anomalies in the reactor parameters, the system displays failure signals on the left and bottom zones of the UIs, in the reactor *function* that is affected by the parameter anomaly (e.g., the yellow box under the *reactivity* – “réactivité” – function in Figure 50). The system monitors three kinds of reactor functions (Figure 50):

- *Safety* functions: composed of the *reactivity* – “réactivité”, *core cooling* – “refroidissement”, and *confinement* sub functions;
- *Production* functions: composed of the “R”, “V”, “G”, “C”, and “A” sub functions;
- *Support* functions: composed of the *electric* – “électricité”, *pneumatic* – “air comprimé”, *I&C*, *fire protection* – “incendie”, and *fan & air conditioning* – “climatisation et ventilation”

sub functions.

When a parameter has an anomaly, a signal is displayed in these functions. They synthesize failure signals according to the parameters, and even when an anomalous parameter is not currently displayed on the UI, the failure signals is displayed on the corresponding function anyway, to warn the user that anomalies exist on parameters that are not currently visible. Such signals are displayed on the UI with different icons according to the kind of signal. Figure 52 illustrates the kinds of signals that can be generated:

1. *nonconformity* ("non-conformité") (NC in red): indicates a nonconformity on a reactor function, with respect to the system requirements, which are expressed in the form of logic equations;

2. *loss of redundancy* ("perte de redondance") (letter `R` in orange): generated when a reactor parameter triggers a *loss of redundancy* anomaly;

3. *variation of a measure* ("variation d'une mesure") (a black arrow): generated when a parameter value used in the calculation significantly varies with respect to its average;

4. *equipment state change* ("changement d'état") (a red square): notifies a change of state of an equipment;

5. *invalid measurement* ("invalidité de mesure") (in a pink semi border): generated when a reactor parameter triggers an *invalid measurement* anomaly;

6. *alerts* (inside a solid yellow square): indicates the number of alerts generated by the system in this reactor function; and

7. *alarm conditions* (inside a solid orange square): indicates the number of alarm conditions generated by the system in this reactor function. An alarm condition threshold is more severe than an alert. These defaults can be associated - or not - with an alarm, depending, for example, on the plant status filter that does not affect these conditions (or defaults).

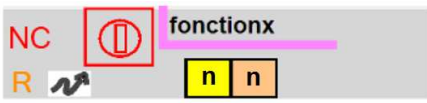
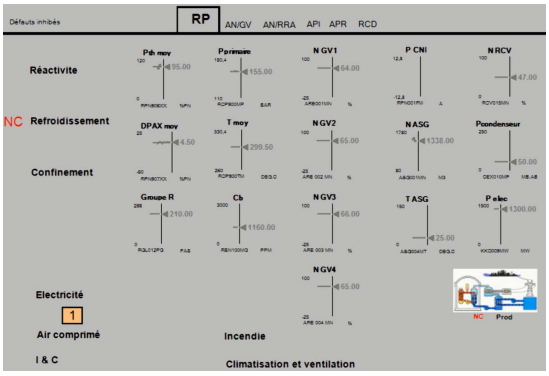
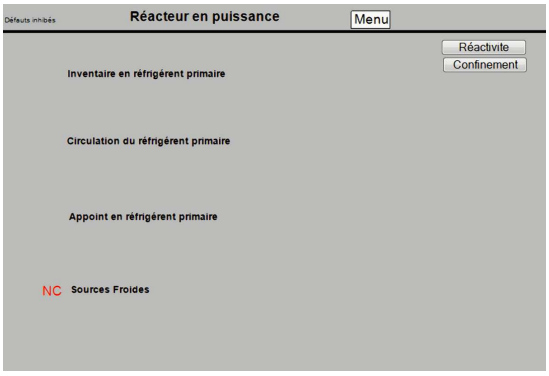


Figure 52: Signals triggered on reactor functions

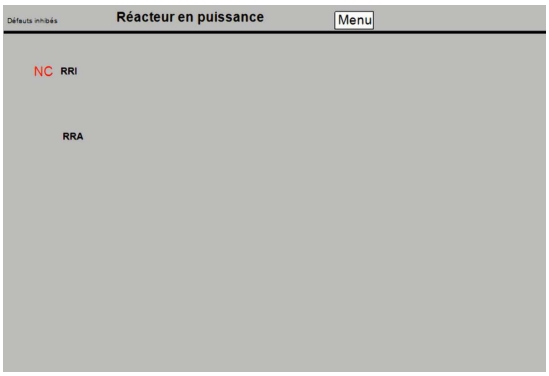
The names of the reactor function under which the failure signals are displayed are clickable, allowing users to access other UIs in order to get more details about the default. Such technique is called the *zoom metaphor*, and it is illustrated in Figure 53: the "Global Synthesis" UI displays a *nonconformity* failure signal in the "Refroidissement" function (Figure 53a); in order to detail the impact of this failure signal, the user can click on the "Refroidissement" function; another UI is displayed with the four "Refroidissement" sub functions, and one can see that the failure comes from the "Sources Froides" sub function (Figure 53b); this sub function can also be clicked, revealing that the failure comes from the "RRI" system (Figure 53c).



(a) A failure is displayed in the "Refroidissement" function ...



(b) ... the failure comes from the "Sources Froides" sub-function ...



(c) ... precisely, from the "RRI" system.

Figure 53: Example of the *zoom metaphor*: from one UI, the user can access other UIs that give more details about what is displayed on the previous UI

3.4 The LIG Prototype

One of the contributions of our research team to the Connexion Project was the application of recent advances of plasticity to the EDF system [Connexion 2013].

3.4.1 The Control Room Version

By applying several ergonomic criteria from the framework proposed in [Bastien & Scapin 1993], the LIG prototype improves the EDF system in several directions. Figure 54 (in French) compares both of them. The UI of the LIG prototype was structured in four zones (instead of the three zones of the EDF system). This organization clearly separates displaying zones from navigation zones on the UI:

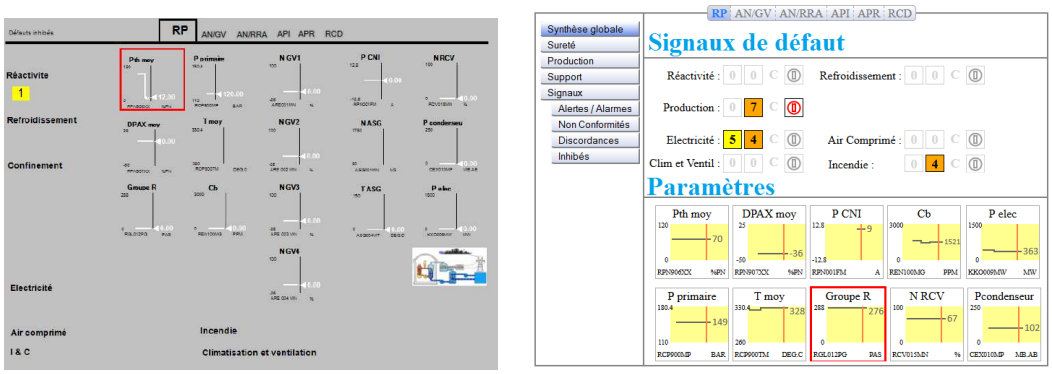
1. The top zone displays six tabs for selecting the plant status, which in the EDF version is also on the top part of the UI, but with a different appearance;

2. Below the plant status zone, the *Failure Signals* ("Signaux de défaut") zone synthesizes the

signals triggered in reactor functions. In the EDF system, such information is displayed inside the menu on the left zone of the UI;

3. At the bottom, the *Parameters* – "Paramètres" – zone displays various reactor parameters. In the EDF system, such information is displayed in the middle of the UI;

4. On the left, a menu permits other UIs to be accessed. In the EDF system, the menu is on the left and at the bottom of the UI.



(a) The EDF version

(b) The LIG version

Figure 54: Two versions of a control room system

Besides, other improvements on the LIG version of the EDF system are:

1. A better spatial organization of the user interface. For instance, in the EDF system, the reactor functions are dispersed at the left and bottom zones of the UI;

2. Some elements on the UI that are clickable are made more prominent (e.g., the six tabs on top of the UI);

3. At the parameter zone of the UI, the reactor parameters are grouped according to the production system to which they relate. For instance, the `Pth moy`, `DPAX moy`, and `PCNI` parameters are positioned one beside the other on the LIG prototype, since they are related to the production system called `RPN`. In the EDF system, these three parameters are dispersed on the UI;

4. In terms of navigation, on the EDF system some UIs do not provide direct access neither to the main UI nor to UIs that were previously accessed. Such navigation capabilities are improved with the hierarchical menu of the LIG prototype;

5. The EDF system is extended to two different modes: *training* and *expert* modes. In the training mode, supplementary guidance is added on the UI, to aid users who are not accustomed to operate the system, such as a *breadcrumb trail*[4]. In the expert mode, these guidance elements are removed to provide lighter UIs to expert users;

[4]The graphical control element *breadcrumb trail* is a navigation aid used in user interfaces to allow users to keep track of their locations within programs or documents (source: Wikipedia).

6. Minor improvements such as alignment of labels, coherence of the police size, etc.

This prototype is implemented using the following technologies: HTML 5, CSS 3, Javascript, PHP (for the server), Server Sent Events for the communication client-server.

3.4.2 Plastic Versions

The control room system has the need to adapt to different contexts of use. For instance, to make users mobile, a tablet version of the UIs could be provided. An example of adaptation is illustrated in Figure 55a, where the control room UI is adapted according to the target platform (a Smartphone). This UI makes users mobile, which is useful when an unexpected event occurs in the plant. While on the PC version (Figure 54b) all reactor signals and parameters are always displayed, on the Smartphone the display is limited to those currently affected by a failure. Besides, the widget representing reactor parameters is re-molded to fit on the size-reduced screen of a Smartphone. Furthermore, while on the PC the menu is always visible (on the left zone of the UI), on the Smartphone it is accessible by a circled button on the top-left corner.

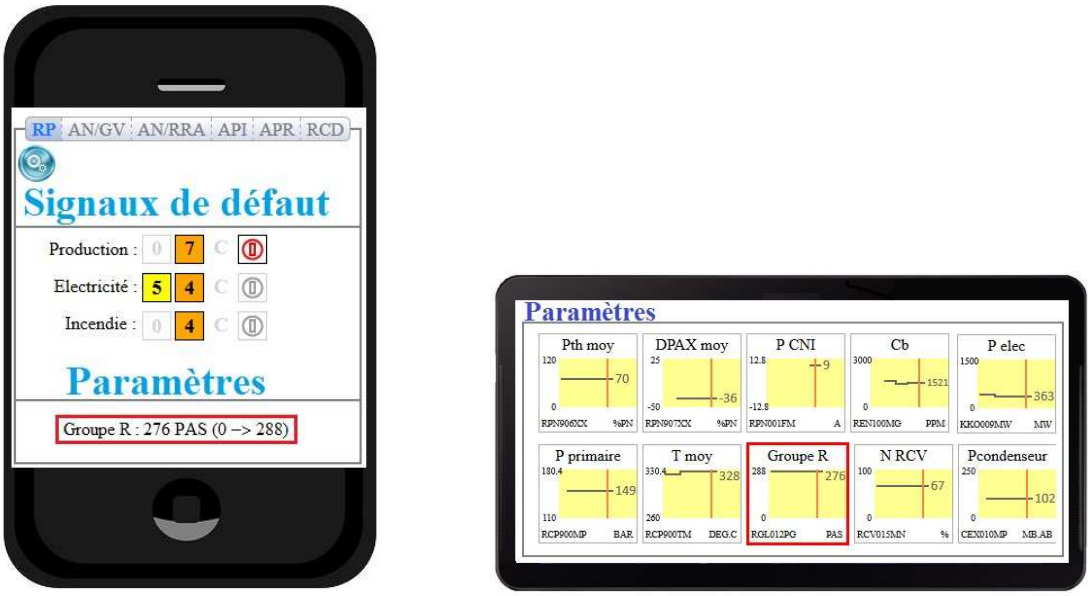
The adaptation applied to the Smartphone UI is placed in the plasticity problem space axes (Figure 4) as follows: re-molding is the *adaptation means*, since the UI visual components are re-molded to adjust the size of the screen; the *UI component granularity* of the adaptation is defined at the interactor level, since the UI widgets are re-molded; the *state recovery granularity* is at the session level: once the UI adaptation occurs, users have to restart their activity from the initial state; the *UI deployment* is static: the UI adaptation is pre-defined at design-time; only changes in the platform are taken into account in the *context of use* of the UI; the *technological space (TS) coverage* is intra-TS, since both UIs are implemented within a single technological space (web technologies); and finally, this example does not contain *meta-UIs*.

Figure 56 illustrates another example of adaptation, in which the UI is adapted according to the target user. This adaptation considers two outermost cases in the training process of users (i.e., control-room operators): training mode (Figure 56a), for users learning how to use the system, and expert mode (Figure 56b). Figure 54b represents an intermediate mode. Figure 56a illustrates the *training* mode. The following elements are added: (1) at the top, a breadcrumb trail helps navigation; (2) some UI zones are entitled (*Failure Signals* – "Signaux de défaut" and *Parameters* – "Paramètres"); (3) non-failure signal symbols have a disabled appearance (e.g., the four symbols beside the *Pneumatic* ("Air Comprimé") function); and (4) reactor functions are line-grouped according to their systems: Safety ("Sureté"), Production, or Support. In expert mode, all this guidance is removed (i.e., the red crosses in Figure 56b), providing lighter UIs for expert users.

Regarding the plasticity problem space, the adaptations applied to the Training and the Expert mode UIs are the same as to the Smartphone UI, except for the *context of use* axis, which considers only changes in the users.

A tablet version (Figure 55b) of the UI illustrates another adaptation means: *redistribution*. The UI is re-distributed on a tablet, but only part of the UI is migrated (i.e., the *Parameters* zone), the other part is displayed on other devices, such as kiosks. The tablet version of the UI also makes users mobile when an unexpected event occurs in the plant.

The adaptation applied to the Tablet UI is placed in the plasticity problem space as follows: redistribution is the *adaptation means*; the *UI component granularity* of the adaptation is defined at the dialog-space level, since the UI zones are the smallest units for redistribution; the *state*



(a) Smartphone UI

(b) Tablet UI

Figure 55: UI platform adaptation

(a) UI in *training* mode

(b) UI in *expert* mode

Figure 56: UI after user adaptation

recovery granularity is at the session level; the *UI deployment* is static; only changes in the platform are taken into account in the *context of use* of the UI; the *technological space (TS) coverage* is intra-TS; and finally, this example contains a *meta-UI with negotiation*: using a meta-UI, the user can decide where to redistribute the UI zones.

The implementation of these prototypes is out of scope of this thesis. Nonetheless, we use

them in some applications of our proposed verification approach.

3.5 The ADACS-N^{TM} Prototype

In the context of the Connexion Project, Atos Worldgrid[5] implemented the same functionalities as the EDF system in their own product called ADACS-N^{TM} (***A**dvanced **D**ata **A**cquisition and **C**ontrol **S**ystem for **N**uclear power*), taking into account the conclusions and improvements proposed by LIG. ADACS-N^{TM} is a commercial product that is deployed in actual nuclear plants. It is a real-time system designed to completely monitor and control a nuclear power plant. ADACS-N^{TM} aims at assisting users (i.e., operators) in their daily tasks in a control room, i.e., in the analysis of a large amount of information, in taking proper actions, and in sending them to the process [Worldgrid 2011a].

ADACS-N^{TM} is a system positioned at level 2 of the overall architecture of a nuclear unit plant (Figure 47), as a computerized process with high availability, integrating all necessary functions for a nuclear control room. ADACS-N^{TM} handles a large amount of data generated by the previous levels at the unit architecture, organizes and displays such data efficiently in graphical user interfaces of several forms, such as synoptic representations of the installation (Figure 57), surveillance mimic displays (Figure 58a), technical datasheets, logbooks, etc.

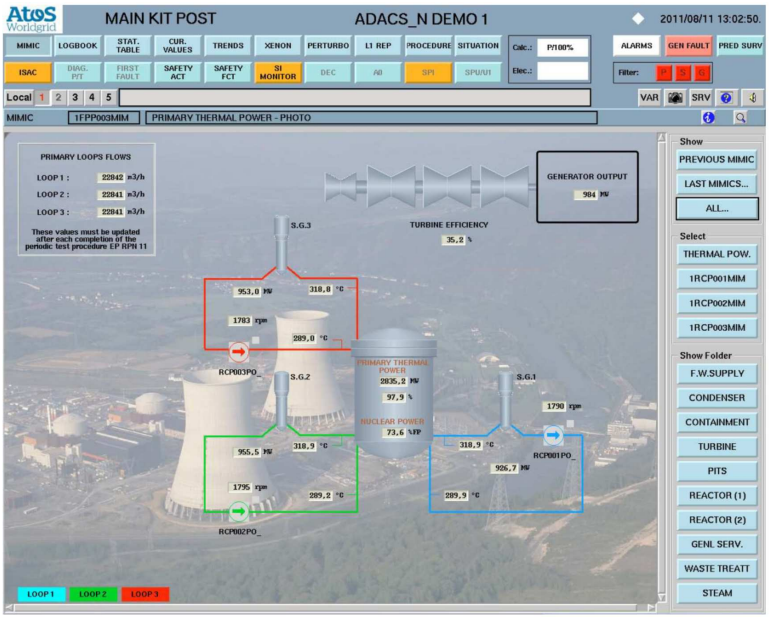
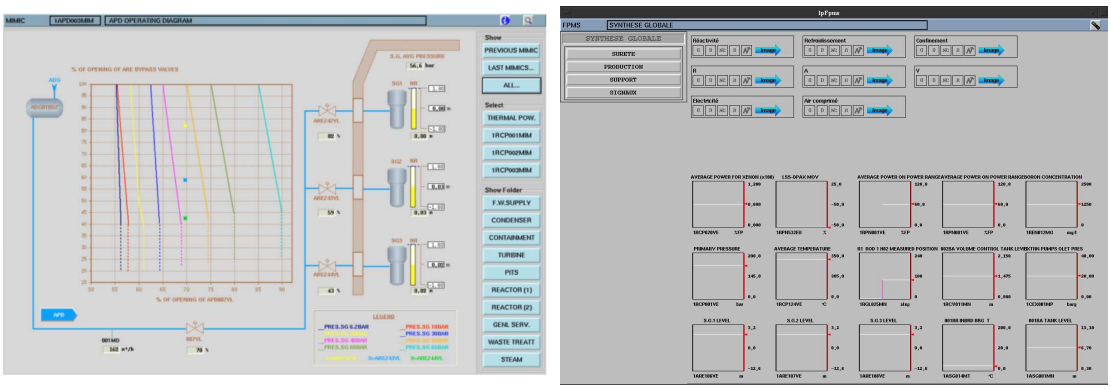


Figure 57: ADACS-N^{TM}: synoptic representations of the installation

Particularly relevant to our study are the user interfaces that implement the EDF system, such as the UI illustrated in Figure 58b. 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 disturbances in the nuclear unit.

[5] `http://fr.atos.net`



(a) Mimic display

(b) Surveillance display

Figure 58: ADACS-N^{TM}: examples of user interfaces

ADACS-N^{TM} structures data by means of *objects* (Figure 59). An ADACS-N^{TM} object is 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, and timestamps) or an output computed by another object. The input data undergo several calculations in the processing unit, generating the object outputs. In our case study, these objects implement the evolution of reactor parameters over time.

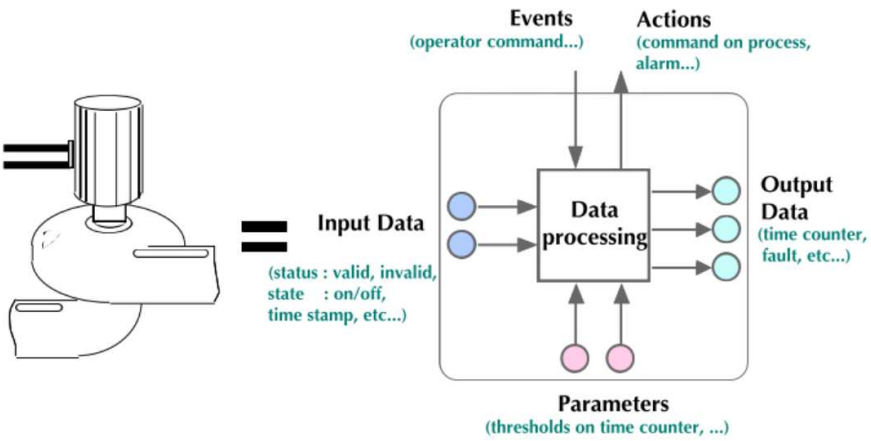


Figure 59: ADACS-N^{TM} objects [Worldgrid 2011b]

ADACS-N^{TM} also has a simulation mode, which is used either for training sessions to nuclear-plant users (i.e., control-room operators), or for data engineering and software testing. The simulation environment is a replica of the real control room system. In this mode, ADACS-N^{TM} can be connected to a set of models that simulate nuclear unit processes. Alternatively, the objects can be manually changed by the training instructors, irrespectively of the laws of physics. Only ADACS-N^{TM} objects that have acquired data as input can be stimulated. Objects that are

dependent of other objects (i.e., their inputs are connected to other object's output) cannot be stimulated. If the set of models is not available, ADACS-N^{TM} can be connected to a stimulator that generates input data.

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. There exists a logging mechanism that records in log files the acquisition of parameters (either real or simulated acquisition), the display of this information on the user interfaces, and the user interactions (Figure 60). Particularly for this case study, reactor parameters are logged in such files.

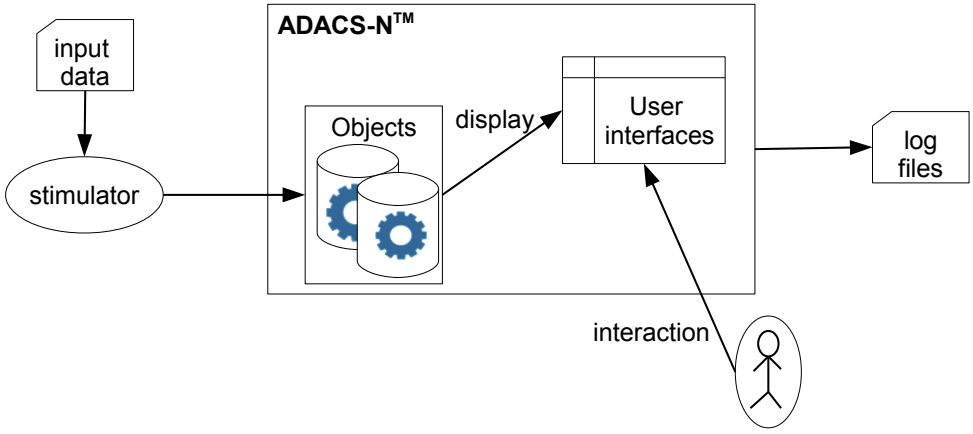


Figure 60: ADACS-N^{TM}: simulation mode and data logging

3.6 Summary

This chapter describes nuclear power plant control rooms, their overall architecture, and the main actors who operate in control rooms. In this context, one specific system has been chosen as one of the case studies of this thesis.

The case study is a system designed to support monitoring activities in the control room, and it was proposed in the context of the Connexion Project by the EDF nuclear electric power generation company. The main UI of this system is described, together with the main zones and widgets displayed in this UI, which information is displayed, and, briefly, how the information is produced. A study to improve ergonomic criteria in the EDF system was conducted in our research laboratory, producing an improved version of this system (the LIG prototype). The EDF and the LIG implementations of the case study are used in the following chapters to illustrate the contributions in this thesis.

Finally, Atos Worldgrid, another partner in the Connexion Project, also studied the EDF case study in the context of the project. Atos Worldgrid implemented the functionalities of the EDF system in its own product called ADACS-N^{TM}. In this chapter, we also present this part of ADACS-N^{TM} that implements the EDF system.

Chapter 4

An Approach to Verifying Interactive Systems

Contents

4.1 Goals **83**
4.2 Formal Techniques **84**
4.2.1 Model Checking 84
4.2.2 Equivalence Checking 85
4.3 Global Approach **85**
4.4 Formal Models Based on the ARCH Architecture **87**
4.5 Languages and Tool Support **88**
4.5.1 CADP Toolbox 89
4.5.2 LNT Specification Language 89
4.5.3 LTS 91
4.5.4 SVL Script Language 91
4.5.5 MCL Property Language 92
4.6 Summary **93**

4.1 Goals

The study of the state of the art described in Chapter 2 shows that:

- no approach so far verifies or compares plastic user interfaces; and
- no approach considers the verification of nuclear power plant systems with users, user interfaces and functional core aspects integrated, while considering usability and functional properties.

Our evaluation of the related work lead us to conclude there is a need of a verification approach of interactive systems covering all the following items:

1. For the verification to be as wide-ranging as possible, the modeling should cover aspects of: users, user interfaces, and functional core;
2. The approach should cover plastic user interfaces, which have specificities that should be considered, like it is shown in the problem space [Vanderdonckt *et al.* 2008, Calvary *et al.* 2011];
3. The kinds of properties the approach verify should include usability and functional properties. When it comes to safety-critical systems, ergonomics and safety requirements should be addressed;

4. The approach should be applicable to safety-critical systems. In particular, the specificities of the nuclear-plant domain described in Section 1.5, page 7, should be addressed, such as the multidisciplinary users that access to system functionalities;

5. For the approach to be applicable to industrial systems, it should scale well to real-life applications. With this goal, as well as to benefit from recent advances in formal methods, we believe that tool support is a necessity.

This chapter gives an outline of our approach to addressing these aspects. The approach allows interactive systems provided with plastic user interfaces to be verified, and can be applied to industrial systems. The chapter starts by describing how we apply formal verification techniques to interactive systems, followed by a global overview of our approach integrating two techniques. The approach is composed of two parts: verification of properties and comparison of models. The aspects common to both parts are introduced in this chapter, such as the modeling according to the ARCH architectural model and tool support. The toolbox and languages which we use to implement the approach are described, as well as how these tools and languages are integrated in the global verification approach.

4.2 Formal Techniques

The formal verification techniques we use in our approach were chosen according to our goal: to verify plastic industrial interactive systems in the context of safety-critical systems. Two techniques integrate our global approach: *model checking* and *equivalence checking*.

4.2.1 Model Checking

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 [Clarke *et al.* 1983, Queille & Sifakis 1982]. In the context of safety-critical systems, model checking can be used to verify safety requirements expressed as properties.

Figure 61 illustrates the model checking technique applied to the verification of interactive systems. Here, the starting point is the interactive system, from which a formal model describing its behavior is manually created. This formal model is then used to verify expected properties. These properties are also manually extracted from the original system, and they express the expected behavior of the system. In our approach, we cover both *usability* and *functional* properties. Usability properties relate to the ergonomic aspects of the system, and can follow existing frameworks such as [Abowd *et al.* 1992, Bastien & Scapin 1993, Vanderdonckt 1994]. In the context of safety-critical systems, it is important to show that the system complies with ergonomic criteria, even if it is not possible to verify all of them. Functional properties relate to the system requirements, and can be extracted directly from the real system, from the requirement document, and from the safety requirements. Properties should be also formalized, which in is this case is done by means of temporal logics.

To complete the process, a satisfiability verification of the properties over the formal model is automatically performed [Clarke *et al.* 1983, Queille & Sifakis 1982]. The analysis feedback is mainly supported by the generation of *counter-examples* when a property is not satisfied. This diagnosis is one of the main benefits of using formal methods to verify interactive systems. It

furnishes a precise way to identify potential problems in the modeled system. The results of the analysis permits the modeled system to be refined.

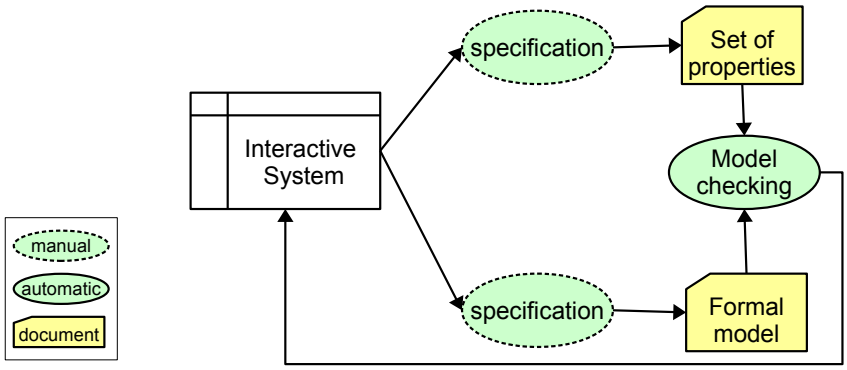


Figure 61: Model checking applied to interactive systems

Model checking has been used in the past years to verify interactive systems in safety-critical systems of several domains, such as avionics [Degani & Heymann 2002], radiation therapy [Turner 1993], healthcare [Thimbleby 2010], etc. In this thesis, we also apply model checking in the context of safety-critical systems, in the nuclear power plant domain.

4.2.2 Equivalence Checking

According to [Combéfis 2013], *equivalence checking* is a suitable technique for model comparison. We follow this idea and propose to use this formal technique in the context of plastic interactive systems. Rather than verifying the satisfiability of properties, equivalence checking permits to show whether two versions of a system exhibit exactly the same behavior or not. In the context of plastic interactive systems, different versions of a system can be generated for different contexts of use (e.g., when executed on a PC or on a Tablet). This technique can be used to compare these different versions of the system.

With this goal, a model of each version of the system is first created, expressing the specificities of each context of use. Then, the models are compared two by two, in the light of a given equivalence relation. Figure 62 illustrates the equivalence checking technique applied to the verification of plastic interactive systems. The numerous equivalence relations available in the literature can be used to show equivalence between two versions of the system at different levels of abstraction. The choice of the equivalence relation depends on the verification goals. The results of the analysis also permit the modeled systems to be refined.

Model checking and equivalence checking are the techniques used in our approach to verify plastic industrial interactive systems.

4.3 Global Approach

We propose a global approach [Oliveira *et al.* 2015c] to assess the quality of safety-critical interactive systems with plastic UIs (Figure 63). We integrate both model checking and equivalence checking: equivalence checking can be used to compare models of the system in

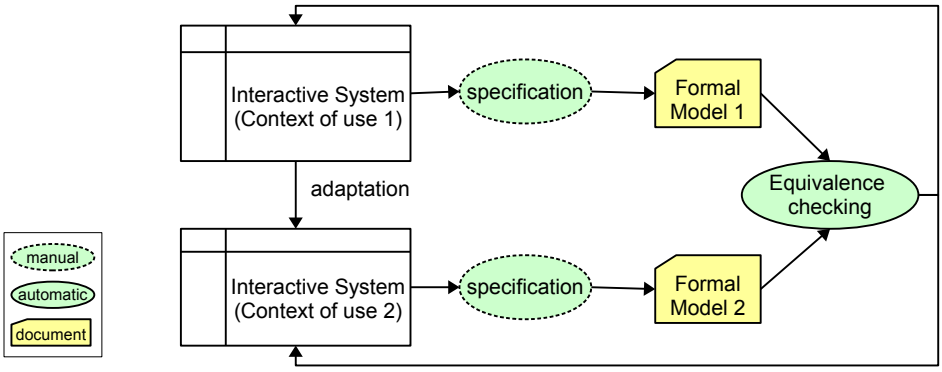


Figure 62: Equivalence checking applied to interactive systems

different contexts of use (i.e., the central part of Figure 63), and for each context of use, a set of properties can be verified over the model of the system by model checking (i.e., the top and bottom parts of Figure 63).

Both parts of the approach can be used either independently or in an integrated way. Independently, disregarding the UI adaptation capabilities of the interactive system, verification of properties by model checking can be performed at any time over the system formal specifications. To take into account the adaptation part of the system, comparison of the different versions of the interactive system can be performed, one for each context of use.

Optionally, comparison of formal models can be integrated with verification of properties: before checking for equivalence between the formal models, checking a set of properties over the formal models can guarantee that they cope with a certain level of quality, increasing the relevance of the equivalence checking results. In case both formal models are expected to satisfy the same set of properties, the verification can be reduced to check the formal model of one version of the interactive system, and to perform equivalence checking in both formal models. If one formal model satisfies the set of properties and this formal model is equivalent to another one, then the second formal model also satisfies the set of properties. It can also be the case that each formal model is expected to satisfy different properties, due to particularities of their context of use. In this case, the approach is fully performed: model checking each formal model with respect to their set of properties, followed by an equivalence verification of the formal models.

Our approach can be used to assess the quality of industrial interactive systems either by verifying a set of properties using model checking, or by comparing different versions of the system using equivalence checking, when the system implements plasticity. In such cases, industrial systems can give inputs to the formal tool so that formal verification can be performed. In this thesis, we will describe an application of the model-checking part of the approach in the validation of an industrial interactive system in the nuclear-plant domain.

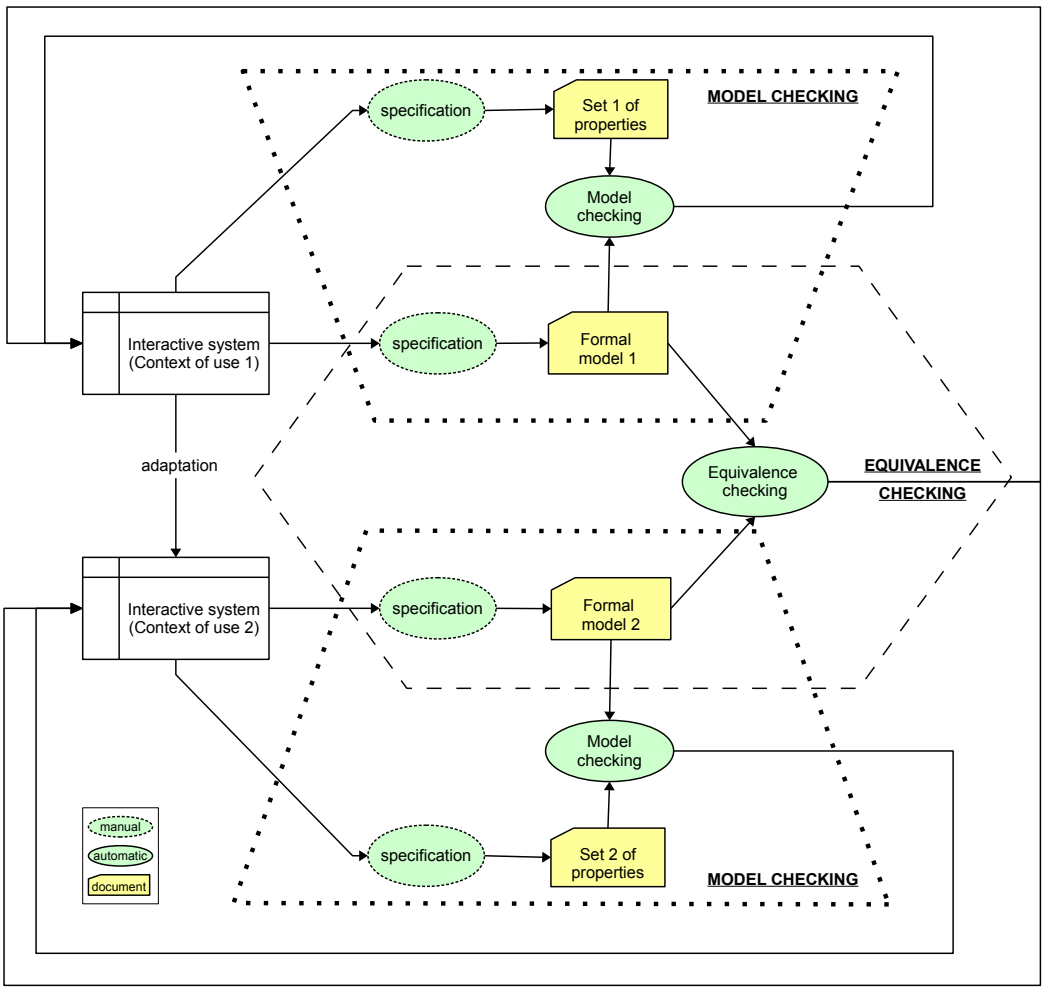


Figure 63: Global approach to verifying interactive systems

4.4 Formal Models Based on the ARCH Architecture

In our approach, the formal models are written manually. Other approaches propose an automatic generation of the formal model, for instance, the approach described in [Paternó 1997], that generates a LOTOS specification directly from a task model. In our case, task models per se do not contain sufficient information to permit the formal model to be automatically generated, since our formal model covers aspects of the user interfaces, the functional core and the user behavior. We write it manually, to be as realistic as possible. Since hand-written modeling can add subjectivity to the formal model, to increase the confidence of the formal model, they are validated with experts in the studied domain. In addition, in order to obtain more reusable results, in our approach interactive system models are represented according to the principles of the ARCH architecture [Bass *et al.* 1991].

Architectural models provide a means to structure systems, using the principle of separation of concerns. In ARCH, systems are decomposed in five main components: the functional core, the functional core adaptor, the logical presentation, the physical presentation, and the dialog controller (i.e., the blue boxes in Figure 64). In our approach, we group some ARCH components into one single component: the functional core and the functional core adaptor components into a *functional core* component, and the logical presentation and the physical presentation into a *UI* component (i.e., the dashed boxes in Figure 64). And finally, the dialog controller ARCH component is normally integrated in our approach. In this way, our approach can cover the functional core and the user interfaces of interactive systems. We take into account this separation of concerns when we create the formal models of the interactive system. In addition, in our approach a component describing the expected user behavior is created.

Figure 64 illustrates how the formal model is organized: each dashed box represents one or more modules of the formal model. In particular, the formal model reflects the fact that users interact only with the user interfaces, not having access neither to the functional core of the system, nor to the dialog controller.

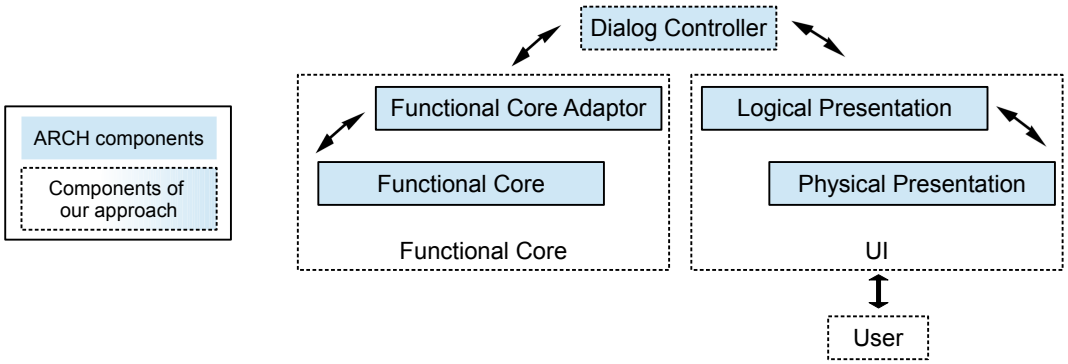


Figure 64: ARCH architecture usage in the formal modeling

Following the classification of verification approaches we propose in Chapter 2, i.e., (i) approaches that model interactive systems as a composition of smaller parts, and (ii) approaches that model them globally, our approach is in the class (i): we model interactive systems as a composition of *modules*.

4.5 Languages and Tool Support

One key enhancement brought by our approach is the application of a more powerful tool support [Oliveira *et al.* 2014]: the CADP[1] toolbox (*Construction and Analysis of Distributed Processes*) [Garavel *et al.* 2013].

[1] `http://cadp.inria.fr`

4.5.1 CADP Toolbox

The choice of the toolbox was mainly motivated by its maturity, continuous evolution, support, and the numerous tools available. CADP 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. Asynchronous systems suit well the modeling of human-computer interactions: the modules that describe the users, the functional core, and the user interfaces can evolve in time at different speeds, which reflects well the unordered sequence of events that take place in human-machine interactions.

CADP has continuously evolved over the past years [Fernandez *et al.* 1996, Garavel *et al.* 2002, Garavel *et al.* 2007, Garavel *et al.* 2013]. It supports the two main formal verification techniques we are interested in: model checking and equivalence checking. Besides, CADP implements: reachability analysis, on-the-fly verification, compositional verification, distributed verification, static analysis, and simulation [Garavel *et al.* 2013].

By taking advantage of the new capabilities added to CADP, it is now possible to perform, for instance, *compositional verification* on individual processes of the model [Garavel *et al.* 2013], enabling to handle much larger state spaces. CADP contains tools to create a graph-representation from the formal model, and the reasoning is performed over this graph. The more complex the system under evaluation is, the larger its graph will be. *Compositional verification* is a way to avoid state-space explosion, by creating an equivalent graph for each component of the model [Garavel *et al.* 2013], replacing a state space by an equivalent but smaller one. In practice, bigger models can be handled, so that one can consider more realistic UI models.

In our approach, we mainly used the EVALUATOR 4.0[2] [Garavel *et al.* 2013] model checker, the BCG_CMP[3] and the BISIMULATOR[4] [Mateescu & Oudot 2008] equivalence checkers, and the OCIS[5] interactive simulator (*Open/Caesar Interactive Simulator*) for step-by-step simulation with backtracking. OCIS allows one to simulate the formal model, and to test it interactively while an execution tree is created and displayed on the UI of the tool (Figure 65). This simulation allows one to explore all the possible executions of the model.

4.5.2 LNT Specification Language

We use the LNT [Champelovier *et al.* 2014] specification language to write formal models of the system. LNT is derived from the E-LOTOS [ISO/IEC 2001] standard. It improves LOTOS [ISO/IEC 1989] and can be translated to LOTOS automatically. LOTOS was originally devised to support standardization of OSI (*Open Systems Interconnection*), but has been used now more widely to model concurrent systems.

LOTOS and LNT are equivalent with respect to expressiveness, but have a different syntax. 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 calculus. In LNT, both parts (data and control) share a common syntax close to the imperative programming style (easier to learn and to read) [Garavel 2015]. In [Paternó 1997] the authors point out

[2] http://cadp.inria.fr/man/evaluator4.html
[3] http://cadp.inria.fr/man/bcg_cmp.html
[4] http://cadp.inria.fr/man/bisimulator.html
[5] `http://cadp.inria.fr/man/ocis.html`

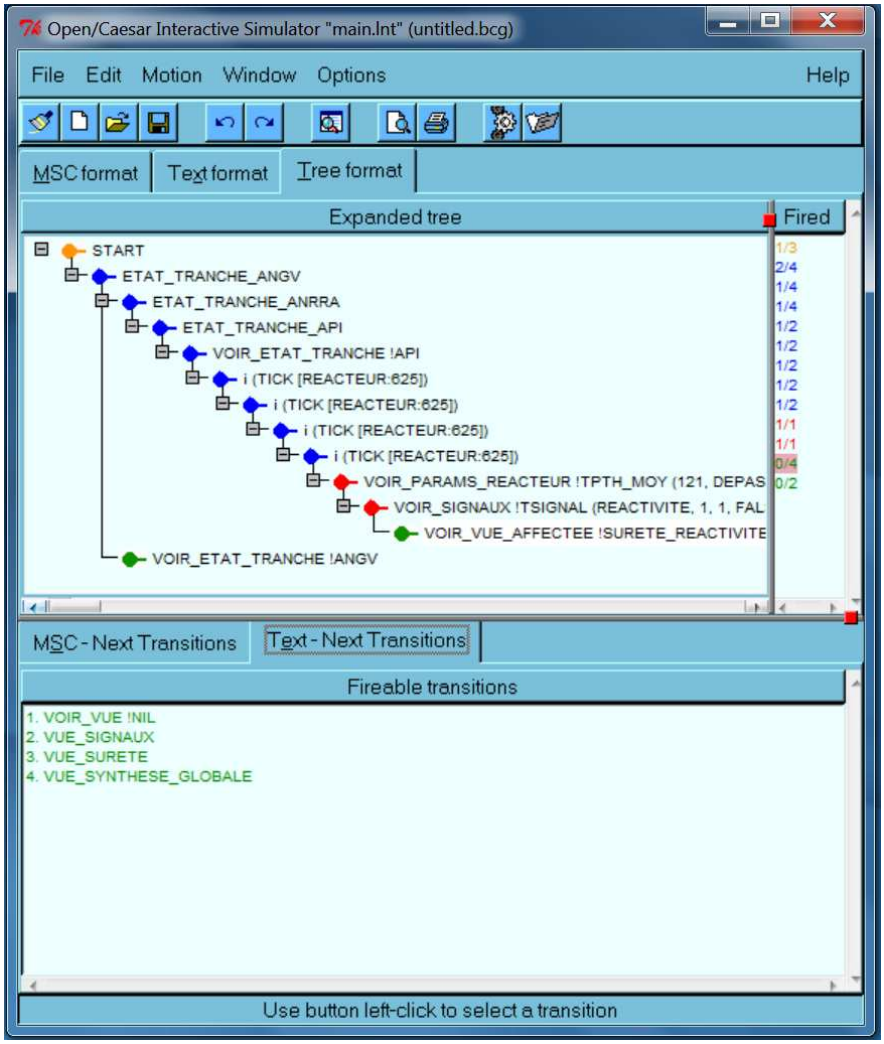


Figure 65: The Ocis (*Open/Caesar Interactive Simulator*) tool

how difficult it is to model a system using Lotos, when quite simple UI behaviors can easily generate complex Lotos expressions. The use of Lnt alleviates this difficulty.

In [Champelovier *et al.* 2014] the authors show the benefits of Lnt over Lotos, notably the user friendliness and the richer data types, to mention only these 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 attenuate the complexity of formal methods and to quicker benefit from their advantages. This mitigates two reasons identified in [Cofer 2012] as some causes to few case studies of formal methods to industrial systems: *usability* (i.e., formal notation and tools are unknown to developers); and *cost* (the creation / maintenance of a formal model is expensive). Besides, the richer data types of Lnt permit more realistic UI models, thus widening the capabilities of verification, covering

verifications on the data type of UI form fields, for instance.

4.5.3 LTS

CADP can generate graph-based models called LTSs (*Labeled Transition Systems*) [Park 1981] (or state-transition diagrams). An LTS (Figure 66) is a graph composed by states and transitions between states. Transitions are triggered by actions, which are attached in the LTS transitions as labels. Intuitively, an LTS represents all possible evolutions of a system formal model. Formally, an LTS is a 4-tuple $\langle Q, A, T, q_0 \rangle$ where:

- Q is a set of states;
- A is a set of actions;
- $T \subseteq Q \times A \times Q$ is a transition relation; and
- $q_0 \in Q$ is the initial state.

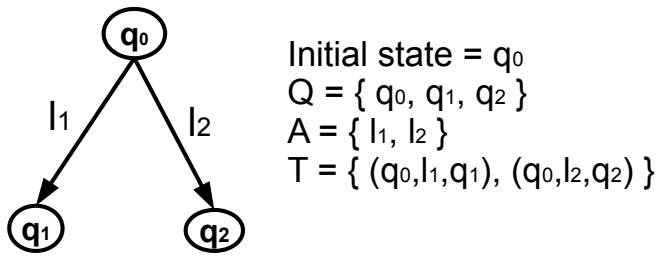


Figure 66: A Labeled Transition System (LTS)

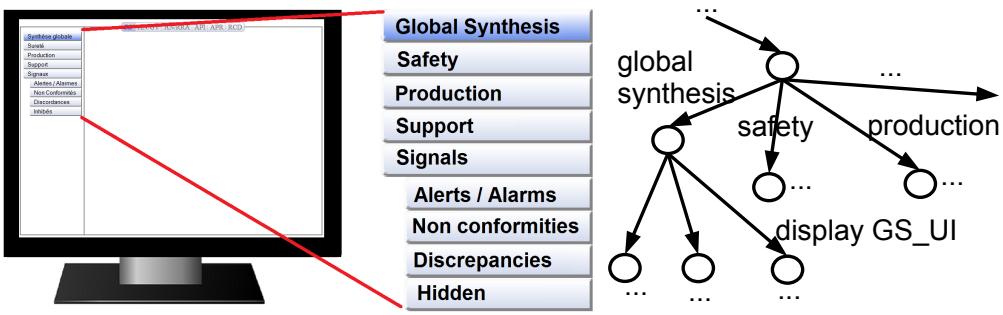
LTSs are suitable to describe systems that change through actions of some kind. In any state of the system, several actions can be performed, leading the system to a new state, the initial state residing before any of these actions have been performed. For any given interactive system, we are generally interested in the set of actions and transitions, since they represent the system dynamics. For example, Figure 67 illustrates how a fragment of a user interface extracted from the case study described in Chapter 3 can be modeled using an LTS. From a given state of the user interface, the three first menu options are available: actions offered by the UI; once executed, these actions change the state of the system. In the LTS, they are modeled using labeled transitions that change the system from one state into another one.

An LTS representation permits the usage of several formal techniques: model checking can be used to verify properties on an LTS, and equivalence checking can compare two LTSs.

4.5.4 SVL Script Language

CADP also provides a scripting language for describing elaborate verification scenarios and to describe verification strategies called SVL[6] (*Script Verification Language*) [Garavel & Lang 2001]. SVL can be seen as a process calculus extended with operations on LTSs, e.g., minimization (also

[6] `http://cadp.inria.fr/man/svl.html`



(a) A menu in a user interface (b) The menu options in an LTS

Figure 67: A UI fragment represented in an LTS

called reduction), abstraction, comparison, deadlock/livelock detection, etc. SVL orchestrate the calls to the CADP tools [Garavel *et al.* 2013]. With this goal, several SVL scripts are implemented mainly in the equivalence checking part of the approach, an in the connection to the industrial system.

4.5.5 MCL Property Language

We use MCL[7] (*Model Checking Language*) [Mateescu & Thivolle 2008] to formalize the expected properties of the interactive systems. 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 contains quantifiers over finite data domains and constructors inspired from functional programming (e.g., `let`, `if-else`, `case`, `while`, `repeat`, etc.) [Mateescu & Thivolle 2008].

The MCL fairness operator allows one to identify the existence of complex unfair (infinite) cycles in the model. An unfair cycle is an infinite sequence made by the concatenation of sub-sequences satisfying the formula, e.g., a sequence of actions over the user interface that once started loops forever. Another advantage of MCL language is the support to data-handling mechanisms on temporal logic formulas. 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:

$$\begin{aligned} &\nu Y(c : nat := 0)\ . \\ &\langle not(req_1 \vee req_2 \vee req_3)^*\ .\ resp\rangle\ true \\ &or \\ &((c < 3)\ and\ [req_1 \vee req_2 \vee req_3]\ Y(c+1)) \end{aligned} \tag{4.1}$$

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)".*

[7] `http://cadp.inria.fr/man/evaluator4.html`, section "Overview of the MCL Language"

The support to data-handling mechanisms is illustrated in this formula by the declaration and initialization of the variable `c`. 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.

Because it is a temporal logic, MCL permits to express *modalities*. Modality sentences assert something not just about what *is* the case, but also about what *could* (or *could not*) be the case. For instance, the *necessity* modality expresses something that *must* be the case, and *possibility* modality expresses something that *could/can/might/may* be the case.

The *necessity* modality is written in MCL using the following pattern: $[\ R\ ]\ F$. It is satisfied by a state of the LTS iff for each transition sequence going out of this state, if this sequence satisfies the formula R, then it must lead to a state satisfying the formula F [Mateescu & Thivolle 2008].

The *possibility* modality is written in MCL using the following pattern: $< R > F$. It is satisfied by a state of the LTS iff there is some transition sequence going out of this state that satisfies the formula R and leads to a state satisfying the formula F.

Figure 68 illustrates how these tools and languages are used to instantiate our global approach.

4.6 Summary

This chapter introduces our global approach to verifying plastic industrial interactive systems using formal methods. The approach tackles two main needs: the usage of formal methods by industrial systems and the verification of plastic user interfaces. We propose an approach based of verification of properties in which model checking is used, and a comparison approach in which equivalence checking is used.

Both parts of the approach can be used either independently or in an integrated way. Independently, the verification of properties by model checking over the system formal specification guarantees a certain level of quality of the system. In order to independently verify the adaptation part of an interactive systems, we propose to compare formal models by means of equivalence checking.

Optionally, the global approach can be used in an integrated way: by verifying a set of properties over different versions of a system model, and comparing these versions with each other afterwards.

Both parts of the global approach share several common points. A formal model of the interactive system is manually created and it is afterward used as an input to the formal verification techniques. Such formal model is structured according to ARCH architecture, in which the functional core, the user interfaces and the dialog controller are modeled, in addition to the users. The global approach is supported by the CADP toolbox, since it is actively maintained, and contains numerous verification tools. Besides, the LNT language is used to model interactive systems, due to its intuitive syntax and semantics. The modular-based programming style [Champelovier *et al.* 2014] proposed by LNT allows the interactive systems to be describe as a composition of *modules* in the formal specification.

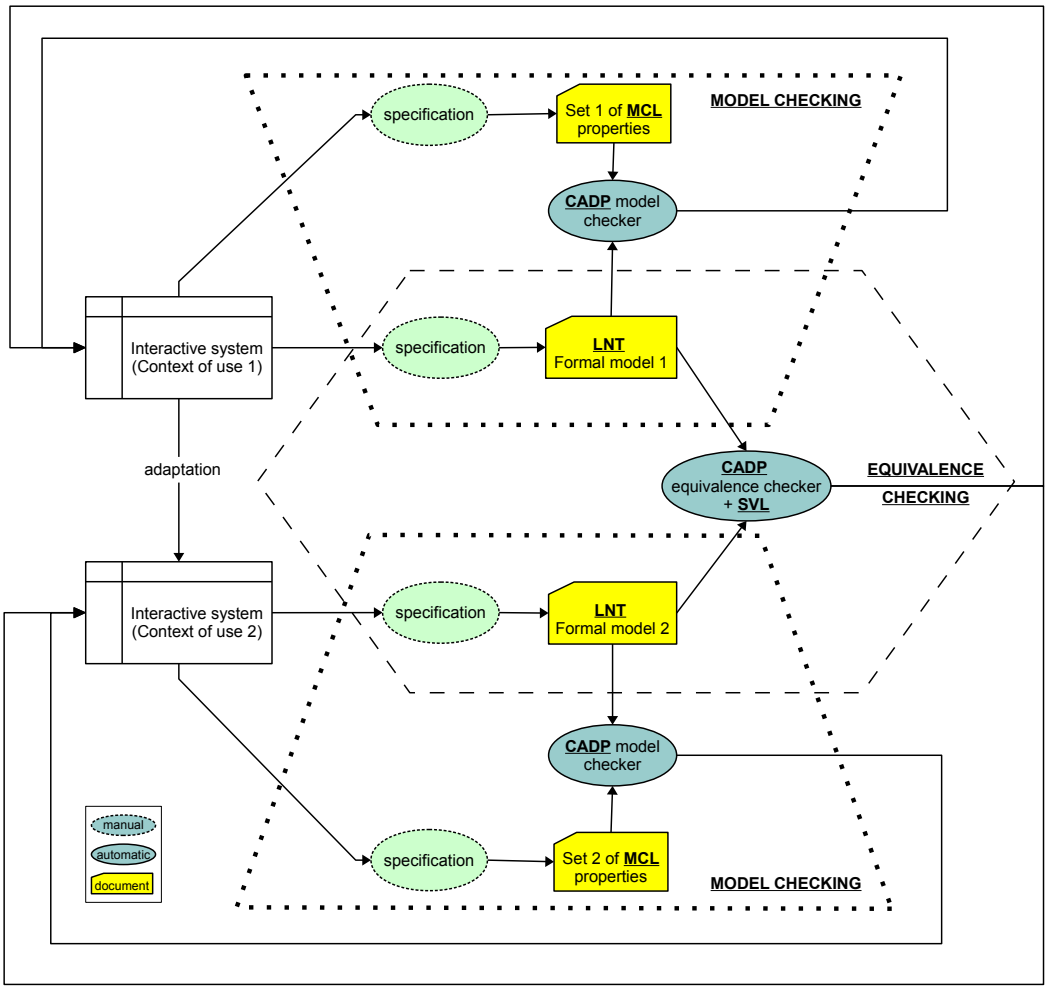


Figure 68: Tools used in the global approach

Chapter 5

Verification of Industrial Interactive Systems

Contents

5.1 Goals **95**
5.2 Formal Model of the Case Study **96**
5.2.1 Overview of the Formal Model 96
5.2.2 User Interface Modules 98
5.2.3 Functional Core Modules 100
5.2.4 Dialog Controller Module 104
5.2.5 User Module 104
5.2.6 Summary of the Formal Model 105
5.2.7 Guidelines to Formally Model Interactive Systems 105
5.2.8 Properties 106
5.2.9 Formal Verification 109
5.3 Propositions to Connect to Industrial Systems **109**
5.3.1 Problem Definition 109
5.3.2 Using the Formal Model to Cross Check the Implementation 110
5.3.3 Proposition 1: Analysis of Traces 111
5.3.4 Proposition 2: Test Case Generation 111
5.3.5 Proposition 3: Co-Simulation 112
5.3.6 Rationale of the Chosen Proposition 112
5.4 On the Connection to an Industrial System **113**
5.4.1 Improvements in the Formal Model 114
5.4.2 Improvements in the ADACS-N^{TM} Platform 116
5.4.3 Connection between ADACS-N^{TM} and the Formal Model 116
5.5 Conclusions of the Connection **128**
5.6 Summary **129**

5.1 Goals

This chapter proposes an approach to verifying interactive systems independently of plasticity. For this, the model checking part of our global approach (cf. Section 4.3 on page 85) is used in an application to the case study described in Chapter 3. Some insights are given into how to formally model interactive systems, extracted from our own experience in the modeling of this case study. We also describe in this chapter several possibilities of the use of the formal model to verify an industrial system. For this, we apply our approach in a common case study with an industrial system called ADACS-N^{TM}.

Besides our laboratory (LIG), an industrial partner in the Connexion Project (Atos Worldgrid) also implemented the EDF system in one of its product, called ADACS-N^{TM}. In order to cross check the ADACS-N^{TM} implementation of the EDF system, an integration of the formal model with ADACS-N^{TM} is described in this chapter. Three different propositions to connect ADACS-N^{TM} to the formal model are detailed and compared, and the chosen proposition is then detailed.

5.2 Formal Model of the Case Study

An important step in formal methods is the modeling of the system under study. Usually, models describe the individual behavior of each component as well as the composition of all components to form a system [Garavel & Graf 2013]. Abstractions are always necessary, since a model is a representation of the reality. However, they should preserve as much as possible the usefulness of the model. This section describes how the case study introduced in Chapter 3 is modeled, the abstractions that are done to focus on the relevant aspects, and the formal verification that the model allows to be performed.

The modeling of this case study paves the way to our verification proposition. Several insights into how to model interactive systems and user interfaces emerge in this process, which are given in Subsection 5.2.7, page 105. In the next subsections the formal model is described, sometimes illustrated with pieces of LNT code. In Appendix B, more details of the LNT specification of this case study is given.

5.2.1 Overview of the Formal Model

Following the separation of concerns proposed by ARCH (Figure 64), the formal model of the control room system contains modules describing the system functional core, the user interfaces (UIs) and the dialog controller. Figure 69 illustrates the modules of the formal model. In order to describe the behavior 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.

Figure 70 illustrates how the modules exchange data with each other. They are coupled as follows: the *user* sets the current *plant status*, which determines the signals that are simulated by the module *generate signals*. This module also receives from the *reactor* module a list of reactor parameters with their current value and status, to simulate failure signals accordingly. A list of reactor parameters and failure 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.

Some bridges that avoid the model to fully follow ARCH are included in the model for optimization reasons. First, the *user* module receives data from the dialog controller, instead of only communicating with the UI modules. As a matter of fact, in this modeling, no treatment is needed in the data coming from the dialog controller. Thus, the controller sends the data

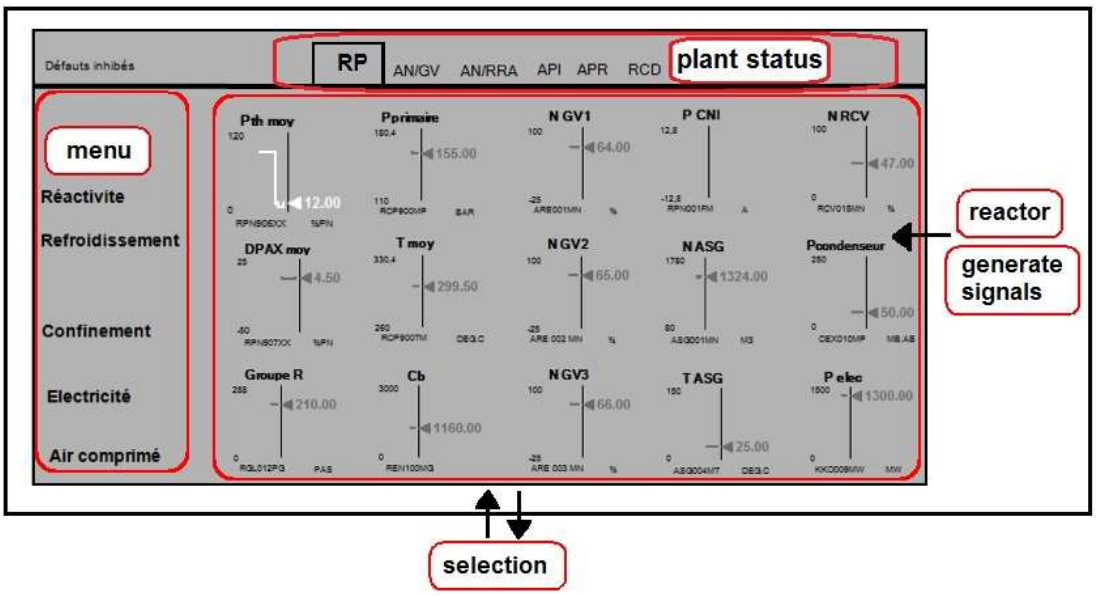


Figure 69: Main modules of the formal model

directly to the user, to avoid the extra flow of these data through the UI before reaching the user. Secondly, for the same reason, the *plant status* UI module communicates directly with the *generate signal* module in the functional core.

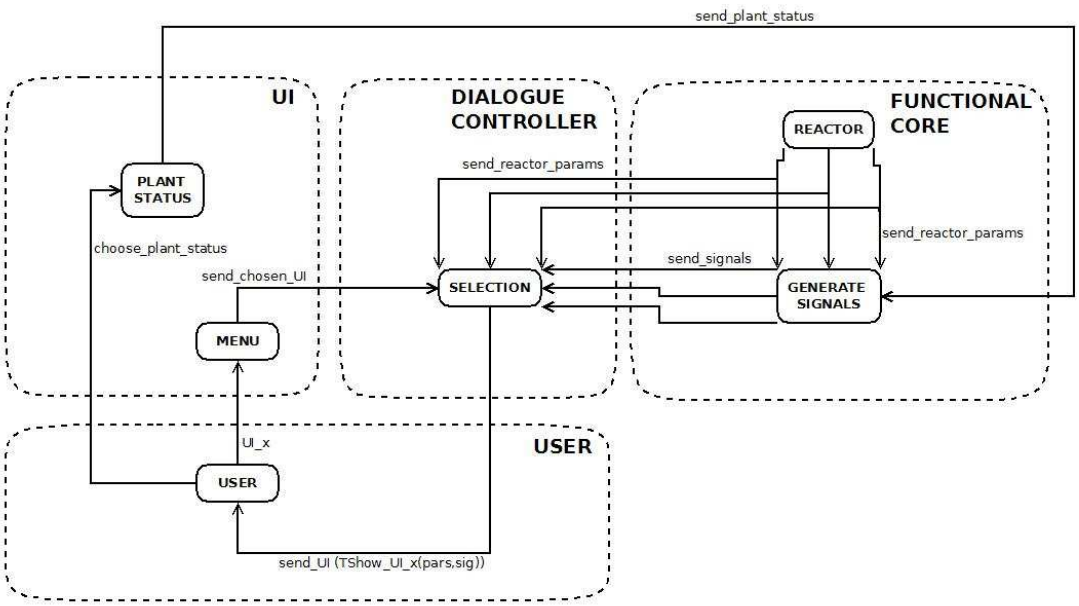


Figure 70: Formal model structure of the EDF system

Each one of these modules are specified using LNT. The LNT language proposes a modular-based programming style [Champelovier *et al.* 2014], which suits well the modeling of interactive systems by composition. Modularity is a key for scalability: it provides a means for organization, abstraction, and re-usability [Sighireanu *et al.* 2004]. The next subsections illustrate how we formally describe each module.

5.2.2 User Interface Modules

In order to model the user interfaces, we identify the UI zones with which users can interact. Each zone is described by one module in the formal model. In the EDF system, two UI zones are identified: the *plant status* zone (at the top of Figure 69) and the *menu* zone (on the left of Figure 69).

a) Plant Status

Plant status module describes the area on the UI where the plant status can be chosen. Six status are available (cf. Section 3.3.1, page 72), and they can be changed either manually (by the user) or automatically (thanks to a system detection). Either way, they change progressively: from RP, the status can transit to AN/GV; from AN/GV it can transit either to AN/RRA, or back to RP, and so forth until the RCD status. Transitions between status that take more than one step further/backwards are not allowed. This kind of constrained accessibility is quite common in user interfaces. We propose to model this by an operator that allows one to implement *choices*. In LNT, this is modeled using the `select` operator (Figure 71, lines 12-32). The command `select G1 [ ] G2 end select` may execute either `G1` or `G2` nondeterministically. To represent that the plant status is constrained to transit in a certain order, the clause `where` can be used: the command `select G1 where A1 [ ] G2 where A2 end select` may execute G1 whenever the condition `A1` holds, or `G2` whenever the condition `A2` holds.

The *plant status* module currently implements six plant status.

b) Menu

The menu on the left of the UIs provides access to all user interfaces (Figure 69). Users interact with the menu mainly to zoom into reactor parameter anomalies, by accessing other UIs that give more details about the anomaly. Figure 72 illustrates the menu behavior. We implement a hierarchical menu: initially, five menu options should be accessible: *global synthesis*, *safety*, *production*, *support*, and *signals*. Except for *global synthesis* option, all menu options have sub-options. A menu option is accessible when the last menu option accessed is:

- its parent; or
- itself; or
- one of its descendants; or
- one of its siblings; or
- one of its siblings' descendants.

```
process plant_status_p [etat_tranche_RP,
                        etat_tranche_ANGV,
                        etat_tranche_ANRRA,
                        etat_tranche_API,
                        etat_tranche_APR,
                        etat_tranche_RCD: None,
                        fixer_etat_tranche: TEtat_tranche,
                        voir_etat_tranche: TEtat_tranche] is
    var etat_tranche_v: TEtat_tranche in
      fixer_etat_tranche (?etat_tranche_v);
      loop
        select
          etat_tranche_RP where (etat_tranche_v == ANGV);
          etat_tranche_v := RP
          []
          etat_tranche_ANGV where (etat_tranche_v == RP) or (etat_tranche_v == ANRRA);
          etat_tranche_v := ANGV
          []
          etat_tranche_ANRRA where (etat_tranche_v == ANGV) or (etat_tranche_v == API);
          etat_tranche_v := ANRRA
          []
          etat_tranche_API where (etat_tranche_v == ANRRA) or (etat_tranche_v == APR);
          etat_tranche_v := API
          []
          etat_tranche_APR where (etat_tranche_v == API) or (etat_tranche_v == RCD);
          etat_tranche_v := APR
          []
          etat_tranche_RCD where (etat_tranche_v == APR);
          etat_tranche_v := RCD
          []
          voir_etat_tranche (etat_tranche_v)
        end select
      end loop
    end var
```

Figure 71: The *plant status* module - an excerpt of LNT code

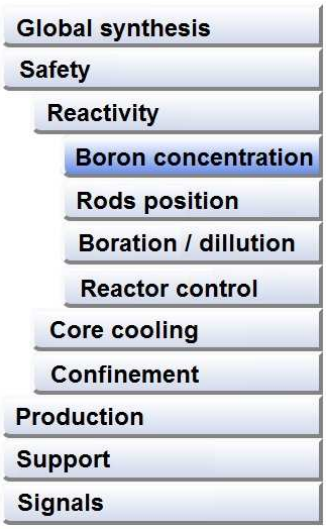


Figure 72: A menu of the control room system

For instance, in Figure 72, the *Reactivity* menu option is accessible when the last menu option accessed was *Safety*, or *Reactivity*, or *Boron concentration* (or the other descendants), or *Core cooling* (or the other siblings), or *Inventory* (a *Core cooling* descendant, or the other

siblings' descendants).

Menu is also a quite common feature of user interfaces. In the formal model, we propose to implement the menu behavior by rules that specify when a given menu option should be accessible, which we implement in the *menu* module by a recursive function (cf. Appendix B).

The *menu* module currently implements 45 menu options.

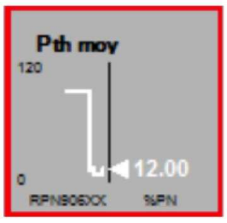
5.2.3 Functional Core Modules

To model the functional core, attention should be paid to the aspects we envisage to verify afterwards. An exhaustive modeling is not advised: depending on the size of the interactive system, the functional core aggregates numerous functions which are not necessarily subject to formal verification. A nuclear reactor system implements complex physics laws. It is not the goal of this work to implement complex nuclear reactor components as they are implemented in the real system. Rather than that, the goal is to reason over such systems, with a special focus on the user interfaces and their adaptation. For this case study, we create two modules to implement functions from the functional core: the *reactor* and *generate signals* modules. The former has functions allowing the reactor parameter values to evolve in time, and the latter generates anomalies in these values, in order to simulate disturbances on the reactor.

a) Reactor

The *reactor* module describes the evolution of the reactor parameter values over time, and it simulates several anomaly scenarios. The module currently includes 29 reactor parameters (Appendix A gives a full list of these parameters). Each reactor parameter has several attributes (cf. Section 3.3.2, page 72). We model the following ones: the parameter name, the parameter current value, a possible anomaly of the parameter, and the minimum / maximum values that each parameter can take. Information such as the past values of the parameter, the sensor that monitors the parameter, and its measurement unit are not included in the model.

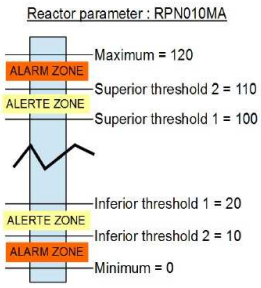
Figure 73 illustrates the structure defined in LNT to model the parameters and their anomalies. The type `TParam` (line 1) contains a list of 29 parameters, each with its current value and anomaly. Five anomalies are modeled (type `TDefault`, line 11), in addition to a special value `nil` when the parameter has no anomaly.




```
1  type TParam is
2    TPth_moy      (valeur_n: nat,  defaut: TDefaut),
3    TDPAXmoy      (valeur_i: int,  defaut: TDefaut),
4    TGroupeR      (valeur_n: nat,  defaut: TDefaut),
5    ...
6    TRPN439EU     (valeur_i: int,  defaut: TDefaut)
7    with "get", "set", "==", "!="
8  end type
9
10
11 type TDefaut is
12   nil,
13   depassementHaut,
14   depassementBas,
15   depassementGradient,
16   perteRedondance,
17   invaliditeMesure
18   with "!=", "=="
19 end type
20
```

Figure 73: Extract of the reactor parameter modeling in LNT

Values of reactor parameters change over different anomaly scenarios. Each anomaly scenario affects a single parameter, and it consists of several *instants*. At each instant, the scenario changes the value of a reactor parameter and may trigger an anomaly on it. During the execution of a scenario over one parameter, the other parameters are assigned to their mean values. For instance, Figure 74 illustrates the simulation of the *threshold overflow* ("dépassement haut") scenario over the `RPN010MA` reactor parameter. The parameter has several attributes such as: a minimum value, two inferior thresholds, a mean value, two superior thresholds and a maximum value. The *threshold overflow* scenario has seven instants: it starts by setting the reactor parameter to its mean value (i.e., an average between its minimum and maximum values). In the next instants, this value is re-calculated according to the formulas indicated in the figure, in function of the parameter's attributes. In the fourth instant, the parameter value exceeds its first superior threshold (i.e., 100), triggering the *threshold overflow* ("dépassement haut") anomaly. In the next instants the parameter value decreases until it reaches its mean value again, making a parabola.



Reactor parameter information : « RPN010MA »			
min	**inf_thres2 (i2)**	**inf_thres1 (i1)**	**mean (m)**
0	10	20	60
sup_thres1 (s1)	**sup_thres2 (s2)**	**max**	
100	110	120	

	Scenario1 = threshold overflow (7 instants)						
instant	**1**	**2**	**3**	**4**	**5**	**6**	**7**
formula	m	s1+m/2	s1-10	s1+1	s1-10	s1+m/2	m
values	60	80	90	101	90	80	60

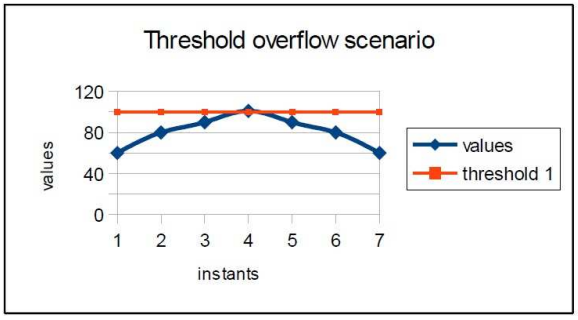


Figure 74: The *threshold overflow* ("dépassement haut") scenario simulated in a reactor parameter

Five anomaly scenarios (cf. Table 3) are simulated in an infinite loop of cycles over all 29 parameters. Table 4 illustrates the number of instants of each anomaly scenario.

Figure 75 illustrates the order in which the five scenarios are simulated. In each cycle (i.e., the columns of the table), all 29 parameters simulate a different anomaly. The first cycle starts by simulating a *threshold overflow* ("dépassement haut") on the `PTHMOY` parameter (line 1), followed by a simulation of a *threshold underflow* ("dépassement bas") on the `DPAX` parameter (line 2), etc. When the first cycle finishes, the second one takes place affecting other anomalies to the same 29 parameters. By varying the parameter and the simulated anomaly at each time, at the end of five cycles all parameters simulate all anomalies, and the infinite loop re-starts.

Table 4: Anomaly scenarios and the number of *instants*

#	**Anomaly type**	# ***instants***
1	threshold overflow	7
2	threshold underflow	7
3	gradient excess	8
4	invalid measurement	6
5	loss of redundancy	8

Reactor Parameter	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	qt. anomalies
1 PTHMOY	1 DH	5 PI	4 RM	3 DG	2 DB	5
2 DPAX	2 DB	1 DH	5 PI	4 RM	3 DG	5
3 GROUPER	3 DG	2 DB	1 DH	5 PI	4 RM	5
4 PPRIMAIRE	4 RM	3 DG	2 DB	1 DH	5 PI	5
5 TMOY	5 PI	4 RM	3 DG	2 DB	1 DH	5
6 CB	1 DH	5 PI	4 RM	3 DG	2 DB	5
7 NGV1	2 DB	1 DH	5 PI	4 RM	3 DG	5
8 NGV2	3 DG	2 DB	1 DH	5 PI	4 RM	5
9 NGV3	4 RM	3 DG	2 DB	1 DH	5 PI	5
10 NGV4	5 PI	4 RM	3 DG	2 DB	1 DH	5
11 PCNI	1 DH	5 PI	4 RM	3 DG	2 DB	5
12 NASG	2 DB	1 DH	5 PI	4 RM	3 DG	5
13 TASG	3 DG	2 DB	1 DH	5 PI	4 RM	5
14 NRCV	4 RM	3 DG	2 DB	1 DH	5 PI	5
15 PCONDENS	5 PI	4 RM	3 DG	2 DB	1 DH	5
16 PELEC	1 DH	5 PI	4 RM	3 DG	2 DB	5
17 R1	2 DB	1 DH	5 PI	4 RM	3 DG	5
18 R2	3 DG	2 DB	1 DH	5 PI	4 RM	5
19 G21	4 RM	3 DG	2 DB	1 DH	5 PI	5
20 G22	5 PI	4 RM	3 DG	2 DB	1 DH	5
21 G1	1 DH	5 PI	4 RM	3 DG	2 DB	5
22 RPN01	2 DB	1 DH	5 PI	4 RM	3 DG	5
23 RPN02	3 DG	2 DB	1 DH	5 PI	4 RM	5
24 RPN03	4 RM	3 DG	2 DB	1 DH	5 PI	5
25 RPN04	5 PI	4 RM	3 DG	2 DB	1 DH	5
26 PARAMX	1 DH	5 PI	4 RM	3 DG	2 DB	5
27 PARAMY	2 DB	1 DH	5 PI	4 RM	3 DG	5
28 PARAMZ	3 DG	2 DB	1 DH	5 PI	4 RM	5
29 PARAMW	4 RM	3 DG	2 DB	1 DH	5 PI	5

Anomalies					
DH	threshold overflow	DG	gradient excess	PI	invalid measurement
DB	threshold underflow	RM	loss of redundancy		

Figure 75: In five cycles of anomalies all reactor parameters are affected by all anomaly scenarios

At the end of the five cycles, 145 scenarios are executed before the loop re-starts (5 cycles $\times$ 29 parameters). At a finer level of granularity, i.e., considering the *instants* of each anomaly scenario (Table 4), since at the end of 5 cycles all the 29 parameters simulate all the 5 anomalies, a total of 812 iterations take place before the loop restarts $((29\times7)+(29\times7)+(29\times8)+(29\times6)+(29\times8))$. This level of granularity in which we model the functional core provides a model large enough for our verification goal: a functional core whose coverage permits a reasonable combination of anomaly scenarios. Even though the model of the functional core does not implement complex physics laws of the nuclear-plant domain, it simulates several scenarios in numerous

reactor parameters over more than 800 iterations, if we consider only the *reactor* formal module. Combined with the other modules of the formal model, in particular the ones that model the display of these anomalies on the UIs and the user interactions, this model provides a reasonable state space for the analysis.

The *reactor* module currently implements five kinds of anomalies on 29 reactor parameters.

b) Generate Signals

The anomalies triggered in the reactor parameters also generate several failure signals in different reactor functions which are displayed on the left of the UIs (cf. Section 3.3.3 on page 74). For instance, the *reactivity* (réactivité) reactor function (on the left in Figure 69) aggregates the following reactor parameters: `Pth moy`, `DPAX moy`, `Cb`, and `NGV1`, meaning that once any of these reactor parameters has any of the five anomalies we modeled (i.e., *threshold overflow*, *threshold underflow*, *gradient excess*, *invalid measurement*, or *loss of redundancy*), a failure signal is generated on the *reactivity* (réactivité) function.

Seven types of signals can be generated. As for the anomalies in the reactor parameters, we are not interested in the physics laws driving these signals in the real system. Rather than that, we focus on analyzing the reactions of the UIs to such signals and the user interactions. Thus, currently the *generate signals* module generates such signals on a reactor function in the following way:

1. *nonconformity* signal: whenever a parameter aggregated in the reactor function has any anomaly;

2. *loss of redundancy* signal: whenever a parameter aggregated in the reactor function has a *loss of redundancy* anomaly;

3. *variation of a measure* signal: whenever a parameter aggregated in the reactor function has a *gradient excess* anomaly;

4. *equipment state change* signal: whenever a parameter aggregated in the reactor function has changed its condition since the last interaction, i.e., it had an anomaly and now it has not, or it had no anomaly and now it has one;

5. *invalid measurement* signal: whenever a parameter aggregated in the reactor function has an *invalid measurement* anomaly;

6. *alerts* signal: whenever a parameter aggregated in the reactor function has any anomaly; and

7. *alarm conditions* signal: whenever a parameter aggregated in the reactor function has any anomaly. An extra alarm is generated if the anomaly is one of the following: *threshold overflow* and *threshold underflow*.

The *generate signals* module currently implements these seven kinds of signals over 37 reactor functions aggregating 29 reactor parameters.

5.2.4 Dialog Controller Module

The *selection* module makes a bridge between the user interface modules and the functional core modules (Figure 70). It receives the parameters and signals from the functional core, filters the ones pertaining to the current UI, and sends them to the user. The user then may select a menu option, to access another UI. In a cycle, the *selection* module receives the menu chosen by the user, displays the corresponding UI to the user, with the parameters and signals belonging to the UI.

The *selection* module currently receives 29 reactor parameters and 37 reactor functions from the functional core, and can send to the user seven different UIs.

5.2.5 User Module

In order to model users, we focus on the actions a user can execute on the user interfaces. We model a "rational" user, i.e., a user that behaves as expected: once a UI displays an anomaly in one reactor parameter, the user interacts with the UI in order to have more details about this anomaly.

In this case study, users can perform two activities: monitoring of reactor parameters and selection of the plant status. In the formal model, the monitoring of reactor parameters is represented by dataflows to/from the *user* module (Figure 70). To the *user* module it is sent the reactor parameters with their current value and possible anomaly, as well as the signals on the reactor functions. In case some discrepancy in such data exists (i.e., anomalies on reactor parameters or failure signals on reactor functions), the user reacts to it by interacting with the menu options of the UI. Thus, from the *user* module to the *menu* module it is sent the menu option chosen by the user.

Another activity a user can perform is the selection of the plan status. This is represented in the formal model by a data flow from the *user* module to the *plant status* module (Figure 70).

Both activities are coded in LNT as follows (Figure 76): the user first selects the plant status (lines 3-10); to monitor the reactor parameters and signals, it receives the dataflow containing the reactor information (line 13) and checks if anomalies exist on the reactor parameter (lines 18-23). Whenever a parameter has an anomaly, the menu option that gives access to a UI detailing such parameter is accessed (line 21).

Our modeling of users does not cover human errors [Fields *et al.* 1995c] neither it is our goal to model human perception [Bolton 2008]. We concentrate on a common monitoring activity of the users (i.e., the control-room operators): the reaction to discrepancies in the reactor [Chériaux *et al.* 2012]. The following hypotheses are made in the modeling: (1) the user makes decisions only on the basis of information provided by the system; (2) the user has a correct understanding of the system functionalities; (3) the user constantly monitors the system and reacts as soon as a reactor parameter or function presents a discrepancy. Such hypotheses provide enough possibilities of user interactions with the UIs to have a reduced, yet realist, formal model. These are abstractions that are done in the modeling of real users.

The *user* module currently implements the selection of six plant status, and the access to seven user interfaces.

```
1  process user_p [(...)]  is
2      (...)
3      select
4           fixer_etat_tranche (RP);
5        [] fixer_etat_tranche (ANGV)
6        [] fixer_etat_tranche (ANRRA)
7        [] fixer_etat_tranche (API)
8        [] fixer_etat_tranche (APR)
9        [] fixer_etat_tranche (RCD)
10     end select;
11
12     loop v in
13          voir_selection(?selection_v);
14
15          va := selection_v.av;
16          case va in
17             afficher_SG(any TParam, ..., any TMsg, any TSignal, ...) ->
18                if (va.Pth_moy_v.defaut != nil)
19                or (va.DPAXmoy_v.defaut != nil)
20                   (...) then
21                   vue_affectee := surete
22                else vue_affectee := synthese_globale
23                end if
24
25             | afficher_Surete(any TParam, ..., any TMsg, any TSignal, ...) ->
26                if (va.Pth_moy_v.defaut != nil)
27                or (va.DPAXmoy_v.defaut != nil)
28             (...)
29          end case
```

Figure 76: The *user module* - an excerpt of LNT code

5.2.6 Summary of the Formal Model

The LNT formal model of the control room system currently contains nine main modules describing: some activities of the functional core, seven user interfaces and two main activities of the users. Table 5 summarizes the formal model. The eight first modules describe the case study, and are distributed in the ARCH components, in addition to the user module. Other modules implement auxiliary functions, and three .tnt/.fnt files contain C code used by the LNT model, mainly for optimization. In total, the formal model has 3430 lines of LNT code.

The formal model can then be used to perform formal verification. To this aim, we use the CADP toolbox to generate an LTS corresponding to the LNT formal model. Table 6 shows the size of the LTS generated from the LNT formal model. This LTS is used afterwards for formal verification.

5.2.7 Guidelines to Formally Model Interactive Systems

We summarize now the main lessons learned from the formal modeling of this case study, in order to provide insights to the modeling of interactive systems by formal methods:

1. Follow one architectural model (e.g., ARCH, PAC, etc.) to structure the model, and to deal with divergent concerns at different places;

2. Define the modules that will form the architectural model components, as well as how these modules exchange information;

Table 5: Summary of the formal model of the EDF system

#	ARCH component	File	Description	# loc
1	user interface	plant status	selection among six plant status	63
2	user interface	menu	selection among 45 menu options	257
3	functional core	reactor	modeling of 29 reactor parameters	365
4	functional core	generate signals	modeling of 37 reactor signals	336
5	functional core	scenarios	describes the five anomaly scenarios of reactor parameters	328
6	functional core	signal_details	describes details concerning failure signals	17
7	functional core	function	describes the reactor functions	46
8	dialog controller	selection	display of parameters and signals on seven UIs	217
9	(user)	user	selection of the plant status and monitoring of reactor parameters	374
10	(auxiliary file)	main	entry point of execution of the model	267
11	(auxiliary file)	library	type definitions	851
12	(auxiliary file)	library.tnt	internal optimizations	280
13	(auxiliary file)	reactor.tnt	internal optimizations	3
14	(auxiliary file)	reactor.fnt	internal optimizations	26
			TOTAL	**3430**

Table 6: Size of the LTS of the EDF system model

# states	# transitions
26 167 456	185 772 171

3. In particular, to define the UI modules, identify the UI zones with which the user can interact. The UI zones correspond to UI modules in the formal model;

4. Folding/unfolding of menus can be modeled by a set of rules which express when each menu option is available;

5. To model the functional core, pay attention to the aspects that will be subject to verification afterwards, i.e., a coverage large enough for the verification goals;

6. To model users, identify all actions that users can execute on the user interfaces.

5.2.8 Properties

We define a set of properties that can be verified over the LTS of the formal model. These properties are written using the MCL language. Using modalities (cf. Subsection 4.5.5 on

page 92), MCL permits interesting properties to be expressed[1]:

- *safety properties*: Informally, a safety property specifies that "something bad never happens". In MCL, this kind of property can be expressed using the pattern `[R]false`, where `R` is the formula expressing the condition we want to never happen;
- *liveness properties*: Informally, a liveness property specifies that "something good eventually happens". In MCL, this kind of property can be expressed using the pattern ⟨`R`⟩`true`, where `R` is the formula expressing the condition we want to eventually happen;
- *fairness properties*: These are similar to liveness properties, except that they express reachability of actions by considering only fair execution sequences. "A sequence is fair iff it does not infinitely often enable the reachability of a certain state without infinitely often reaching it." [Queille & Sifakis 1983]. In MCL, this kind of property can be expressed using the pattern `[R1]`⟨`R2`⟩`true`, where `R1` is potentially followed by another execution sequence satisfying the formula `R2`.

Using these MCL patterns, we specify a set of nine properties (Table 7), covering both classes we are considering in this thesis: *usability* and *functional* properties. The former aims at verifying whether the system follows ergonomic properties to ensure a good usability, and the latter aims at assessing that the system follows its expected behavior. In the following, we describe one property of each category, what they aim to verify, and how they are formalized in MCL. The formalization of the other properties is given in Appendix C. Considering the framework or ergonomic properties defined in [Abowd *et al.* 1992] all the *usability* properties we identified are classified as robustness properties, with the subcategory observability>reachability, which refers to the possibility of navigating through the observable states of the system.

Table 7: Summary of properties of the EDF system

Kind	**#**	**Description**
usability	1	"From any UI, one can always go directly to the main UI (i.e., without passing through any other UI)."
	2	"A UI is only accessible along the hierarchy of UIs."
	3	"One can always come back directly to the parent UI (i.e., without passing through any other UI before)."
	4	"From any state one can always reach any UI."
	5	"There is no deadlock in the system."
functional	6	"The UIs that display the signal details should be always accessible independently of the evolution of the reactor parameters."
	7	"Starting from any state, the reactor generates **all five** anomalies on each parameter."
	8	"Starting from any state, the reactor generates **at least** one anomaly on each parameter."
	9	"Starting from any state, the reactor generates **at least** one anomaly on each parameter belonging to a specific plant status."

[1] http://cadp.inria.fr/man/evaluator4.html, section "Overview of the MCL Language"

- Property 1) "From any UI, one can always go directly to the main UI (i.e., without passing through any other UI).":

  $[\ true^*\ ]$
  $\langle\ (\ not\ UIs\ )^*\ .\ 'GLOBAL_SYNTHESIS'\ \rangle\ true$

  This formula may be read as:

  From every reachable state

  ⟨there exists a sequence of steps...

  ...not passing through any UI...

  ...and leading to the GLOBAL_SYNTHESIS UI ⟩

  This property ensures that, in all user interfaces, there is always the possibility to come back to the main UI (called *global synthesis*, Figure 50) with one single user interaction, i.e., without the need to access intermediate UIs beforehand.

- Property 7) "Starting from any state, the reactor generates **all five** anomalies on each parameter.":

  $macro\ FAILURES_COVERAGE\ (P)\ =$
  $[\ true^*\ ]$
  $\langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ DEP_HAUT.^{*\prime}\ \rangle\ true$
  and
  $\langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ DEP_BAS.^{*\prime}\ \rangle\ true$
  and
  $\langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ DEP_GRAD.^{*\prime}\ \rangle\ true$
  and
  $\langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ INV_MESURE.^{*\prime}\ \rangle\ true$
  and
  $\langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ PERTE_RED.^{*\prime}\ \rangle\ true$
  $end\ macro$

  $FAILURES_COVERAGE\ (\ 'TPTH_MOY'\ )$
  and
  $FAILURES_COVERAGE\ (\ 'TDPAXMOY'\ )$
  ...
  (it continues for all reactor parameters)

  This property is expressed by means of a macro named $FAILURES_COVERAGE$. It verifies that the five anomaly scenarios (i.e., *threshold overflow* – "dépassement haut", *threshold underflow* – "dépassement bas", *gradient excess* – "dépassement gradient", *invalid*

measurement – "invalidité de mesure", and *loss of redundancy* – "perte de redondance") are simulated over each reactor parameter (e.g., `Pth moy` parameter).

The properties proposed here cover ergonomic aspects of the system under verification, as well as the some requirements of the modeled system. More properties could be formalized and verified. The main goal is a proof of concept of the approach: to demonstrate that both kinds of properties can be formalized and verified over the system formal model using our approach.

5.2.9 Formal Verification

CADP provides the OCIS[2] tool (*Open/Caesar Interactive Simulator* - Figure 65) for step-by-step simulation with backtracking. It allows one to simulate the formal model, and to test it interactively while an execution tree is created and displayed on the UI of the tool. This simulation allows one to explore all the possible executions of the model. We use this tool to explore and simulate our formal model step by step.

Another tool available in CADP is the EVALUATOR 4.0[3] model checker (for handling MCL formulas [Garavel *et al.* 2013]). It evaluates MCL formulas over the LTS of the formal model. In the end, EVALUATOR 4.0 may provide a diagnosis of the evaluation, i.e., an excerpt of the system behavior that illustrates either the validity (example) or the invalidity (counter-example) of the formula. Applying the model checking part of our global approach (cf. Section 4.3 on page 85), the nine properties are satisfied over the formal model.

The formal verification of the LNT model allows the model to be used with other purposes. In the next section we describe the use of the formal model to validate part of an industrial system called ADACS-N™.

5.3 Propositions to Connect to Industrial Systems

This section describes some possibilities to connect formal specifications to industrial systems, and a connection of the formal model of the EDF system to an industrial system called ADACS-N™.

5.3.1 Problem Definition

An interactive system is expected to implement a specification. However, a chance exists that a system does not implement the specifications as expected, thus, a way to verify whether the implementation follows the specifications is needed.

In the context of the Connexion Project, our research laboratory (LIG), Atos Worldgrid and EDF conduct in-depth investigation of (and explore differently) the case study described in Chapter 3: EDF prototypes a control room system, aiming at providing a better overview of plant functions and system status [Chériaux *et al.* 2012]. Such system is the reference to guide Atos Worldgrid and LIG in their own contributions within the project. LIG proposes a prototype of the same system, applying recent advancements of plasticity to it (cf. Section 3.4 on page 76). In addition, we develop the formal model of the EDF system which we have just described (cf. Section 5.2 on page 96).

[2] `http://cadp.inria.fr/man/ocis.html`
[3] http://cadp.inria.fr/man/evaluator4.html

On the other hand, Atos Worldgrid implements on their industrial product called ADACS-N^{TM} (cf. Section 3.5 on page 80) some user interfaces of the EDF system, following the enhancements proposed by LIG. In this context, a question that raises is whether ADACS-N^{TM} correctly implements the EDF specifications or not. The EDF specifications consist of informal requirements given by EDF during project meetings, exchange of e-mails with questions/answers, some documentation, print screens, and a video illustrating some functionalities of the EDF system. This makes the basis from which ADACS-N^{TM} implements the EDF system. However, a chance exists that the ADACS-N^{TM} part implementing the EDF system could be improved, by making specifications more precise.

5.3.2 Using the Formal Model to Cross Check the Implementation

A formal specification can be used to validate an implementation. 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" [1012-2004 2005].

In the context of the Connexion Project, the LIG laboratory is provided with the same EDF specifications as ADACS-N^{TM} to create the formal specification of the case study. Thus, we propose to use the formal specification to cross check the ADACS-N^{TM} implementation (Figure 77). The ultimate goal is to give clues about whether ADACS-N^{TM} implements the EDF specifications as expected. If the formal specification and ADACS-N^{TM} are inter-operable, then it is **probable** that both the formal model and ADACS-N^{TM} interpreted the EDF specifications in the same way, and that they are both correct. Since the formal specification is a model of the real system, as for all model-based approaches, there is no guarantee that the model is correct, therefore, the approach cannot fully determine whether ADACS-N^{TM} implements the EDF specifications as expected or not. It can, yet, 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 specifications in the same way, there is a high probability that they are both correct (not neglecting the possibility that both interpret the specifications equally wrong, if the specifications are ambiguous). Finally, if the formal specification and the implementation diverge, then at least one of both groups of people misunderstood the specifications.

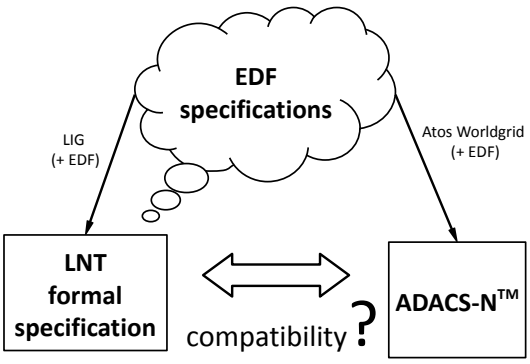


Figure 77: Models of the case study

We propose and compare three different ways to connect the formal specifications to the ADACS-N^{TM} system: *analysis of traces*, *test case generation*, and *co-simulation*.

5.3.3 Proposition 1: Analysis of Traces

Analysis of traces consists in interpreting log files and verifying whether the formal model can simulate the same sequence of traces or not. With this purpose, a translation of the log files into a format that can be treated by the formal model is required. Once the scenarios are transformed into this format, one can verify if a given sequence is included in the set of sequences of the formal model, meaning that the formal model simulates the scenario in the same way as the industrial system. In this proposition, the translation of the log files into a format that can be treated by the formal model is needed, so that the analysis can be automatized.

ADACS-N^{TM} includes a log mechanism that records in execution trace files information such as: the acquisition of reactor parameter values, the display of these values on the user interfaces, and the user interactions with the UIs. These execution trace files could be used in this proposition (Figure 78).

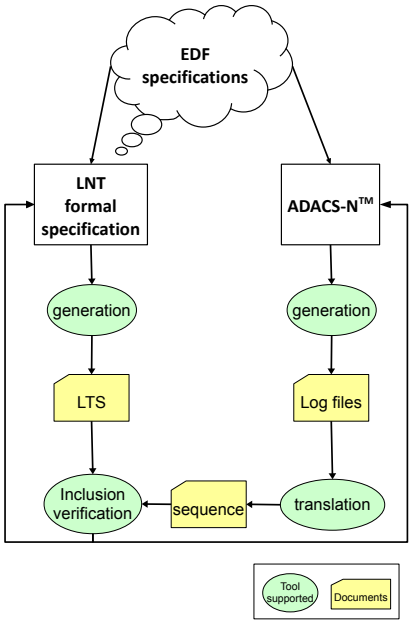


Figure 78: Analysis of traces proposition

5.3.4 Proposition 2: Test Case Generation

In *test case generation* (applied to the Connexion Project in Figure 79), the formal model can be used as a basis to derive test suites. These test cases can be executed on the industrial system, which provides the real inputs. A test execution engine is required to execute the test cases on the industrial system and to implement the test oracles that output the test verdicts. The

ultimate goal in this proposition is to generate test cases that are feasible, meaning that they can be executed over the industrial system. Once a feasible test is executed over the industrial system and do not pass, it can indicate a flaw either in the industrial system or in the formal model. In this proposition, two things need to be developed: the test case generation from the formal model and the test execution engine.

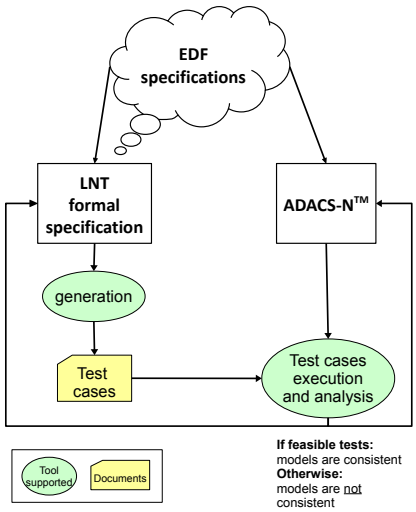


Figure 79: Test case generation proposition

5.3.5 Proposition 3: Co-Simulation

In *co-simulation*, both the formal model and the industrial system are executed in parallel. The outputs of one are connected to the inputs of the other and vice versa. This proposition requires either a translation in both ways (i.e., from the industrial system to the formal model and vice versa), or the formalization of a language common to both the industrial system and the formal model. If both the industrial system and the formal model can execute in parallel in co-simulation, it means that both are aligned in the way they implement the specifications. For instance, in Figure 80 this proposition is applied to the context of Connexion Project.

5.3.6 Rationale of the Chosen Proposition

We compare now the three propositions, in order to choose one of them for further investigations. Two criteria are analyzed: the degree of coupling between ADACS-N^{TM} and the formal model required for each proposition, and the availability of the industrial partner for each proposition.

The co-simulation proposition requires ADACS-N^{TM} and the formal model to be highly coupled: both are able to run in parallel only if they agree on the way the EDF specifications are implemented. For the test case generation proposition, both ADACS-N^{TM} and the formal model are less coupled: they are executed independently, the formal model generates test cases that can be executed over ADACS-N^{TM}. However, some knowledge about the way the EDF specifications are implemented in each model is still required, in order to implement the test oracles. The

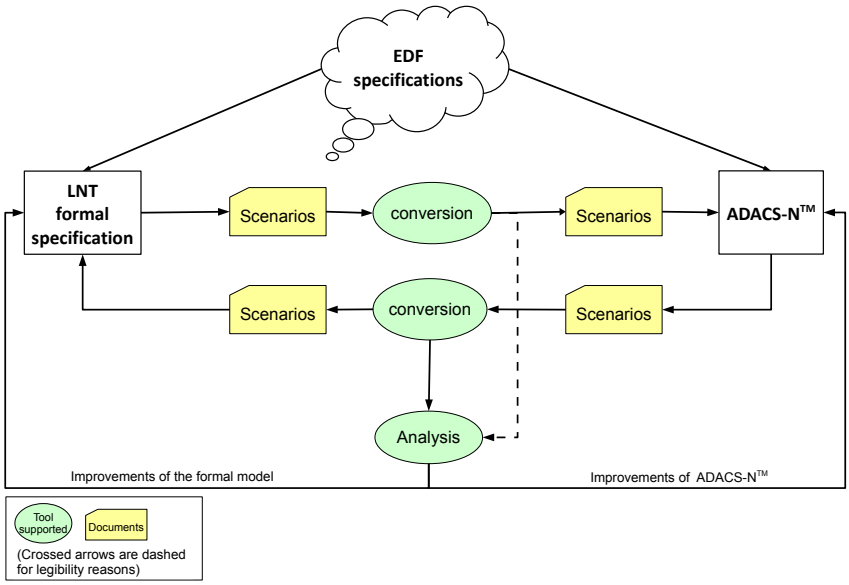


Figure 80: Co-simulation proposition

analysis of traces proposition requires the least degree of coupling between ADACS-N^{TM} and the formal model. Both run completely independently, the logs of ADACS-N^{TM} being analyzed over the formal model.

Regarding the availability of the industrial partner, co-simulation is the most effort-consuming: a reasonable comprehension of both ADACS-N^{TM} and the formal model is required, as well as the development of a tool to translate ADACS-N^{TM} outputs into inputs of formal model and vice-versa. Test case generation is less effort-consuming. It requires an initial effort to understand both ADACS-N^{TM} and the formal model, to identify test cases, to generate them from the formal model, and to implement a means to execute them over ADACS-N^{TM}. Finally, analysis of traces is the least effort-consuming approach, therefore, the fastest to accomplish. ADACS-N^{TM} already generates log files, on which only a few modifications are necessary and a translation into a format that can be treated by our formal model.

Due to time constraints of the industrial partners, **analysis of traces** is the chosen proposition, and it is detailed in the following subsections. The other two propositions, i.e., test case generation and co-simulation, are not further investigated in this thesis.

5.4 On the Connection to an Industrial System

To pave the way to integrate the formal model and ADACS-N^{TM}, an analysis is performed over both artifacts. Such preliminary analysis is beneficial to the formal model realism: it improves the formal model in several directions and approximates it to the ADACS-N^{TM} implementation of the EDF system.

5.4.1 Improvements in the Formal Model

The following modifications are implemented in the formal specification before the connection to ADACS-N^{TM}:

1. Alignment of the list of reactor parameters: 27 new parameters are added and six unused parameters are removed, thus, increasing the number of parameters from 29 to 50;

2. Adjustment in the names of parameters: some parameters have different names in the EDF specifications and in ADACS-N^{TM}. We add references to both terminologies in the formal model;

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 thresholds and two inferior thresholds, increasing the number of thresholds from 1 to 3 at each extremity (Figure 81). When the value of the parameter cross these thresholds, *alerts* and/or *alarm conditions* are triggered on the system;

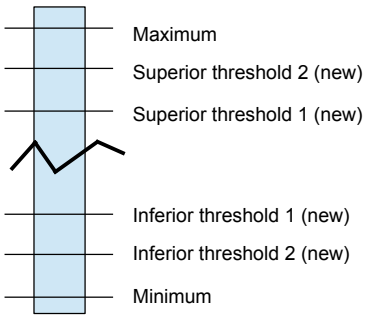


Figure 81: New thresholds in the reactor parameters

5. Addition of two new anomaly scenarios, *super threshold overflow* ("dépassement très haut") and *super threshold underflow* ("dépassement très bas"), increasing the number of anomaly scenarios from five to seven. Such anomalies are triggered in a reactor parameter when its value exceeds its *superior/inferior* thresholds n.2 (Figure 81), in contrast to the *threshold overflow* and *threshold underflow* anomalies, already present in the model, and triggered when a reactor parameter value exceeds its *superior/inferior* thresholds n.1. The *maximum* and *minimum* thresholds are not expected to be exceeded;

6. Alignment of the list of failure signals per UI, as well as the menu on the left of the UIs. For instance, on the LIG prototype (Figure 54b), the Global Synthesis UI contains seven failure signals: (1) *reactivity* – "réactivité", (2) *production*, (3) *electric* – "électricité", (4) *fan & air conditioning* – "clima. et ventil.", (5) *core cooling* – "refroidissement", (6) *pneumatic* – "air comprimé", and (7) *fire protection* – "incendie". In contrast, the same UI in the ADACS-N^{TM} platform (Figure 58b) contains eight failure signals: the failures (1), (3), (5), (6), and in addition, *confinement*, R, A, and V.

Table 8 gives figures about the new version of the formal model. All modules are affected to some extent (except `reactor.tnt` and `reactor.fnt`) Specially, 50 reactor parameters are now modeled (line 3), and seven anomaly scenarios are simulated (line 5). The new formal model has 4444 lines of code.

Table 8: Summary of the new version of the formal model, to connect to ADACS-N™

#	ARCH component	File	Description	# loc
1	user interface	plant status	**1** plant status is always selected	19
2	user interface	menu	selection among **7** menu options	234
3	functional core	reactor	modeling of **50** reactor parameters	404
4	functional core	generate signals	modeling of **12** reactor signals	178
5	functional core	scenarios	describes the **seven** anomaly scenarios of reactor parameters	451
6	dialog controller	selection	display of parameters and signals on seven UIs	90
7	(user)	user	selection of the plant status and monitoring of reactor parameters	211
8	(auxiliary file)	scheduler	orchestration of the exchange of data	93
9	(auxiliary file)	main	entry point of execution	111
10	(auxiliary file)	library	type definitions	2191
11	(auxiliary file)	library.tnt	internal optimizations	433
12	(auxiliary file)	reactor.tnt	internal optimizations	3
13	(auxiliary file)	reactor.fnt	internal optimizations	26
			TOTAL	**4444**

Table 9 shows the size of the LTS generated from this LNT formal model, using CADP. This LTS is smaller than the previous version of the formal model (i.e., 26 167 456 states and 185 772 171 transitions, Table 6) mainly because several aspects of the formal model were deactivated, since they were not evaluated in this connection. Section j), on page 127, lists the aspects that were deactivated in the formal model, for instance, three anomaly scenarios are simulated in the previous version of the formal model and deactivated in this new one, namely: *gradient excess* ("dépassement gradient"), *loss of redundancy* ("perte de redondance"), and *invalid measurement* ("invalidité de mesure").

Table 9: Size of the LTS of the new version of the formal model

# states	# transitions
17 832 752	31 225 752

5.4.2 Improvements in the ADACS-N™ Platform

Also before to connect the formal model and ADACS-N™, Atos Worldgrid improved the ADACS-N™ platform as follows:

1. generation of failure signals once a reactor parameter receives values that are beyond its thresholds;

2. addition of two failure signals in the *Reactivity* ("Réactivité") user interface (i.e., *boron concentration* and *boration/dilution*, in addition to *rods position* and *reactor control* signals, already in the ADACS-N™ UI);

3. 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;

4. addition in the log: user interactions with the user interfaces;

5. addition in the log: all UI status changes for inputs, threshold overshooting, triggered alerts;

6. generation of a unique log file, including all logged events, sorted by timestamps.

5.4.3 Connection between ADACS-N™ and the Formal Model

The alignment of the formal model with ADACS-N™ allows further comparison between them. In the following subsections we detail the trace analysis done in ADACS-N™.

a) Description of the Approach

The whole approach for comparing both implementations is described in Figure 82. The entry points of the approach are both the LNT formal model and ADACS-N™. Both are connected by means of a Parser, which is in dark green in Figure 82 to indicate that this tool is developed in the context of this thesis. The other tools in light green are either provided by the CADP toolbox or by ADACS-N™.

On the left of the figure, the LNT formal model describing the EDF system is automatically transformed into an LTS (*Labeled Transition System*) using CADP.

The right part of Figure 82 consists in the stimulation of ADACS-N™. ADACS-N™ can either run autonomously, stimulated by several anomaly scenarios, or it can be fed by other platforms. With this goal, a simulation mode allows ADACS-N™ to be connected to a stimulator from which ADACS-N™ receives input data. These input files are manually created, and they contain anomaly scenarios in parameters of the nuclear unit. ADACS-N™ receives these data from the stimulator and reacts accordingly, following several physics laws. The results of such calculations are surfaced to ADACS-N™ user interfaces, allowing users to be aware of the current status of the nuclear unit and to take actions correspondingly, by interacting with the ADACS-N™ UIs.

The results of ADACS-N™ calculations, the information displayed on the user interfaces, and the user interactions with the UIs are logged into a unique log file. Therefore, such log

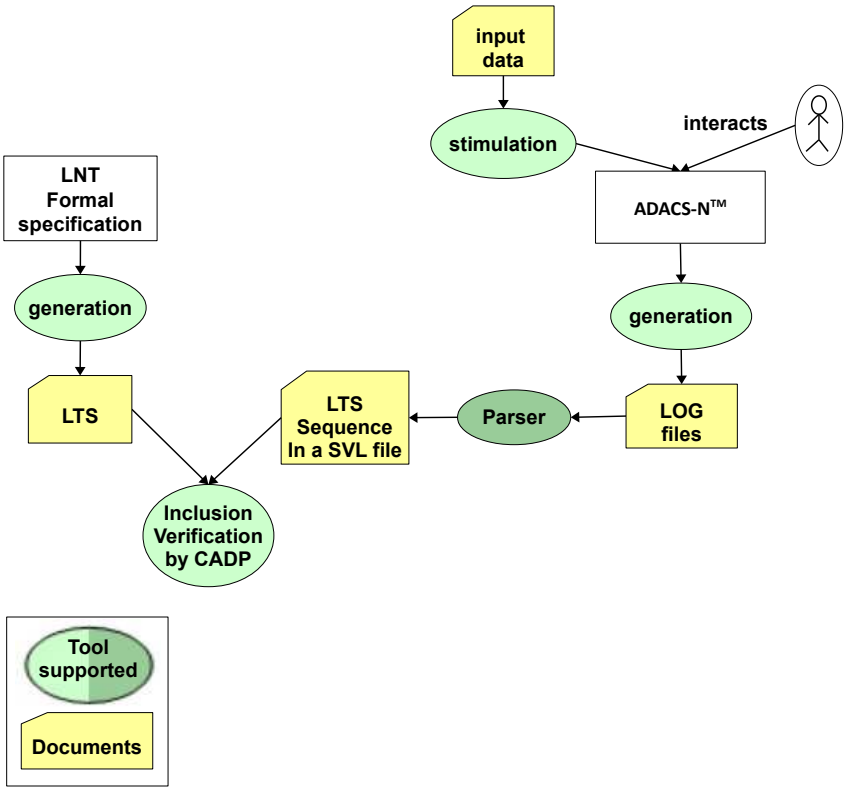


Figure 82: Analysis of traces in details

file consists in a complete trace of an anomaly scenario in ADACS-N^{TM}: anomaly in a reactor parameter, followed by the display on the UI, and the user interactions.

The horizontal part of the Figure 82, which links the LNT formal model and ADACS-N^{TM}, consists of a translation of ADACS-N^{TM} log files into a trace that be searched in the LTS of the formal model. A Parser is developed in Java to automatize this translation. It takes as input an ADACS-N^{TM} log file, extracts the lines containing the scenario, and translates them into a sequence of transition labels of the LTS.

The last step of the approach is the verification that the sequence os labels extracted from ADACS-N^{TM} log file is included in the LTS of the formal model (Figure 83). If the trace is found, it means that the formal model simulates the scenario in the same way as ADACS-N^{TM}. EVALUATOR 4.0 model checker[4] (available in the CADP toolbox) is used to perform such verification.

b) Simulated Scenarios

In this study, four kinds of scenarios are analyzed: *threshold overflow*, *threshold underflow*, *super threshold overflow*, and *super threshold underflow* (cf. Table 3).

[4]http://cadp.inria.fr/man/evaluator4.html

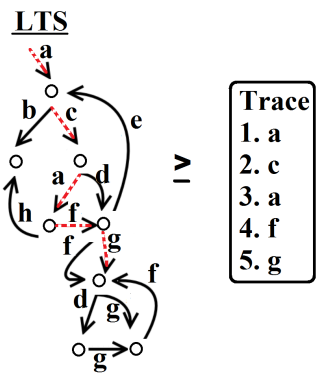
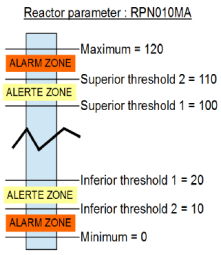


Figure 83: Example of inclusion verification of traces

Figure 84 illustrates one of those scenarios: *super threshold underflow* ("dépassement très bas") on the `RPN010MA` reactor parameter. The parameter has several attributes such as: a minimum value, two inferior thresholds, a mean value, two superior thresholds and a maximum value. The *super threshold underflow* scenario has 11 instants: it initializes the parameter with its mean value, decreasing it progressively according to the formulas described for each instant. At the instant n.4, the parameter value exceeds its first inferior threshold. At this point, a *threshold underflow* ("dépassement bas") anomaly is triggered on the reactor parameter, and an alert signal is generated. The parameter value continues to decrease, until it exceeds its second inferior threshold at instant n.6, triggering a *super threshold underflow* ("dépassement très bas") and an alarm condition signal. The parameter value then progressively increases until it reaches its mean value again.



Reactor parameter information : « RPN010MA »			
min	inf_thres2 (i2)	inf_thres1 (i1)	mean (m)
0	10	20	60
sup_thres1 (s1)	sup_thres2 (s2)	max	
100	110	120	

Scenario7 = super threshold underflow (11 instants)											
instant	1	2	3	4	5	6	7	8	9	10	11
formula	m	i1+m/2	i1+10	i1-1	i1+i2/2	i2-1	i1+i2/2	i1-1	i1+10	i1+m/2	m
values	60	40	30	19	15	9	15	19	30	40	60

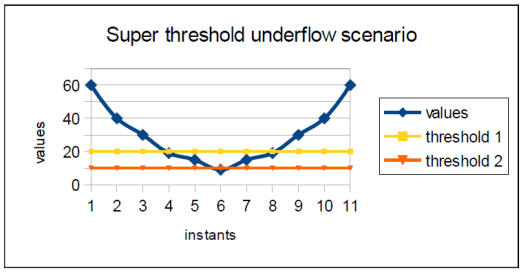


Figure 84: The *super threshold underflow* anomaly scenario on a reactor parameter

c) ADACS-N^{TM} Input File

In order to stimulate ADACS-N^{TM} objects, input files containing several scenarios are manually created. Figure 85 illustrates an example of such files.

```
CM ******************************;
CM *  SCENARIO 1 MODELE FORMEL  *;
CM ******************************;
DEBUT_SCENARIO SCEN1;

GEN_MSG NEANT CCT INFORMATION_N1_NON_DATE INFO_NON_DATE  1 (
1RGL001MN  1 ( VALUE ANA  160.0 0));
C_DELAY 10.0;

GEN_MSG NEANT CCT INFORMATION_N1_NON_DATE INFO_NON_DATE  1 (
1RGL001MN  1 ( VALUE ANA  190.0 0));
C_DELAY 10.0;

GEN_MSG NEANT CCT INFORMATION_N1_NON_DATE INFO_NON_DATE  1 (
1RGL001MN  1 ( VALUE ANA  201.0 0));
OPER_ACK ;

GEN_MSG NEANT CCT INFORMATION_N1_NON_DATE INFO_NON_DATE  1 (
1RGL001MN  1 ( VALUE ANA  190.0 0));
C_DELAY 10.0;

GEN_MSG NEANT CCT INFORMATION_N1_NON_DATE INFO_NON_DATE  1 (
1RGL001MN  1 ( VALUE ANA  160.0 0));
C_DELAY 10.0;

GEN_MSG NEANT CCT INFORMATION_N1_NON_DATE INFO_NON_DATE  1 (
1RGL001MN  1 ( VALUE ANA  120.0 0));

FIN_SCENARIO;
```

Figure 85: Example of an ADACS-N^{TM} input file

The input file contains the following commands:

- CM: introduces comments into the file;
- GEN_MSG: stimulates an ADACS-N^{TM} object (e.g., `1RGL001MN`) with an input value (e.g., 160.0). An ADACS-N^{TM} object corresponds to a reactor parameter in our modeling;
- C_DELAY: makes the stimulator wait a certain amount of time before executing the next line of the input file; this command is useful for observing the display of the ADACS-N^{TM} object output on the user interfaces;
- OPER_ACK: indicates that the stimulator should wait for a RETURN key press in order to continue the execution of the input file. This command is useful to allow the user to interact with the user interface once an anomaly is surfaced from the ADACS-N^{TM} objects on the UI. Once the interactions take place (and by consequence are recorded into the log file), the execution of the scenario by the stimulator can continue.

The input file illustrated in Figure 85 consists in a stimuli of the ADACS-N^{TM} object identified by `1RGL001MN`. It progressively changes its value starting from 160.0, increasing up to 201.00, and decreasing down to 120.0. The stimulator executes each line, sending to ADACS-N^{TM} the lines identified by GEN_MSG. Depending on the calculations of ADACS-N^{TM} objects, a given value may generate alerts, or alarm conditions, or may invalidate the object, etc. For example, once the `1RGL001MN` object gets the value 201.0, the object outputs an alarm condition because such value exceeds the maximum threshold set to this object (i.e., 200.0). Such alarm condition is displayed in the ADACS-N^{TM} user interface in the visual component representing the `1RGL001MN` object, to inform the user that an anomaly occurred in this nuclear unit parameter (in this case, it achieved a value bigger than its maximum threshold). The user, in turn, interacts with the system, by navigating through the UIs in order to have more details about the anomaly.

The ADACS-N^{TM} object calculations resulted from the stimulation, the display of these information on the user interfaces and the user interactions are logged into trace files.

d) ADACS-N^{TM} Log File

Three kinds of information are recovered from the log files (Figure 86):

- Highlighted in red, and identified by the keyword "Adacs.Recorder.TOX", these lines represent the calculation results of the stimulated object. In the first line of the log, the value of the `1RGL001MN` object changed to 160.0. Line n.4 indicates that the ADACS-N^{TM} object entered in a state in which its value is over the maximum value (identified by the keyword HIGH_BEGIN), due to the change in the value in the previous line. For this reason, the ADACS-N^{TM} object outputs an alarm condition (line n.5).

- The display of this alarm condition on the user interface is illustrated in the line n.6, identified by the keyword "Adacs.Recorder - FPMS Action" in blue. This line indicates that one alarm condition is displayed in a signal called *reactivity* (réactivité) on the user interface.

- The user perceives the notification on the user interface, and interacts with the menu options in order to search the source of the displayed anomaly. Such interactions are also identified in the log file by the keyword "Adacs.Recorder - FPMS Action", followed by the keyword "<Bouton MENU>" in green (line n.11), indicating that the user chose the *reactivity* menu option.

Each user interaction with the menu options displays a different user interface. The display of such user interfaces is also recorded in the log. Specifically, we are interested in which signals are displayed in each user interface. For instance, line n.7 indicates that the user chose the *safety* ("sûreté") menu option. The next 3 lines (i.e., 8-10) indicate that the corresponding user interface has 3 signals (*core cooling* – "refroidissement", *reactivity* – "réactivité", and *confinement*, and an alarm condition was displayed in the *reactivity* (réactivité) signal (i.e., "nThresholds=1" in line n.9).

ADACS-N^{TM} log files in such format are translated into a trace that be searched in the LTS of the formal model. A Parser was developed with this goal.

```
1  2015-03-13 Ada - 2015-03-13 14:11:16:801  Adacs.Recorder.TOX DIVERS TR  VALUE_CHANGE 1RGL001MN   1.60000E+02
2  2015-03-13 Ada - 2015-03-13 14:11:26:810  Adacs.Recorder.TOX DIVERS TR  VALUE_CHANGE 1RGL001MN   1.90000E+02
3  2015-03-13 Ada - 2015-03-13 14:11:36:818  Adacs.Recorder.TOX DIVERS TR  VALUE_CHANGE 1RGL001MN   2.01000E+02
4  2015-03-13 Ada - 2015-03-13 14:11:36:818  Adacs.Recorder.TOX DIVERS TR  OVERSHOOTING HIGH_BEGIN 1RGL001MN
5  2015-03-13 Ada - 2015-03-13 14:11:36:818  Adacs.Recorder.TOX DIVERS TR  ALARM_APPEAR 1RGL001MN  HIGH
6  2015-03-13 Adacs.Recorder - FPMS  Action : Réactivité Param :  nSignals=0, nThresholds=1
7  2015-03-13 Adacs.Recorder - FPMS  Action : <Bouton MENU> Param : Choix de l'élément <SURETE>
8  2015-03-13 Adacs.Recorder - FPMS  Action : Refroidissement Param :  nSignals=0, nThresholds=0
9  2015-03-13 Adacs.Recorder - FPMS  Action : Réactivité Param :  nSignals=0, nThresholds=1
10 2015-03-13 Adacs.Recorder - FPMS  Action : Confinement Param :  nSignals=0, nThresholds=0
11 2015-03-13 Adacs.Recorder - FPMS  Action : <Bouton MENU> Param : Choix de l'élément <REACTIVITE>
12 2015-03-13 Adacs.Recorder - FPMS  Action : Pos Grappes Param :  nSignals=0, nThresholds=1
13 2015-03-13 Adacs.Recorder - FPMS  Action : Pilotage réacteur Param :  nSignals=0, nThresholds=0
14 2015-03-13 Adacs.Recorder - FPMS  Action : Boric/Dilut Param :  nSignals=0, nThresholds=0
15 2015-03-13 Adacs.Recorder - FPMS  Action : Conc. bore Param :  nSignals=0, nThresholds=0
16 2015-03-13 Adacs.Recorder - FPMS  Action : <Bouton MENU> Param : Choix de l'élément <POSITION GRAPPES>
17 2015-03-13 Adacs.Recorder - FPMS  Action : <Bouton MENU> Param : Choix de l'élément <SYNTHESE GLOBALE>
18 2015-03-13 Adacs.Recorder - FPMS  Action : Confinement Param :  nSignals=0, nThresholds=0
          (...)
```

Figure 86: An ADACS-N^{TM} log file (adapted)

e) LTS Sequence

First, the Parser generates a sequence of transition labels such as the one illustrated in Figure 87. Each line corresponds to a labeled transition in an LTS, and should be enclosed by “”. Figure 87 identifies the three kinds of actions we simulate in the scenarios: lines 1 and 2, in red, represent object calculations resulted from the stimulation, i.e., the evolution of reactor parameters over time; lines 7 and 8, in blue, represent the display of such information on the user interfaces; and line n. 12, in green, represents the user interactions with the menu options. For legibility reasons, only the beginning of the transition labels are illustrated in this picture.

Secondly, the Parser generates a MCL formula containing the sequence of transition labels. The MCL formula has the following format:

$$\langle\ true^* \ .\ L_1\ .\ L_2\ .\ \ldots\ .\ L_n\ \rangle\ true^* \tag{5.1}$$

This formula represents a liveness property (cf. Section 5.2.8 on page 106), and 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 are the transition labels initially generated (Figure 87) Intuitively, this MCL formula verifies whether the sequence of steps representing the scenario can be found in the LTS or not, which in a positive case indicates that the formal model implements the scenario in the same way as ADACS-N^{TM}.

Finally, the Parser writes this MCL formula as a property in a SVL script, to be checked automatically. Figure 88 illustrates an example of a property. It has a name (line 1), an optional comment (line 2), the file name containing the LTS in which the model checker should verify the property (line 4), the MCL formula containing the sequence representing the ADACS-N^{TM} scenario (lines 5-32), and finally, the command `expected TRUE`, indicating to the model checker that the property should be evaluated to TRUE to pass.

```
1  "VOIR_PARAMS_REACTEUR..."
2  "VOIR_SIGNAUX"
3  "VOIR_PARAMS_REACTEUR..."
4  "VOIR_SIGNAUX"
5  "VOIR_PARAMS_REACTEUR..."
6  "VOIR_SIGNAUX"
7  "VOIR_VUE !SYNTHESE_GLOBALE"
8  "VOIR_SELECTION !AFFICHER_VUE..."
9  "VUE_SURETE"
10 "VOIR_VUE !SURETE"
11 "VOIR_SELECTION !AFFICHER_VUE..."
12 "VUE_SURETE_REACTIVITE"
13 "VOIR_VUE !SURETE_REACTIVITE"
14 "VOIR_SELECTION !AFFICHER_VUE..."
15 "VUE_SURETE_REACTIVITE_POSITIONGRAPPES"
16 "VUE_SYNTHESE_GLOBALE"
17 "VOIR_VUE !SYNTHESE_GLOBALE"
18 "VOIR_SELECTION !AFFICHER_VUE..."
19 "VOIR_PARAMS_REACTEUR..."
20 "VOIR_SIGNAUX"
21 "VOIR_VUE !SYNTHESE_GLOBALE"
22 "VOIR_SELECTION !AFFICHER_VUE..."
23 "VOIR_PARAMS_REACTEUR..."
24 "VOIR_SIGNAUX"
25 "VOIR_PARAMS_REACTEUR..."
26 "VOIR_SIGNAUX"
```

Figure 87: A sequence of transition labels

f) Parser

The Parser is implemented in Java and consists of four classes (Figure 89). The `Parser.java` class reads the log file line by line and looks for the keywords that identify: changes in ADACS-N^{TM} objects, displays on the user interfaces, or user interactions. This class sends the entire line to the corresponding class (either `ReactorEvolution.java`, or `UIReaction.java`, or `OPAction.java`). Each one of such classes implements the rules of transformation of these three kinds of actions simulated by the scenarios, and returns the translated line to the main class. Finally, the main class writes such lines in a SVL script.

Some transformations are needed to translate ADACS-N^{TM} log files into a trace that be searched in the LTS of the formal model. For instance, some information is need to be grouped, others need to be split, etc. Figure 90 illustrates how both files correspond to each other: there are cases in which one line in the ADACS-N^{TM} log corresponds to one line in the generated script, cases in which n lines in the log correspond to one line in the generated script, cases in which one line in the log corresponds to n new lines, and finally cases in which n lines in the log correspond to m new lines. This means that both ADACS-N^{TM} and the LNT formal model represent some information in different ways.

Table 10 gives a few figures about the code of the Parser.

```
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 88: A property containing a sequence of transition labels

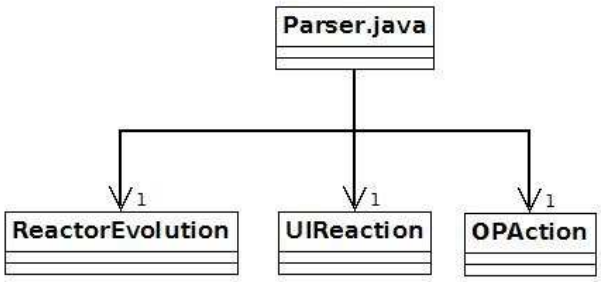


Figure 89: The `Parser` class diagram

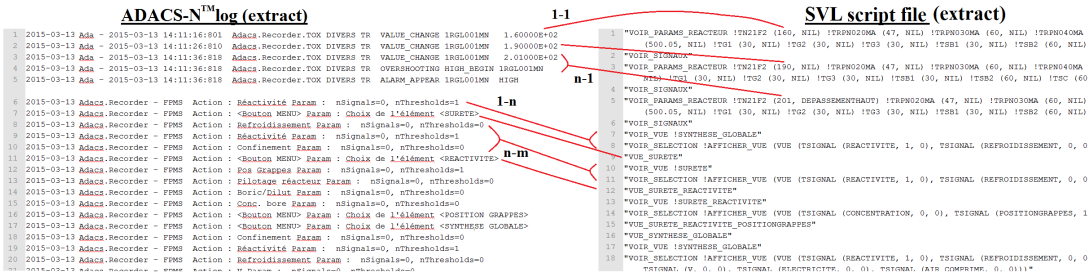


Figure 90: Correspondence between the ADACS-N™ log files and the translated traces

Table 10: Summary of the Parser

#	.java File	# loc
1	Parser	217
2	ReactorEvolution	136
3	UIReaction	143
4	OPAction	51
5	ParserGUI	92
	TOTAL	**639**

g) Inclusion Verification

The inclusion verification consists in verifying whether the sequence of labels extracted from ADACS-N™ log files are included in the LTS of the formal model or not. Using the EVALUATOR[5] tool (available in CADP), we verify the properties expressing the extracted ADACS-N™ scenarios over the LTS of our formal model. Figure 91 illustrates an example of verification. The property passes if the formula is satisfied over the LTS, or it fails otherwise.

```
1 property P1
2  | SCENARIO N.1
3
4 PASS
```

Figure 91: Property verification

h) Coverage of the Validation

Figure 92 illustrates the coverage of the validation of the ADACS-N™ part implementing the EDF system. The LNT model we use simulates anomalies over 50 reactor parameters (# lines of the table). In ADACS-N™, 20 of such parameters (in light gray in the table) have acquired data as input, and can be stimulated with input values (the other 30 parameters are calculated in function 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.

[5]http://cadp.inria.fr/man/evaluator4.html

The LNT model simulates seven anomalies over reactor parameters (i.e., *threshold overflow* – "dépassement haut", *threshold underflow* – "dépassement bas", *gradient excess* – "dépassement gradient", *loss of redundancy* – "perte de redondance", *invalid measurement* – "invalidité de mesure", *super threshold overflow* – "dépassement très haut", and *super threshold underflow* – "dépassement très bas"). ADACS-N^{TM} simulates four of them: *threshold overflow*, *threshold underflow*, *super threshold overflow*, and *super threshold underflow*. ADACS-N^{TM} provides us with 38 log files, each one containing an anomaly scenario in one parameter. These samples are highlighted in Figure 92 as dark gray cells. For instance, in one scenario, a *threshold underflow* is simulated in the `RPN010MA` parameter (line 9). Our formal model simulates the seven anomalies 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}.

i) Results of the Validation

We use the Parser to generate a property describing each one of the 38 anomaly scenarios we are provided with. We apply the *model checking* part of our global approach (cf. Section 4.3 on page 85) to verify the satisfiability of these properties. The EVALUATOR tool is used in order to verify the satisfiability of the 38 scenarios over the LTS. Preliminary verifications shave how that the traces are not included in the LTS. A deeper analysis shows that ADACS-N^{TM} and the formal model are divergent in the way anomalies in reactor parameters are synthesized in the failure signal UI. This is due to a lack of alignment in the way the EDF specifications were communicated to Atos Worldgrid and the LIG laboratory, making both implementations diverge.

Once this first conclusion is observed, we modify the formal model to synthesize anomalies in the failure 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, and the total time of inclusion verification is four minutes in average.

We summarize below some figures of the verification:

- # ADACS-N^{TM} trace files containing the anomaly scenarios: 38;
- # total of lines of the 38 trace files: 1710;
- # generated SVL file: 1;
- # lines of SVL file: 1700;
- generation time of the SVL file of properties: 3s;
- inclusion verification time: ± 4 min;
- # loc Parser in Java: 639.

The following information is present in ADACS-N^{TM} **and** in the LNT formal model. Thus, the inclusion verification enables one to check that these aspects of the EDF system are implemented in the same way in both ADACS-N^{TM} and the formal model:

Reactor Parameter	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 4	Cycle 5	qt. anomalies
1 PTHMOY	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	7
2 CB	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	7
3 R1	3 DG	4 PR	5 IM	6 **DTH**	7 **DTB**	1 **DH**	2 **DB**	7
4 R2	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 **DB**	3 DG	7
5 G21	5 IM	6 **DTH**	7 DTB	1 DH	2 DB	3 DG	4 PR	7
6 G22	6 DTH	7 **DTB**	1 DH	2 DB	3 DG	4 PR	5 IM	7
7 NRCV	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7
8 NRCV2	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	7
9 RPN010MA	2 **DB**	3 DG	4 PR	5 IM	6 **DTH**	7 **DTB**	1 **DH**	7
10 RPN020MA	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 **DB**	7
11 RPN030MA	4 PR	5 IM	6 **DTH**	7 DTB	1 DH	2 DB	3 DG	7
12 RPN040MA	5 IM	6 DTH	7 **DTB**	1 DH	2 DB	3 DG	4 PR	7
13 RPN013MA	6 DTH	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	7
14 RPN023MA	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7
15 G123	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	7
16 G1	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	7
17 G2	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 DB	7
18 G3	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 DB	3 DG	7
19 SB1	5 IM	6 DTH	7 DTB	1 **DH**	2 DB	3 DG	4 PR	7
20 SB2	6 DTH	7 DTB	1 DH	2 **DB**	3 DG	4 PR	5 IM	7
21 SC	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	6 **DTH**	7
22 SD1	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7 **DTB**	7
23 N21F2	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 **DH**	7
24 N21B10	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 **DB**	7
25 N21K14	4 PR	5 IM	6 **DTH**	7 DTB	1 DH	2 DB	3 DG	7
26 N21P6	5 IM	6 DTH	7 **DTB**	1 DH	2 DB	3 DG	4 PR	7
27 N22K2	6 **DTH**	7 **DTB**	1 **DH**	2 **DB**	3 DG	4 PR	5 IM	7
28 N22B6	7 **DTB**	1 **DH**	2 **DB**	3 DG	4 PR	5 IM	6 **DTH**	7
29 N22F14	1 **DH**	2 **DB**	3 DG	4 PR	5 IM	6 **DTH**	7 **DTB**	7
30 N22P10	2 **DB**	3 DG	4 PR	5 IM	6 **DTH**	7 **DTB**	1 **DH**	7
31 Wflow	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 DB	7
32 Bflow	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 DB	3 DG	7
33 DBflow	5 IM	6 DTH	7 DTB	1 DH	2 DB	3 DG	4 PR	7
34 RPN418EU	6 DTH	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	7
35 RPN425EU	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7
36 RPN432EU	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	7
37 RPN439EU	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	7
38 DPAX	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 DB	7
39 GROUPER	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 DB	3 DG	7
40 PPRIMAIRE	5 IM	6 DTH	7 DTB	1 DH	2 DB	3 DG	4 PR	7
41 TMOY	6 DTH	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	7
42 NGV1	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7
43 NGV2	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	7
44 NGV3	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	7
45 PCNI	3 DG	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 DB	7
46 PCNP	4 PR	5 IM	6 DTH	7 DTB	1 DH	2 DB	3 DG	7
47 NASG	5 IM	6 DTH	7 DTB	1 DH	2 DB	3 DG	4 PR	7
48 TASG	6 DTH	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	7
49 PCONDENS	7 DTB	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7
50 PELEC	1 DH	2 DB	3 DG	4 PR	5 IM	6 DTH	7 DTB	7

Anomalies							
DH	threshold overflow	DG	gradient excess	IM	invalid measurement	DTB	super threshold underflow
DB	threshold underflow	PR	loss of redundancy	DTH	super threshold overflow		

Figure 92: Validation coverage of ADACS-N^{TM} $\times$ 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 instant of the four anomaly scenarios;
3. the occurrence of an anomaly once a given reactor achieves a value which is higher than its first superior threshold;
4. the occurrence of an anomaly once a given reactor achieves a value which is lower than its first inferior threshold;
5. the occurrence of an anomaly once a given reactor achieves a value which is higher than its second superior threshold, and this anomaly accumulates with the anomaly # 3;
6. the occurrence of an anomaly once a given reactor achieves a value which is lower than its second inferior threshold, and this anomaly accumulates with the anomaly # 4;
7. the surfacing of such anomalies on the user interface, in the corresponding failure signal (among all signals present on the UI);
8. the list of signals of each user interface;
9. the user interactions, by accessing the menu options;
10. the transmission of the failure signal 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 anomalous state anymore.

j) Limitations of the Validation

The following improvement points could be added in the ADACS-N^{TM} trace files. They are present in the formal model but are not covered by the validation. For the ADACS-N^{TM} portion implementing the EDF system, this information was not present in the trace files:

1. three anomaly scenarios: *gradient excess* ("dépassement gradient"), *loss of redundancy* ("perte de redondance"), and *invalid measurement* ("invalidité de mesure");
2. 30 reactor parameters (among 50 parameters, 20 are covered by this validation);
3. the display of the reactor parameters on the user interfaces (only the display of the failure signals are covered);
4. the name of the user interface that is displayed once the user accesses a menu option;
5. the calculation of the failure signals before they are displayed on the user interfaces;
6. log of the alarm conditions in the signals analyzed in this case study. In the context of this case study, ADACS-N^{TM} limits its traces to alerts.

5.5 Conclusions of the Connection

This case study analyzed part of the ADACS-N^{TM} implementation of the EDF system described in Chapter 3. This connection of the formal specification and ADACS-N^{TM} is a feasibility study. Few log files are analyzed (38 log files). In order to corroborate the results, more log files would be required. The approach cannot fully answer whether the system implements correctly the specifications or not. The formal specification may contain errors, therefore, may not be a correct representation of the real system. The approach can, thought, give directions to the answer. The redundancy aspect of the approach (i.e., both the LIG laboratory and Atos Worldgrid used the EDF specifications) justify the analysis, and the results can give clues about whether the system implements the specifications correctly or not.

The connection of the LNT formal model to ADACS-N^{TM} permitted the analysis of the intersection "zone" between them (in blue in Figure 93), and it was a fruitful source of improvements on both the formal specification and the ADACS-N^{TM} implementation. After an initial alignment, several correspondences were shown between ADACS-N^{TM} and the formal model. More importantly, one major mismatch was found in the way both ADACS-N^{TM} and the formal model synthesize failure signals once a reactor parameter has an anomaly (c.f Subsection i) on page 125), due to issues in the way this requirement was communicated to Atos Worldgrid and the LIG laboratory.

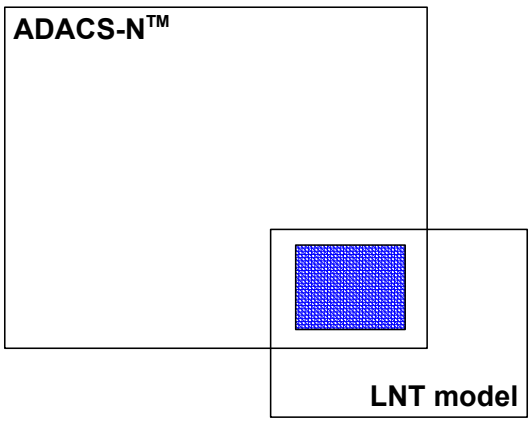


Figure 93: Intersection zone of ADACS-N^{TM} and the LNT formal model which is analyzed

A positive side-effect of the connection is that it allowed the formal specification to be cross checked too. One of the challenges in model-based approaches is to ensure the reliability of the model. Since this type of approaches highly relies on the models, they are expected to be as representative as possible of the real system. Hand-written models have the advantage of being subject to a human analysis and reasoning. Depending on the designer expertise, a good understanding of the system may result on 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. This connection provides a means to mitigate such difficulties: the cross check of the formal specification and the implementation brought realism to the formal model. The application of such techniques aiming at connecting the formal model

to a real system mitigates one reason identified in [Cofer 2012] as one of the causes to few case studies of formal methods to industrial systems: *fidelity* (i.e., there is no guarantee that the models really correspond to the system).

Finally, a Parser was implemented to translate ADACS-N^{TM} logs to LTS sequences, of which the main ideas can be re-used to further connect other formal models of industrial systems. Several perspectives were identified to further improve this integration (c.f Subsection j) on page 127), which can be subject to future work.

5.6 Summary

This chapter details our investigations on the verification of interactive systems independently of plasticity. With this goal, the model checking approach is applied to an industrial case study in the nuclear-plant domain (i.e., the EDF case study). This case study is modeled using the LNT formal specification language and following ARCH, allowing the system to be modeled as a composition of modules, and covering the three aspects we are considering in this thesis: user interface, functional core and users. Several insights emerge from this modeling, such as how to model user interfaces and the functional core, and can be a source for guiding the formal modeling of interactive systems in the future. Nine *usability* and *functional* properties are identified and verified in this case study.

Going further, we present in this chapter our investigations of connecting the formal specification to industrial systems. Three propositions are investigated, namely: *analysis of traces*, *test case generation* and *co-simulation*. We describe and compare the three propositions, and the rationale of a selected proposition to be applied in the Connection Project is given.

In the context of the project, besides the LIG laboratory, an industrial partner called Atos Worldgrid also implemented the EDF system in one of its products, called ADACS-N^{TM}. In order to cross check the ADACS-N^{TM} implementation of the EDF system, an integration of the formal model with ADACS-N^{TM} is also described in this chapter. *Analysis of traces* is used for this integration, and it is further detailed in the chapter. The *model checking* part of our global approach is used to validate part of ADACS-N^{TM}. This connection brings several improvements to both the formal model and to the industrial system. A divergence is highlighted between both the formal model and ADACS-N^{TM}, and fixed in the formal model after a diagnosis analysis. Several improvements are brought from this connection to both the formal model and ADACS-N^{TM}.

Such application of our approach to two industrial case studies (i.e., the EDF case study and the connection to the ADACS-N^{TM} system) indicates that the approach scales well to real-life application.

Verifying interactive systems independently of the adaptation capabilities of their UIs permitted the setting of an initial environment of formal verification for this thesis, allowing us to further investigate how to verify plastic user interfaces.

Chapter 6

Verification of Plastic Interactive Systems

Contents

6.1 Goals **131**
6.2 Improvements in the Formal Model **132**
6.2.1 *Parameters* Module 133
6.2.2 *Signals* Module 134
6.2.3 Overview of the New Formal Model 134
6.3 Needs Raised by Plasticity **136**
6.4 Propositions to Verify Plasticity **137**
6.4.1 Common Part 137
6.4.2 Proposition 1: Verification of the UI Versions 137
6.4.3 Proposition 2: Verification of the Plasticity Engine 139
6.4.4 Proposition 3: UI Comparison 141
6.4.5 Rationale of the Chosen Proposition 142
6.5 On the Comparison of User Interfaces **144**
6.5.1 Interactive System LTS 144
6.5.2 Equivalent User Interfaces 146
6.5.3 Equivalent Modulo "X" User Interfaces 149
6.5.4 Non-Equivalent User Interfaces 150
6.5.5 Inclusion of User Interfaces 150
6.6 Application of the Approach **151**
6.6.1 Equivalent User Interfaces 152
6.6.2 Equivalent *Modulo the Breadcrumb Trail* User Interfaces 153
6.6.3 Non-equivalent User Interfaces and Inclusion 155
6.6.4 Validation in the Case Study 155
6.6.5 Discussion 157
6.7 Summary **159**

6.1 Goals

Chapter 5 described our approach to verifying interactive systems independently of plasticity. This chapter moves a step forward and describes the part of the approach that considers plastic interactive systems. In the context of safety-critical systems, one needs to ensure that some critical functionalities of the UIs are preserved after the UI adaptation. We present and compare three possibilities to verify plasticity using formal methods. The chosen proposition is then

detailed and applied to the case study described in Chapter 3. To experiment these propositions on the case study, its formal specification is subject to several improvements, which are also described in this chapter.

Precisely, we cover two aspects of plastic user interfaces: interaction capabilities and appearance. We provide a means to representing both aspects in one single model called *Interactive System LTS*, which is used afterwards for formal verification.

6.2 Improvements in the Formal Model

One of the contributions of our research team in the Connexion Project is a new prototype implementing the specifications of the case study (cf. Section 3.4 on page 76). We use the LIG prototype as the basis for introducing plasticity in our verification proposition. An example of the main UI of the LIG prototype is illustrated in Figure 94, in French, with the main UI zones identified by numbers: (1) plant status; (2) failure signals; (3) reactor parameters; and (4) menu.

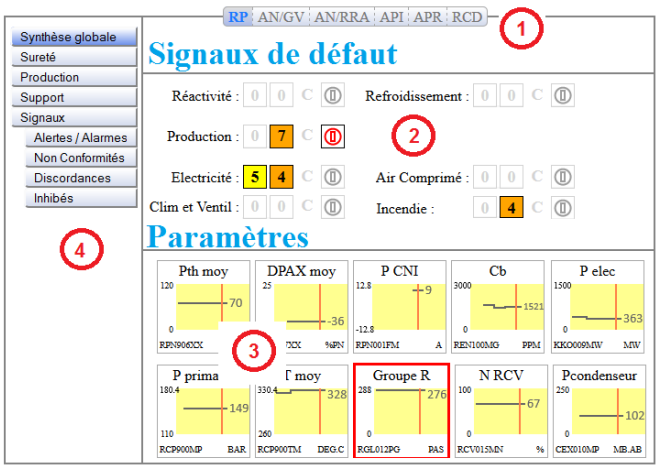


Figure 94: The LIG prototype of the control room system running on a PC

In order to introduce plasticity, the formal specification needs several modifications. Special attention is paid to the modeling of the UIs: to be able to reason over the UI adaptation, the basis of UI modeling have to be modified. Two modules are included in our UI component of the formal model (Figure 95): *parameters* and *signals* modules. The former models the UI zone that displays the reactor parameters (zone n.3 in Figure 94) and the latter models the UI zone that displays the failure signals (zone n.2 in Figure 94). With this increment, the four zones of the UIs are now described in the formal model, in contrast to the previous version of the formal model (Figure 70) which modeled only both UI zones the user can interact with (i.e., zones n.1 and n.4).

Besides, communications between modules are now fully separated according to ARCH (Figure 95): the dialog controller communicates with both the UI and the functional core modules, and the user module communicates only with the UI modules.

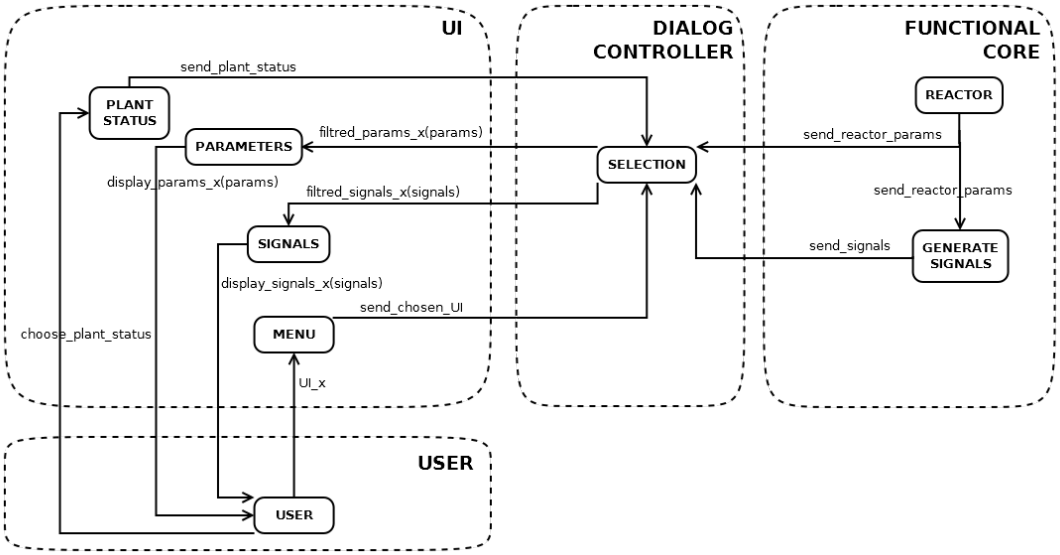


Figure 95: New formal model structure

The state space of this formal model is larger than its previous version (i.e., the version depicted in Figure 70), and the suppression of optimization bridges that avoid the model to fully follow ARCH (cf. Section 5.2.1 on page 96) is compensated by other alternatives, such as the simplification of the functional core modules (i.e., decreasing from 29 down to 25 the number or reactor parameters which are modeled) and the inclusion of a *scheduler* in the model. A *scheduler* is a special module exclusively responsible for orchestrating the *rendez-vous* between the modules in the formal model. The exchange of data in an LNT formal model takes place by *rendez-vous* through *gates* between two or more modules. For instance, the menu option chosen by a user is sent to the menu module through a gate when a *rendez-vous* between both modules takes place (Figure 95). The *rendez-vous* orchestration provided by the *scheduler* permitted the model to evolve into this new version in which new modules are included, increasing the state space of the formal model.

The UI modules of the formal model now include some treatment of the data. As for the previous version of the model, the dialog control (i.e., the *selection* module) still receives the reactor parameters and signals from the functional core, and filters the ones pertaining to the current UI. But, instead of sending them directly to the *user* module, it sends them to the *parameter* and to the *signals* modules, which treats the data before sending them to the *user* module. Such treatments are described in the next subsections.

6.2.1 *Parameters* Module

The *parameters* module transforms raw data received from the dialog control into curves and symbols, and send them to the *user* module. Figure 96 provides an excerpt of the LNT function that performs this transformation: according to the parameter anomaly, a given symbol is chosen (lines 15-24). For instance, if the parameter has a *threshold overflow* ("dépassement

haut") anomaly, the *top yellow arrow* ("fleche jaune haut") symbol is chosen (cf. Table 3). Each parameter is transformed into a format consisting of its name and corresponding symbol (lines 4-10), and this transformed data is sent to the user (line 11).

```
1  function Transformer_Param(p:TParam):TParam_Courbe is
2    (...)
3        symb := Transformer_Defaut(p.defaut);
4      case p in
5          TPth_moy      (valeur_n, defaut) -> nomParam := Pth_moy
6        | TDPAXmoy      (valeur_i, defaut) -> nomParam := DPAXmoy
7        (...)
8      end case;
9
10     pc := TParam_Courbe(nomParam, symb);
11     return pc
12 end function
13
14
15 function Transformer_Defaut(def:TDefaut): TSymbole is
16   case def in
17         nil                    -> return rien
18       | depassementHaut        -> return flechejaunehaut
19       | depassementBas         -> return flechejaunebas
20       | propagationInvalidite -> return cadremagenta
21       | depassementGradient    -> return cadrerouge
22       | redondanceMesure       -> return barreorange
23   end case
24 end function
```

Figure 96: The *parameters* module - an excerpt of LNT code

6.2.2 *Signals* Module

A similar transformation is done within the *signals* module. In this case study, seven signals are associated to each reactor function (cf. Figure 52), the following four of which are covered in our modeling: *alerts*, *alarm conditions*, *nonconformity*, and *equipment state change*. Initially, the four signals present no failure (lines 3-6 in Figure 97). Depending on the raw value received from the dialog controller module, a given symbol is assigned to each kind of failure (lines 7-18). For instance, the *yellow square* ("carre jaune") symbol corresponds to the *alert* failure (lines 7-9). It represents the fact that, in the real system, once a reactor function has an alert, a yellow square is displayed on the UI under this reactor function (e.g., Figure 50). The transformed signals are sent to the user (line 21).

6.2.3 Overview of the New Formal Model

Table 11 summarizes the new version of the formal model. All modules are affected to some extent (except `reactor.tnt` and `reactor.fnt`). Specially, the *parameters* (line 3) and *signals* modules (line 4) are added. The other modules of the formal model also passed through slight modifications, but their essence is roughly the same as in Section 5.2 (page 96), i.e., the previous version of the model. The new formal model has 2462 lines of code.

Table 12 shows the size of the LTS generated from this LNT formal model, using CADP. This LTS is bigger than the previous version of the formal model (i.e., 26 167 456 states and 185

```
function Transformer_Signal(s:TSignal):TSignal_Courbe is
    (...)
    symb1 := signal_vide;
    symb2 := signal_vide;
    symb3 := signal_vide;
    symb4 := signal_vide;
    if Nat(s.nbAlertes) > 0 then
      symb1 := carrejaune
    end if;
    if Nat(s.nbAlarmes) > 0 then
      symb2 := carreorange
    end if;
    if s.etatConforme == false then
      symb3 := ncrouge
    end if;
    if s.etatStable == false then
      symb4 := carrerouge
    end if;

    sc := TSignal_Courbe(s.nom, symb1, symb2, symb3, symb4);
    return sc
end function
```

Figure 97: The *signals module* - an excerpt of LNT code

Table 11: Summary of the new version of the formal model, to include plastic UIs

#	ARCH component	File	Description	# loc
1	user interface	plant status	**1** plant status is always selected	32
2	user interface	menu	selection among **8** menu options	72
3	user interface	parameters	transformation de **25** reactor parameters	117
4	user interface	signals	transformation de **13** reactor signals	73
5	functional core	reactor	modeling of **25** reactor parameters	305
6	functional core	generate signals	modeling of **13** reactor signals	222
7	functional core	scenarios	describes the **five** anomaly scenarios of reactor parameters	328
8	dialog controller	selection	filter of **25** parameters and **13** signals	74
9	(user)	user	selection of the plant status and monitoring of reactor parameters	170
10	(auxiliary file)	main	entry point of execution	108
11	(auxiliary file)	library	type definitions	652
12	(auxiliary file)	library.tnt	internal optimizations	280
13	(auxiliary file)	reactor.tnt	internal optimizations	3
14	(auxiliary file)	reactor.fnt	internal optimizations	26
			TOTAL	**2462**

772 171 transitions, Table 6) mainly because new modules were added in the formal model (i.e., the *parameters* and the *signals* module), which increases the state space of the formal model.

Table 12: Size of the LTS of the new version of the formal model, to include plastic UIs

# states	# transitions
33 053 947	189 539 691

This new version of the formal model isolates the UI aspects from the other concerns, paving the way for our comparison approach using formal methods.

6.3 Needs Raised by Plasticity

In Section 3.4, page 76, we presented several versions of the EDF system, developed by our research laboratory. The UIs of the case study change according to five contexts of use: running on PC, running on a Smartphone, running on Tablet, running in Expert mode and running in Training mode. When analyzing different versions of a UI, we are particularly interested in two aspects: their interaction capabilities and appearance.

UI *interaction capabilities* concern the ways users can interact with the UI (and, conversely, how the UI reacts to this interaction). For instance, in Figure 94, the user can interact with the UI in the following ways: by selecting the plant status and by accessing different views using the menu options. Conversely, when the user performs each of those actions, the UI responds in some way (e.g., by displaying the reactor parameters according to the selected plant state). When comparing two UIs, we want to know whether users can perform the same actions in both of them, and whether the UIs reacts in the same way or not. In this point of view, we are interested in *what* the user can do (e.g., "select menu option 1") , and in *what* the UI does in reaction (e.g., "display main UI"). It does not concern neither *how* such interaction capabilities are provided (e.g., which widget is used to display the menu) nor *how* the UI displays the outputs. This relates to the UI appearance.

UI *appearance* concerns the elements present on the uI (where they are presented, in which color, etc.). For instance, we may want to know which symbol represents the absence of unexpected events in the reactor.

Plasticity exposes users to different user interfaces that can diverge from each other at several levels, which may cause loss of information and/or consistency. This raises the question of similarity between UIs. One may be interested in knowing whether all UI aspects (e.g., the UI interaction capabilities and appearance) are preserved once the UI is adapted, and if it is not the case, which aspects are discarded/added. In addition, one may also be interested to know whether they are similar in terms of appearance and interaction capabilities, or verify common properties in both of them. Besides, in the context of safety-critical systems, we also need to ensure that some critical functionalities of the UIs are preserved after the UI adaptation. For instance, all versions must always display the reactor parameters whatever their form.

6.4 Propositions to Verify Plasticity

It would be paralyzing to search for a single optimal way to add plasticity in the model. Several choices are possible, and it is not obvious which one is the most appropriate. We compare now three different approaches [Oliveira *et al.* 2015b], in order to give some rationale of the chosen approach, and investigate this one deeper. Two of the propositions consist in verifying properties on the formal models, either by directly checking the UI versions or by checking the *plasticity engine*[1] used to generate the UI versions. The third proposition is based on the comparison of UI versions.

6.4.1 Common Part

Figure 98 illustrates the common part of all propositions. The entry point of all propositions is the interactive system to be verified, from which a formal model is written describing some aspects of the user behavior, the system functional core and the user interfaces. The formal specification allows the usage of several formal verification techniques. For instance, model checking can be used to verify that all UI versions display reactor signals, case in which the second input of the formal verification would be a set of properties. Alternatively, equivalence checking can be used to compare two versions of a UI to check consistency, for instance, in the display of signals (either both of them display a signal or none of them does), case in which the second input of the formal verification would be the formal model of another UI.

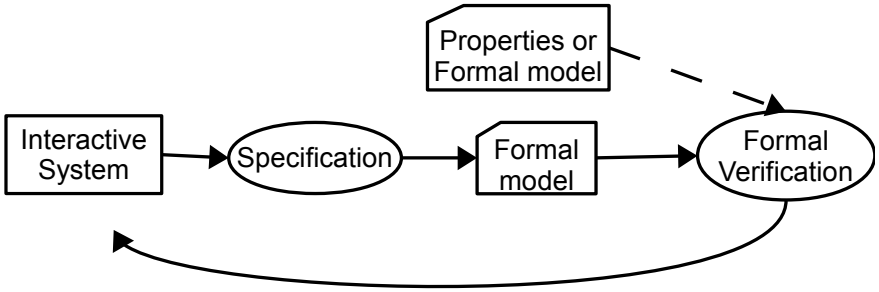


Figure 98: Common part to all propositions

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

6.4.2 Proposition 1: Verification of the UI Versions

Considering ARCH, the first proposition contains modules for the functional core, the dialog controller and the UIs. In our example, the UI module contains the four zones of the UI, i.e., the plant status, the menu, the failure signals, and the parameters. In this proposition, the whole interactive system is represented by one single formal model, including all plastic user interfaces (Figure 99). For instance, if the UIs are intended to adapt according to two different devices, both versions of the user interface are described in the formal model, and the transformation rules needed to adapt the user interfaces are also included. In our example of nuclear power

[1] cf. Section 1.3 on page 3 for the definition of *plasticity engine*

plant control system, consider two UI versions: one for the expert mode and one for training mode. They are both described in the UI modules. The goal of this proposition is to directly verify the different UI versions.

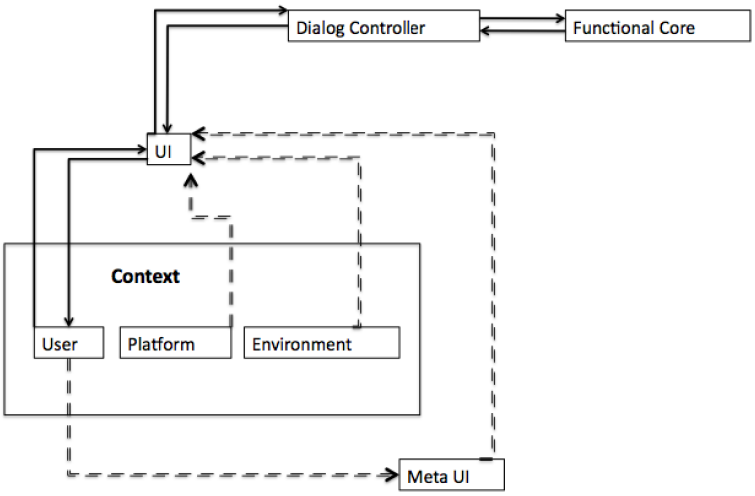


Figure 99: Modeling UI versions

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

The user can also interact with a meta UI, which permits the UI mode (i.e., expert or training) to be chosen and some UI elements to be distributed over different 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. In the context of plasticity, such repartition of UIs over devices is called UI *redistribution* [Vanderdonckt *et al.* 2008]. A communication between the meta UI and UI modules permits the UI versions to be modified 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 this information, the UI modules can choose the appropriate representation of the UI. This means that the UI modules contain the UI adaptation logic (i.e., the transformation rules).

In the formal model, the transformation rules represent all the adaptation effects in the corresponding UI zones. For example, on the uI parameter (resp. signal) zone, there are two cases: the display of only the problematic parameter (resp. signal) on a smartphone and the display of all parameters (resp. signals) on large screens (Figure 100).

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

```
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 100: Excerpt of LNT code describing the UI modules

represented in the LTS.

Once the formal model is created, model checking can be used to verify properties over the model (Figure 61). 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 MCL [Mateescu & Thivolle 2008], this property is formalized as follows:

$$[true^*]\langle true^* \ . \ 'DISPLAY_.^*_SIGNAL.^*'\rangle true \tag{6.1}$$

In this MCL formula, $DISPLAY_.^*_SIGNAL.^*$ is a regular expression that matches in the LTS transitions labeled with either gate "DISPLAY_FAILURE_SIGNAL" or gate "DISPLAY_ALL_SIGNAL" and arbitrary offers.

To summarize, in this first proposition the UI versions are modeled in the UI formal modules, which contains the description of the UIs, their behavior, and the adaptation logic. The adaptation logic is embedded into the description of the UIs. Model checking permits to perform verification of properties over all the UI versions.

6.4.3 Proposition 2: Verification of the Plasticity Engine

In the second proposition, we consider that a plasticity engine is formally described in the model (Figure 101). The goal is to explicitly verify the adaption logic of the engine.

In this proposition, the formal model is also created following ARCH. 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., the excerpt of LNT code in Figure 102). The UI, the context of use, the meta UI, and the functional core

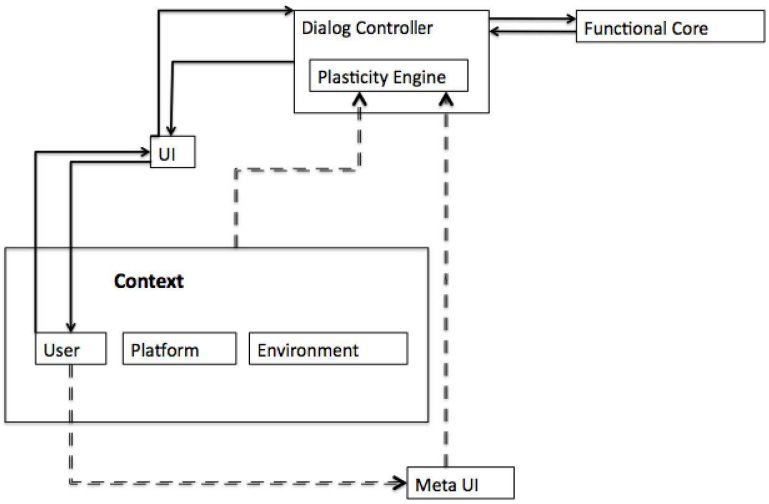


Figure 101: Modeling a plasticity engine

are specified to provide the information the plasticity engine needs to transform the UIs (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 adapts the UI correctly: the engine must apply all transformations on the UIs, leaving to the UI modules only the tasks of displaying the result UIs 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 102: Excerpt of LNT code describing the 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, the expert-mode UI (Figure 56b) displays all reactor signals in the *Failure Signal* zone (i.e., failure and non-failure signals), while the smartphone UI displays in the same zone only the failure signals (Figure 55a). 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, this is expressed

as:

$$[true^* \ . \ \prime FAILURE_SIGNAL_\backslash(.*\backslash)\prime] \\ \langle true^* \ . \ \prime DISPLAY_FAILURE_SIGNAL_\backslash 1\prime\rangle \ true \tag{6.2}$$

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 $\backslash n$ (here, n is a digit) matches the same string of characters that is matched by the n-th expression enclosed between "\(" and "\)" earlier, counting from the left. Intuitively, the first regular expression matches a failure in a given reactor signal (i.e., "FAILURE_SIGNAL_*signalname*") and the reactor signal name are matched in the second regular expression (i.e., "DISPLAY_FAILURE_SIGNAL_*signalname*"), expressing the requirement that once a failure signal occurs, this signal is always displayed on the UI.

Going further, we propose to verify the plasticity engine itself by verifying the coverage of the transformation rules. The transformation rules are expected to cope with changes in the context of use and correspondingly adapt the UI. In this case, we could verify whether the engine takes into account all expected changes in the context of use or not. To avoid the explosion that could result from the combination of contexts of use, we propose to limit the verification on the contexts of use that are considered. In our example, consider two changes in the context of use: the platform (e.g., PC or smartphone) and the user (i.e., expert or training). Concerning 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:

$$\langle true^* \ . \ \prime DISPLAY_SMARTPHONE_UI.*\prime\rangle \ true \tag{6.3}$$

To summarize, the second proposition 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.

6.4.4 Proposition 3: UI Comparison

In this last proposition, 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. The main motivation is to verify to which extent plastic UIs are similar. The UIs for each context of use are represented by their own formal model. Before creating the formal models, designers must have the rendering of the UI versions adapted to each context of use, allowing the specification of a formal model for each one of them to be done. This proposition requires as many formal models as the number of contexts of use to cope with, including all the specificities of each context of use (platform, user, environment). In our example, the combination of both platforms and both user expertises can give rise to four formal models. Then, the formal models are compared to each other to identify their degree of similarity.

In this proposition, the formal models also follow ARCH. No module for the context of use nor for the meta UI is included. All specificities brought from the context of use are included

in the corresponding UI formal model. The UI formal models reflect their context of use by describing the UI generated for such context. For example, in our case study, the UI module contains the four zones of the UI, i.e., the plant status, the menu, the failure signals, and the parameters. Figure 103 illustrates how the parameter module of each UI version is formalized, i.e., the display of only the problematic parameter on a smartphone and the display of all parameters on large screens.

```
PARAMETERS Smartphone.LNT
  display_failure_param();

PARAMETERS PC.LNT
  display_all_params();
```

Figure 103: Excerpt of LNT code of each UI version

These formal models are then compared, pairwise, using equivalence checking (Figure 62), allowing one to measure to which extent the user interfaces are the same. In case they are not equivalent, the UI divergences are highlighted, and the possibility of leaving these divergences out of the analysis is provided, re-interacting the equivalence verification.

In our example, the Expert mode and the Smartphone 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 can be made to reason about the interaction capabilities and not their appearance. After these abstractions, both UI versions are shown equivalent. Meaning that, at a certain level of abstraction, both UIs provide users with the same interaction capabilities.

To summarize, the comparison proposition enables the verification of UI versions without worrying how they are obtained. It requires to completely describe each UI version. While considering the multiplicity of contexts 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: it permits to ensure, for instance, that critical features are maintained in all UI versions.

6.4.5 Rationale of the Chosen Proposition

Table 13 summarizes the three propositions to verify interactive systems with plastic user interfaces. In the first proposition, one single formal model represents all the UI versions, the adaptation logic is embedded into the description of the UIs, and model checking permits properties to be verified over all the UI versions. In the second proposition, one single formal model includes an explicit representation of the plasticity engine, and the verification of the engine is done by model checking. Finally, in the third proposition, there are n formal models, one for each context of use. There is no representation of the plasticity engine, and the verification is done using equivalence checking to compare the formal models.

Each proposition has its strengths and drawbacks. To compare them, we focus on the following criteria: the number and complexity of the UI versions to be modeled, and the number and complexity of adaptation rules.

In the first proposition, 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

Table 13: Summary of propositions to verify plastic user interfaces

Approach	**# Formal models**	**Verif.technique**
Verification of the UI versions	one, covering all UI versions	model checking
Verification of the plast. engine	one, plast.engine formalized	model checking
UI comparison	n, one for each context	equiv. checking

below a certain threshold and the number of adaptation rules for the UI is low, it can be a good and simple proposition. Once the adaptation representation becomes too onerous in the formal model, prejudicing its readability, it is recommended to separate it into another module (i.e., the plasticity engine module). A deeper investigation in order to define this threshold to guide the choice of either proposition could be a further work of this thesis. In terms of verification to be performed, this proposition permits the same properties to be verified 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*.

The novelty of the second proposition is the representation and verification of a plasticity engine. One can imagine to automatically generate the code of 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. They tend to grow in number to follow the system evolution, i.e., once a new context of use is covered by the system, new transformation rules are needed. Contexts of use can even be discovered at runtime [Ganneau *et al.* 2008], which makes the identification of transformation rules at design time harder. This makes the formalization of such transformation rules difficult. Moreover, this proposition relies on the existence of a plasticity engine, which is not always the case.

Both the first and the second propositions rely on two things: (1) the modeling of adaptation rules and (2) the verification of properties by model checking. Although several advances have been made in plasticity of user interfaces (e.g., [Sottet *et al.* 2006, Demeure *et al.* 2006, Sottet *et al.* 2007, Coutaz *et al.* 2007, Vanderdonckt *et al.* 2008, Ganneau *et al.* 2008, Demeure *et al.* 2008, Serna *et al.* 2010]), research on the plasticity engine and its adaptation rules are not yet in a mature state to move to the next step, their modeling and verification, which makes the usage of both propositions difficult. Concerning the verification of properties by model checking, we used this approach for the verification of interactive systems independently of the UI adaptation (cf. Chapter 5), and have shown in this section the feasibility to plastic UIs too (modulo the difficulties to model the plasticity engine). To propose an alternative to the use of model checking, we conducted deeper investigations on the use of equivalence checking in the context of plastic UIs, which have not been often applied in the HCI context (cf. Tables 1 and 2 in Chapter 2).

Comparison of formal models bypasses the difficulties of modeling the adaptation logic. Even though it requires to create one model for each UI version and to compare UI versions two by two, which can be time-consuming if the number of UI versions is significant, the effort is alleviated because there is no need to represent the plasticity engine in the formal model. It is based on the existing versions of the UI: the formal models focus on representing the UI versions, without knowing how they have been produced. Finally, with this proposition, designers do not

verify some properties, rather than that, they check the consistency between two UI versions thanks to equivalence checking. This proposition is detailed in the following subsections. The other two propositions are not further investigated in the context of this thesis.

6.5 On the Comparison of User Interfaces

To compare different versions of a UI, we apply the equivalence checking part of our global approach (cf. Section 4.3 on page 85). We present in this sections the formal framework proposed in [Oliveira *et al.* 2015a], with some refinements on the formal definitions.

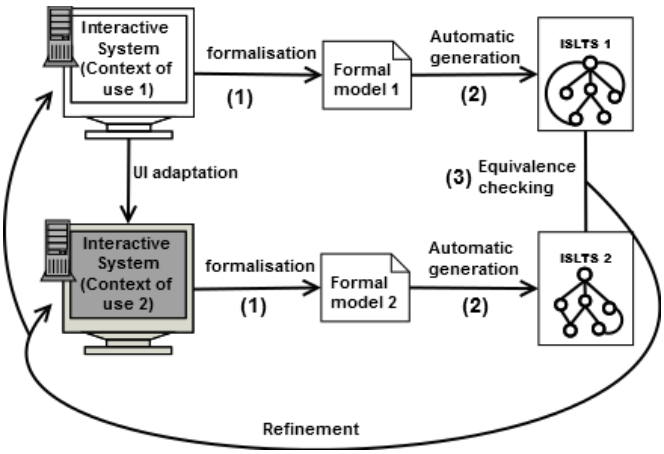


Figure 104: Comparison of user interfaces

Figure 104 illustrates a refinement of the equivalence checking part of our global approach. Here, the LTS representations of the formal model are added. The first step (1) consists in creating a formal model of the UIs. The formal model should reflect as much as possible the real system. The formal model is used to automatically generate (2) an ISLTS (*Interactive System LTS*). We derived ISLTS from LTS (*Labeled Transition System*), as will be detailed in the sequel.

The verification of ISLTS equivalence (3) is performed thanks to *equivalence checking* (Figure 105). First, we verify whether both ISLTS are equivalent or not. If they are, considering interaction capabilities and appearance, both UIs are equivalent. If both ISLTS are not equivalent, we verify whether one *includes* the other. If one ISLTS includes another one, one UI contains at least all interaction capabilities (and corresponding appearance) of the other one. The third possibility is that both UI models are neither equivalent nor included one in the other.

This analysis is supported by three abstraction techniques, which are explained in the following. The results of the comparison allow the formal models and/or the real UIs to be refined.

6.5.1 Interactive System LTS

We enhance LTS (cf. Section 4.5.3 on page 91) to model user interfaces.

Definition 1 (ISLTS)**.** *An* ISLTS *(Interactive System LTS) is a 6-tuple* $\langle Q, C, L, A, T, q_0 \rangle$ *where:*

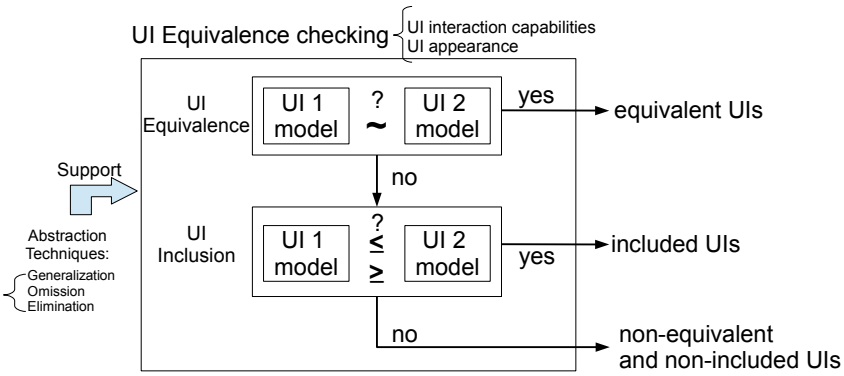


Figure 105: Equivalence checking of user interfaces

- Q is a set of states in which interactive system can be;
- C is a set of UI components;
- L is a set of action names;
- A is a set of actions. They model the system dynamics: actions users can perform on the UI and the UI response to these actions. Each action $a \in A$ has the form $l(c_1, ..., c_m)$ where $l \in L$, $m \geq 0$, and $\forall i \in [1..m], c_i \in C$. Intuitively, actions can carry a list of UI components, representing the UI appearance after the action is performed. For a given action $a \in A$, when $m = 0$ (i.e., the action does not carry any UI component), the parentheses are omitted and the action has the form l, where $l \in L$;
- $T \subseteq Q \times A \times Q$ is a transition relation that changes the interactive system state once an action $a \in A$ is performed. We also use the notation $q \xrightarrow{a} q'$ for $(q, a, q') \in T$;
- $q_0 \in Q$ is the initial state of the interactive system.

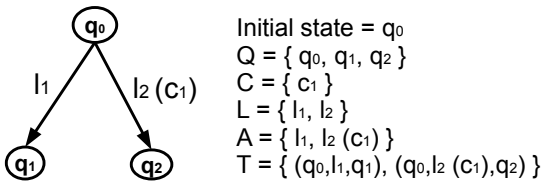


Figure 106: An ISLTS (*Interactive System LTS*)

Figure 106 illustrates an ISLTS with the sets Q, C, L, A, T and the element q_0. An ISLTS provides a means to representing both UI interaction capabilities and appearance in one single model: they are represented in the set A of actions. Each action names an interaction capability, and ISLTS enriches LTS actions with data (the set C of components) to represent the UI appearance.

According to the domain, the set C is composed by subsets detailing the components of the UI. Figure 107 illustrates an example of an action representing the display of reactor parameters with their current value and status. Concerning the UI appearance, the set C represents how the reactor parameters are displayed on the UI, which in this example is with their values and status. Consider a subset $P = \{Pth_moy, GroupeR, ...\}$ of reactor parameter names and a subset $S = \{normal, fail\}$ of reactor parameter status. In the example of Figure 107, $c \in C$ has the form $p(v, s)$, where $p \in P$, $v \in \mathbb{R}$ and $s \in S$, e.g., $C \ni \{Pth_moy(70, normal), GroupeR(276, fail)\}$. The set C is domain-dependent and can have other formats, not changing the way it is integrated in ISLTS actions, i.e., each action $a \in A$ has the form $l(c_1, ..., c_m)$ where $l \in L$, $m \geq 0$, and $\forall i \in [1..m], c_i \in C$. In the example of Figure 107, $L \ni ShowParams$ and $A \ni ShowParams(Pth_moy(70, normal), GroupeR(276, fail))$.

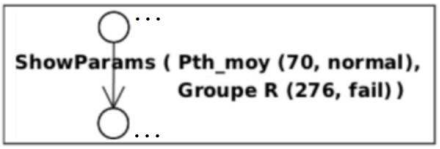


Figure 107: UI appearance in an ISLTS

6.5.2 Equivalent User Interfaces

Once the UIs are modeled as ISLTS, one can perform UI comparison thanks to *equivalence checking*. We introduce now several definitions we derived from formal techniques to apply to HCI. We combine the notion of equivalence with several abstract techniques.

Definition 2 (**Equivalent user interfaces**). *Given two* ISLTS *M and H, if an equivalence relation R exists between the states of M and H, then M and H are said* R equivalent *(written $M \sim_R H$).*

There are several equivalence relations available in the literature, such as *strong bisimulation* [Park 1981], *branching bisimulation* [van Glabbeek & Weijland 1996], *safety equivalence* [Bouajjani *et al.* 1991], etc. Which relation to choose depends on the level of details of the model and the verification goals. We use *strong* and *branching* bisimulation relations, due to the strong implications provided by the former and to the flexibility provided by the latter.

Strong bisimulation [Park 1981] is the most restrictive relation. It relates two standard LTSs in the following way: Two LTSs M and H are strongly bisimilar if there exists a relation $R \subseteq Q_M \times Q_H$ (called *strong bisimulation*) such that:

1. The initial states of M and H are related by R;
2. If $R(m, h)$ and $m \xrightarrow{a} m'$, then there exists a state h' such that $h \xrightarrow{a} h'$ and $R(m', h')$;
3. Conversely, if $R(m, h)$ and $h \xrightarrow{a} h'$, then there exists a state m' such that $m \xrightarrow{a} m'$ and $R(m', h')$.

This formal definition concerns the LTS states and the actions that trigger state transitions. Such concern for LTS actions suits our UI interaction and appearance modeling in the ISLTS

actions. Since in an ISLTS every action $a \in A$ has the form $l(c_1, ..., c_m)$ where $l \in L$, $m \geq 0$, and $\forall i \in [1..m], c_i \in C$, when comparing ISLTS actions both UI interaction capabilities and UI appearance are taken into account.

Strong bisimulation is intuitively illustrated in Figure 108. Two systems (each one represented by an ISLTS) are strongly equivalent whenever they can perform the same actions (possibly enriched with UI components) to reach strongly bisimilar states, i.e., they agree on each step they take.

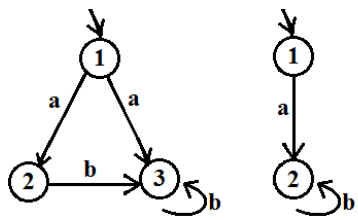


Figure 108: Two strongly equivalent ISLTS

Diversely, there are cases in which certain actions (together with the UI appearance after the action execution) may be skipped in the analysis. These actions receive a special label τ in the LTS, and several equivalence relations exist that deal with τ actions in a special way. $\tau \in A$ represents an action that is considered irrelevant in the context of the analysis, and can be ignored, even though it is still present in the UI model. We call this abstraction technique an *omission*:

Definition 3 (Omission). *Given an* ISLTS $U = \langle Q, C, L, A, T, q_0 \rangle$, *and a set* $O \subseteq A$, Omit(O,U) $= \langle Q, C, L, A \backslash O, T', q_0 \rangle$, *where* $T' = \{(q, a, q') \mid (q, a, q') \in T$ *and* $a \notin O\} \cup \{(q, \tau, q') \mid (q, a, q') \in T$ *and* $a \in O\}$.

Tagging some actions $a \in A$ with the τ label allows weaker equivalence relations to bypass such actions when checking equivalence between models. Since in an ISLTS every action $a \in A$ has the form $l(c_1, ..., c_m)$ where $l \in L$, $m \geq 0$, and $\forall i \in [1..m], c_i \in C$, once an action a is ignored, a UI interaction capability is intentionally disregarded, together with the UI appearance that results from such action (possibly modeled in the body of the a action).

This abstraction is useful, for instance, when users are provided with a functionality activated in different ways in two UIs. For example, two user interfaces U_1 and U_2 that have menus with the same options, as illustrated by the sets A_1 and A_2 of actions below. The menu is always visible in U_1 and hidden in U_2, as illustrated by the absence (resp. presence) of the "open_menu" action in the set A_1 (resp. A_2) of actions:

$$A_1 = \{choose_menu_option1, choose_menu_option2\}$$

$$A_2 = \{open_menu, choose_menu_option1, \\ choose_menu_option2\}$$

$$O = \{open_menu\}$$

Once one displays the menu in U_2, the menu behaves exactly like in U_1. By including "open_menu" in the set O of omitted actions, *omission* permits the action of menu activation to

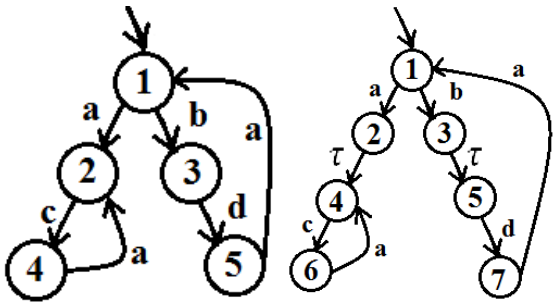
be ignored when comparing the user interfaces, even though it is still present in the U_2 model.

When omitted actions (τ) are present in the model, weaker equivalence relations are more appropriate. *Branching bisimulation* [van Glabbeek & Weijland 1996] is one of the most commonly used. It considers sequences of τ-actions. We write $m \Rightarrow m'$ for a path from m to m' having an arbitrary number (≥ 0) of τ-actions. Branching bisimulation relates two standard LTSs in the following way: Two LTSs M and H are branching bisimilar if there exists a relation $R \subseteq Q_M \times Q_H$ (called *branching bisimulation*) between the states of M and H such that:

1. The initial states of M and H are related by R;
2. If $R(m, h)$ and $m \xrightarrow{a} m'$, then either $a = \tau$ and $R(m', h)$, or there exists a path $h \Rightarrow h' \xrightarrow{a} h''$ such that $R(m, h')$ and $R(m', h'')$;
3. Conversely, if $R(m, h)$ and $h \xrightarrow{a} h'$, then either $a = \tau$ and $R(m, h')$, or there exists a path $m \Rightarrow m' \xrightarrow{a} m''$ such that $R(m', h)$ and $R(m'', h')$.

Similarly to strong bisimulation, branching bisimulation also concerns LTS states and actions. Thus, it considers the UI interaction capabilities and appearance as the previous equivalence relation: given the form of an action in an ISLTS (i.e., $l(c_0, ..., c_m)$ where $l \in L$, $m \geq 0$ and $\forall i \in [1..m]\ c_i \in C$), actions are taken into account with the components present on the UI after the execution of the action.

The essence of branching bisimulation is illustrated in Figure 109. Intuitively, both ISLTS depicted in this figure are branching equivalent because for each state, the same actions (preceded by zero or more τ actions) can be triggered.



(a) Without τ-steps (b) With τ-steps

Figure 109: Two branching equivalent ISLTS

Regardless the chosen equivalence relation, our approach permits to reason over UI models at different levels of details. As an illustration, consider a UI fragment of our case study (Figure 110). This UI fragment displays the number of *alert* signals in the *Production* reactor function. The UI on the left represents the absence of alerts by a `0` beside *Production*. By contrast, on the UI on the right, nothing is displayed in the same zone. In a more abstract version of these UIs, this information is represented by `default_empty_symbol`. We call this kind of abstraction a *generalization*, and it concerns the UI appearance and interaction capabilities.

Definition 4 (Generalization). *Given an* ISLTS $U = \langle Q, C, L, A, T, q_0 \rangle$*; three sets* C' *of UI components,* L' *of action names, and* A' *of actions such that every action* $a' \in A'$ *has the form*

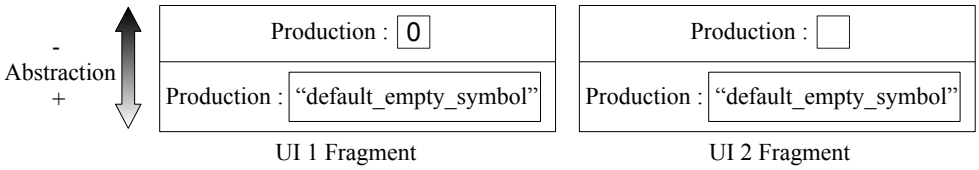


Figure 110: *Generalization* abstraction technique

$l'(c'_1, ..., c'_m)$ *where* $l' \in L'$, $m \geq 0$ *and* $\forall i \in [1..m]$, $c'_i \in C'$*; and a total function* $G \subseteq A \rightarrow A'$*, such that* $\tau \in A'$ *and* $G(\tau) = \tau$*, the* generalization *of* U *with respect to* G*, written* $Gen(G, U)$*, is the* ISLTS $\langle Q, C', L', A', T', q_0 \rangle$ *such that* $T' = \{(q, a', q') \mid (q, a, q') \in T\}$.

Generalization allows elements describing the UI appearance to be represented in a more abstract way in the formal model. Although this abstraction technique concerns mainly the UI appearance, it can also be used to represent UI interaction capabilities in a more abstract way, since the generalization is performed at the level of the ISLTS actions A. Given the form of an action in an ISLTS (i.e., $l(c_0, ..., c_m)$ where $l \in L$, $m \geq 0$ and $\forall i \in [1..m]$ $c_i \in C$), generalization covers the UI interaction capabilities together with the components present on the UI after a given interaction.

An example of generalization is illustrated below:

$$A = \{ShowSignals(zero)\}$$

$$A' = \{ShowSignals(default_empty_symbol)\}$$

$$G = \{ShowSignals(zero) \ \mapsto \ ShowSignals(default_empty_symbol)\}$$

In the original UI model, the absence of alert signals in the reactor function is represented by the number `zero`. While in a more abstract UI model, the absence of alert signals in the reactor function is generalized to `default_empty_symbol`.

The use of regular expressions enables sophisticated transformations on the UI appearance representation. Consider $A \ni \{ShowParams(Pth_moy(70, normal), GroupeR(276, fail))\}$ the example of the set A of actions described in Definition 1. This action represents the display (on the UI) of reactor parameters with their current value and anomaly condition. Instead of displaying all reactor parameters, this action could be generalized in a more abstract visualization, where only failed parameters are displayed. Regular expressions permit the transformation from $A = \{ShowParams(\ \underline{Pth_moy(70, normal),\ GroupeR(276, fail)}\ )\}$ into $A' \ni \{ShowParams(\ \underline{Failure\ in\ GroupeR}\ )\}$.

6.5.3 Equivalent Modulo "X" User Interfaces

There are cases in which certain divergences between two user interfaces are considered acceptable. For instance, when a navigation aid is present in one UI and absent in another one. Knowing that the UIs present this difference, one may still want to analyze the remaining aspects of the UIs. Equivalence modulo "X" permits this reasoning. We introduce another abstract technique that permits UI elements to be eliminated in the model before performing the analysis:

Definition 5 (Elimination). *Given an* ISLTS $U = \langle Q, C, L, A, T, q_0 \rangle$*, and a set* $X \subseteq A$*,* Eliminate(X,U) $= \langle Q, C, L, A \backslash X, T', q_0 \rangle$*, where* $T' = \{(q, a, q') \mid (q, a, q') \in T$ *and* $a \notin X\}$*.*

Each action $x \in X$ also has the form $l(c_1, ..., c_m)$ where $l \in L$, $m \geq 0$, and $\forall i \in [1..m], c_i \in C$, so both UI interaction capabilities and appearance are taken into account. Intuitively, the set $X \subset A$ is a set of actions (enriched or not by the UI appearance) eliminated in U before the comparison analysis. In this case, some UI aspects are left out of the analysis. Contrary to *omission*, in which the elements are still present in the model and are just ignored.

In the example below, extracted from the case study, a breadcrumb trail is present in U_1 and not in U_2. To verify if both UIs are equivalent disregarding this divergence, we eliminate the representation of this functionality in the U_1 model:

$$A_1 = \{select_breadcrumb_trail, choose_plant_state, \\ show_params\};$$

$$A_2 = \{choose_plant_state, show_params\}$$

$$X = \{select_breadcrumb_trail\}$$

$$A_1 \setminus X = \{choose_plant_state, show_params\}$$

In this case, both user interfaces are equivalent *modulo* the elements of X:

Definition 6 (Equivalent modulo "X" user interfaces). *Given two* ISLTS*,* M *and* H*, and a set* $X \subset (A_M \cup A_H)$*, we say that* M *and* H *are equivalent modulo "X" for a certain relation* R *if* Eliminate(X,M) *and* Eliminate(X,H) *are equivalent for* R*.*

Each action $x \in X$ has the form $l(c_1, ..., c_m)$ where $l \in L$, $m \geq 0$, and $\forall i \in [1..m], c_i \in C$, allowing the action x to carry a list of components present on the UI after the execution of the action. In this way, some UI interaction capabilities together with the resulted appearance are left out of the analysis.

To show that two UI models are equivalent modulo "X", we also consider strong and branching bisimulation relations. The *elimination* abstraction technique is necessarily used and the *generalization/omission* abstractions can be used or not.

6.5.4 Non-Equivalent User Interfaces

There are cases in which two UIs present a large number of divergences, requiring too much abstraction techniques to demonstrate their equivalence. In this case, the UIs are said non-equivalents. At the present time, decide whether or not too much abstraction techniques have been used requires a human intuition. Objective metrics to define thresholds under which the abstraction techniques can be used (without compromising the usefulness of the results) may be a future work of this thesis.

6.5.5 Inclusion of User Interfaces

Two UIs can still relate to each other in another way: one can include the other. For instance, in a control room, users have at their disposal UIs displayed on PCs to monitor the reactor.

Once a UI highlights an anomaly in a reactor parameter, mobile users (provided with a tablet containing only part of the PC-version UI) are charged to perform a maintenance in the proper place and can observe the system reaction on the tablet.

Definition 7 (**Inclusion of user interfaces**). *Given two* ISLTS *M and H, if a pre-order relation R exists between the states of M and H, then M and H are* included *one in the other with respect to R.*

Intuitively, it means that a given user interface U_1 contains *at least* all interaction capabilities (and the appearance) of another user interface U_2.

The pre-orders corresponding to the equivalence relations used to show equivalence between two LTS are used to show their inclusion (i.e., strong, branching, etc.). For instance, a pre-order of strong bisimulation is defined as follows [Park 1981]: An LTS M is included in another LTS H if there exists a relation $R \subseteq Q_M \times Q_H$ between the states of M and H such that:

1. The initial states of M and H are related by R;
2. If $R(m, h)$ and $m \xrightarrow{a} m'$, then there exists a state h' such that $h \xrightarrow{a} h'$ and $R(m', h')$.

Similarly to equivalence relations, *pre-order* relations also concern LTS states and actions. Thus, the UI interaction capabilities and appearance are considered in the analysis: given the form of an action in an ISLTS (i.e., $l(c_0, ..., c_m)$ where $l \in L$, $m \geq 0$ and $\forall i \in [1..m]\ c_i \in C$), actions are taken into account with the components present on the UI after the execution of the action.

Figure 111 illustrates an inclusion between two ISLTS. $H \leq M$ intuitively means that M can do everything that H can do, i.e., M includes H.

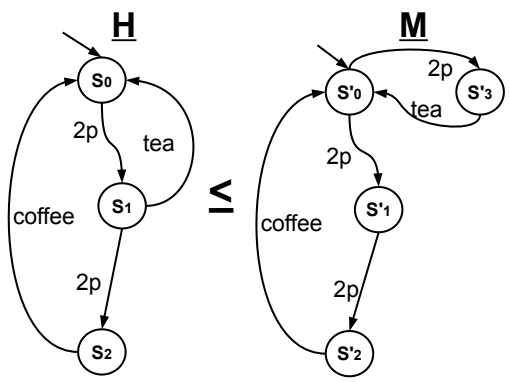


Figure 111: Example of an inclusion relation

The *generalization*, *omission* and *elimination* abstractions can be used to show that one UI model includes another one.

6.6 Application of the Approach

This section illustrates an application of the approach to the case study. We compare the UI versions with each other in several ways (Figure 112), and we can show two equivalent UIs,

two equivalent UIs modulo one functionality and two non-equivalent UIs that are, nonetheless, included one in the other.

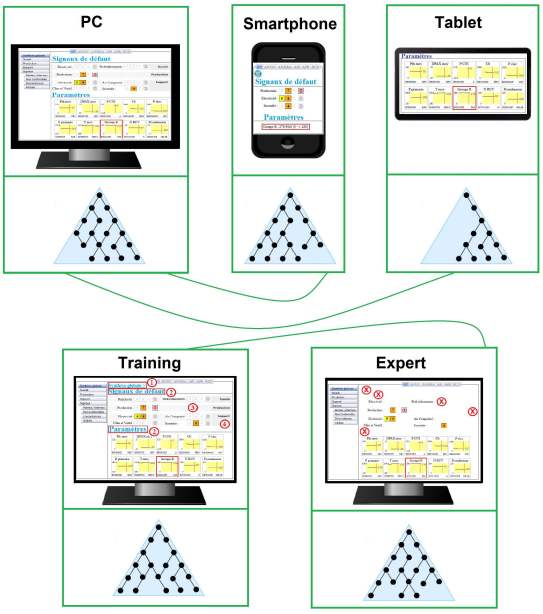


Figure 112: The five UI versions are compared by means of the ISLTS comparison

6.6.1 Equivalent User Interfaces

In order to demonstrate equivalence between two UIs, consider a UI adaptation according to the platform, to which re-molding is applied: PC (Figure 94) and Smartphone (Figure 55a).

Regarding the interaction capabilities of the user interfaces, users have two ways to interact with the UIs: by selecting the plant state and by accessing other user interfaces using the menu. Users can select the plant status in the same way on both UIs. This is reflected in the ISLTS of both UI formal models by identical states, actions and transitions. The menu, though, is made available in distinct ways: on the PC version the menu is always visible and on the Smartphone it is accessible by a button on the UI top-left corner. Due to these differences on the UIs, the corresponding ISLTS are different. This is illustrated in Figure 113, in which the ISLTS fragments represent part of the hierarchical menu. Each transition of these ISLTS fragments represents the action of choosing the corresponding menu and sub-menu options. In this case, "open menu" is an example of τ action: it is a user action that does not have an impact on the available menu options: they are always the same. We use *omission* abstraction to ignore the "open menu" action in the analysis, as if the menu was always visible on the Smartphone UI.

Concerning the UI appearance, both signals and parameters are displayed in the same zones. For this analysis we deliberately neglect the re-molding in the parameter widgets. We focus on the way both UIs display failures: on the Smartphone only the reactor parameters and signals with some failure are displayed, while on the PC all items are always displayed, even non-failure ones. Figure 114 illustrates such differences in an ISLTS fragment. Both frames on top of the

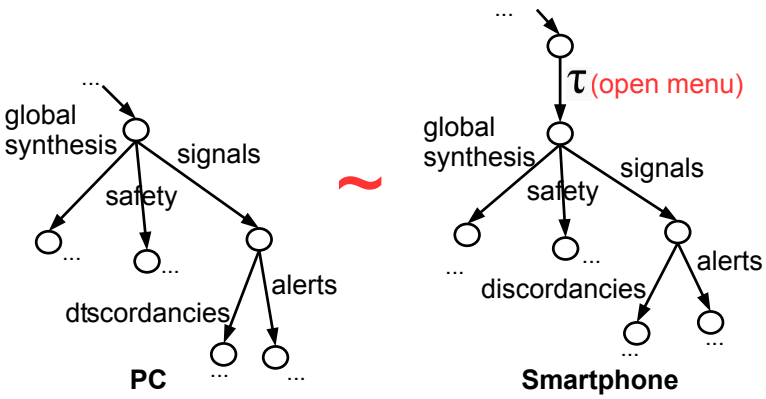


Figure 113: ISLTS fragments of PC and Smartphone UIs

figure represent at a given moment the display of reactor parameters on the UI. While on the PC ISLTS this transition is labeled with an action containing the whole list of reactor parameters, the Smartphone ISLTS contains only the problematic parameter (i.e., "Groupe R"). In this case, we use the *generalization* abstraction. Actions containing detailed information are generalized in less detailed actions (i.e., the bottom frames in Figure 114, with the action renamed into "Failure in x").

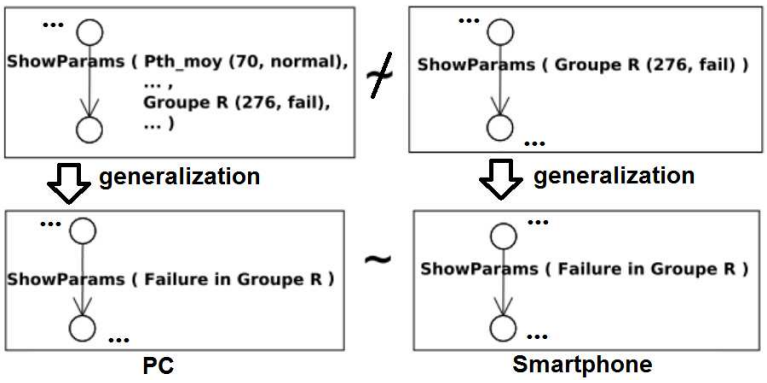


Figure 114: *Generalization* in an ISLTS - case 1

Using the *generalization* and *omission* abstractions, together with branching bisimulation relation, the PC UI model and the Smartphone UI model are equivalent.

6.6.2 Equivalent *Modulo the Breadcrumb Trail* User Interfaces

We demonstrate now two equivalent UIs modulo a particular functionality. Consider a UI adaptation according to the user expertise, to which re-molding is applied: training mode (Figure 56a) and expert mode (Figure 56b).

Regarding the appearance, there are several differences between both UIs (detailed in Section 3.4.2, page 78). We use *generalization* to represent differences n.2, 3 and 4 in Figure 56a.

Consider, for instance, the difference n.3: non-failure signal symbols have a disabled appearance in the training mode. Figure 115 illustrates the representation in an ISLTS: in training mode (i.e., the top-left frame), once a given signal is in non-failure status (e.g., *Reactivity* – "Reactivité"), the corresponding symbols are displayed with a disabled appearance (i.e., "disabled_sig"); in expert mode (i.e., the top-right frame), no symbols are displayed beside the signal (i.e., "empty"). Using the *generalization* abstraction, in both ISLTS these actions are generalized into "default_symbol" label).

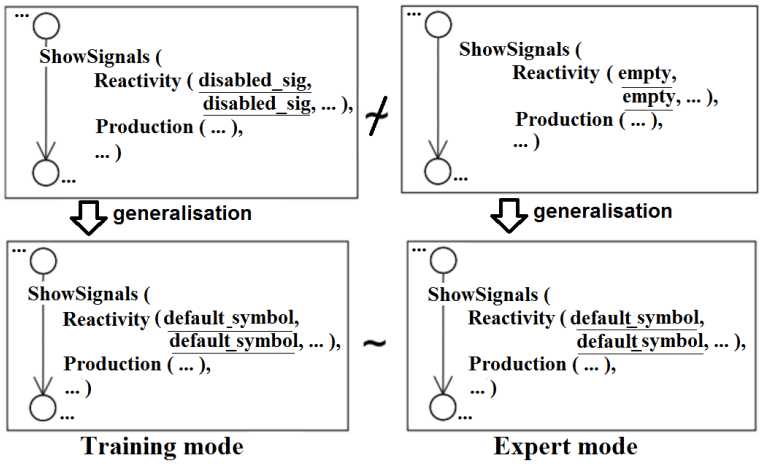


Figure 115: *Generalization* in an ISLTS - case 2

Concerning the UIs interaction capabilities, the training-mode UI contains one additional navigation aid: a breadcrumb trail (i.e., the difference n. 1 in Figure 56a). We set the equivalence checking to be done disregarding this feature. We use *elimination* abstraction (Figure 116) to search (in the ISLTS) actions corresponding to the breadcrumb trail (i.e., the pattern `bct_`). Once a match occurs, all the successor states (and transitions) are eliminated in cascade from the ISLTS.

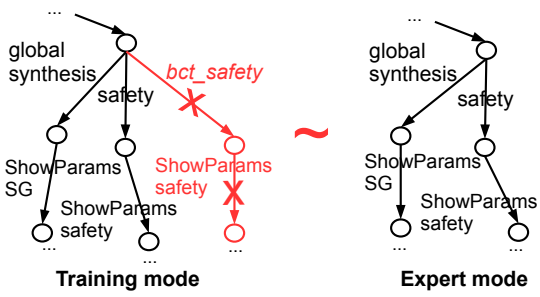


Figure 116: *Elimination* in an ISLTS

Using *generalization* abstraction (for the items n.2, 3, and 4 of Figure 56a) and *elimination* abstraction (for the item n.1), together with strong bisimulation relation, the training and expert UI models are equivalent modulo the breadcrumb trail.

6.6.3 Non-equivalent User Interfaces and Inclusion

We demonstrate now two non-equivalent UIs. Consider a UI adaptation according to the target platform, to which redistribution is applied: PC (Figure 94) and Tablet version (Figure 55b).

Regarding the UI interaction capabilities, the functionalities related to user interactions are available only on the PC version (i.e., the menu and the plant status selection). With respect to their appearance, the UIs also differ from each other: the tablet version does not contain the plant status, the reactor signals and the menu zones. The divergences of both UIs are too large to consider the use of *elimination* abstraction. Indeed, the tablet-version UI is equivalent to the PC version modulo ["plant-status-related actions", "signals-related action" and "menu-related actions"]. If we abstract all these actions away, many aspects are overlooked. In this case, applying no abstraction techniques, both UI models are non-equivalent, because the user can perform several actions on the PC version which are not available on the tablet version.

However, we can show that the PC version contains at least all functionalities (regarding the interaction capabilities and appearance) of the tablet version (Figure 117). The PC-version UI model included the tablet-version UI model (i.e., $Tablet_model \leq PC_model$).

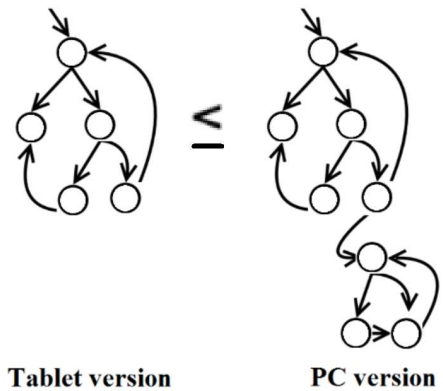


Figure 117: The PC version includes the Tablet version

6.6.4 Validation in the Case Study

An LNT formal model is manually written for the five contexts of use (i.e., PC, Smartphone, Tablet, Training and Expert Mode). The case study shows that the approach scales well. It was initially designed for one context of use (i.e., PC), later extended to five contexts of use. Each formal model contains three UIs. Each UI model describes about 20 curves and symbols (UI appearance) and 14 user interactions (UI interaction capabilities), generating significantly large ISLTS for the analysis in a reasonable time (maximum 3h). The formal models include also part of the functional core, allowing the simulation of several reactor parameters anomalies. The CADP toolbox is used to support the formal verification process. Table 14 summarizes the number of lines of the LNT specifications, the ISLTS size, and the ISLTS generation time. For larger case studies, CADP provides means (e.g., compositional verification, on-the-fly verification, etc.) to handle state-space explosion, a concern all model checkers have to handle. Model checkers address it by various manners, but human intuition is always needed in the process.

Table 14: Summary of the formal models in the different contexts of use

Context of use	# loc	# ISLTS states	# transitions	Time
PC	2462	33,053,947	189,539,691	1,5H
Smartphone	2558	41,944,680	208,554,613	3H
Tablet	1686	4438	5547	2s
Training mode	2579	160,681,601	946,293,368	1,5H
Expert mode	2410	16,678,151	76,202,201	10min

In order to illustrate how the sets C of UI components, L of action names, and A of actions are coded in LNT, consider the example described in Figure 107 (page 146). The set C is domain-dependent, i.e., is composed by subsets detailing the components of the UI according to the domain. In this example, an element $c \in C$ has the form $p(r, s)$, where $p \in P$, $P = \{Pth_moy, GroupeR\}$ is a subset of reactor parameter names, $r \in \mathbb{N}$, $s \in S$, and $S = \{normal, fail\}$ is a subset of reactor parameter status. In addition, in this example the set L is defined as $L = \{ShowParams\}$. Finally, each action $a \in A$ has the form $l(c_1, ..., c_m)$ where $l \in L$, $m \geq 0$, and $\forall i \in [1..m], c_i \in C$. Figure 118 illustrates how these sets are coded in LNT: by means of types. The set P is defined as a type called `TParamName` containing the reactor parameter names. Similarly, the set S is defined as a type called `TStatus` containing the reactor parameter status. The set C is defined as a type called `TParam` containing a list of parameters with their name, value, and status, and finally the set A is defined as a type called `TShowParams` containing an action name and a list of UI components of the set C.

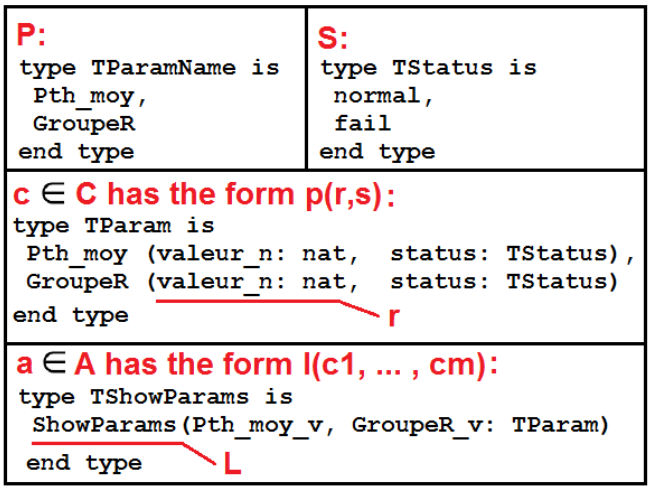


Figure 118: ISLTS in LNT code

The abstract techniques are implemented using SVL. While LNT was chosen mainly for its capacity to facilitate modeling, the choice of SVL considerably strengths the approach. SVL offers means to describing operations over LTS, which can hardly be done by hand in large LTS. The *generalization* abstraction is implemented using the `rename` SVL operator, *omission* is implemented using the `hide` operator and *elimination* using the `cut` operator, all together

Table 15: Summary of the comparisons

Models	# O	# G	# E	Result	Comp. time
PC x Smartphone	1	22	0	Equivalent	7min
Training x Expert	0	6	1	Equiv\breadcrumb	19min
PC x Tablet	0	0	0	Tablet included in PC	4s

with regular expressions.

```
1 "newLTS_PC.bcg" = generation of
2   total rename "SHOW_PARAMS !UI_.* (.*(\(.*\), FAIL).*)" ->
3                "SHOW_PARAMS (FAILURE in \1)"
4   in "LTS_PC.bcg"
```

Figure 119: Example of a SVL script

The SVL scripts isolate the original formal models from the abstractions. These scripts transform the ISLTS by applying the abstractions, before performing the equivalence verification. SVL scripts implement the three cases described in Section 6.6, page 151. Figure 119 illustrates an example of *generalization* in SVL, representing the example illustrated in Figure 114. Given the `LTS_PC.bcg` file (line 4), this script renames all transitions labeled with "`SHOW_PARAMS_[anyUI] ( [paramName], FAIL )`" into "`SHOW_PARAMS ( FAILURE in [paramName] )`" (lines 2-3), generating a new LTS with the renamed transitions (line 1). All SVL scripts are provided in Appendix D.

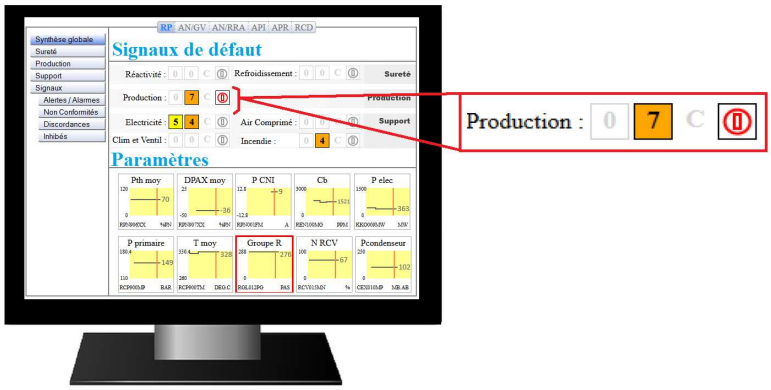
Once the ISLTS are transformed, they are compared with each other. Table 15 illustrates the summary of the comparisons, where `O` indicates the number of omissions done, `G` the number of generalizations and `E` the number of eliminations. The comparison of the ISLTS is done using either the BCG_CMP[2] or BISIMULATOR[3] [Mateescu & Oudot 2008] tools, provided by CADP. These tools check several equivalence relations between two LTS and generate a counter example if they are not equivalent.

6.6.5 Discussion

The abstraction techniques introduced in this chapter support UI model comparison. The principle is to first create abstract models of the UIs, used afterwards to perform equivalence checking. Figure 120 compares the abstraction techniques applied to a UI fragment that considers only appearance. Concerning the level of abstraction, the *generalization* technique is the one that abstracts the least, by mapping components into generic representations. *Omission* abstracts more, by obfuscating aspects in the model, and *elimination* is the most significant abstraction, that eliminates UI aspects of the model.

Figure 121 illustrates the different levels of equivalence between two UI models. The strongest equivalence relation two UI models can have is when, with none of these abstractions, they are

[2]http://cadp.inria.fr/man/bcg_cmp.html
[3]http://cadp.inria.fr/man/bisimulator.html



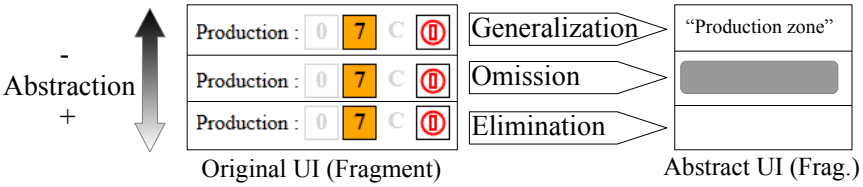


Figure 120: Summary of the abstraction techniques

equivalent. This is achieved only when two UIs are almost identical, which is possible, but rare. In practice, since plastic UIs have to cope with several changes in the context of use, numerous divergences are present within the UI versions. The challenge is to verify equivalence between the UI models in spite of these divergences. Abstraction techniques provide a means to do that, and weaker equivalence relations between the models can be shown. The more abstractions are applied to the models, the weaker the equivalence between the models becomes. Transversally, the inclusion between two UIs can be verified at any level of abstraction.

Once the appearance of two UIs diverges, *generalization* is the first technique to be considered. By generalizing the representation of UI components in the models, this technique permits a weaker equivalence between models to be shown.

Omission and *elimination* are more likely to be used when the UI interaction capabilities diverge. First, by omitting in the UI models interaction capabilities that are punctual, such as the opening of a menu, or the opening of a combobox, and that do not have a considerable impact in the functionalities of the UIs. Diversely, elimination is recommended for complex interaction capabilities. In particular, UI functionalities that depend on the functional core, that load information on the UI, or that enable other UI functionalities. Once such interaction capabilities are present in one UI and absent in the other, elimination provides a means to verifying the equivalence between both UI models disregarding (modulo) that.

However, abstraction techniques should be carefully used. The abstraction should never exceed a threshold (manually identified) over which the analysis is no longer interesting. One should keep in mind that things that are abstracted away are left out of analysis, and interesting situations may be overlooked when the system models become a black box (Figure 122).

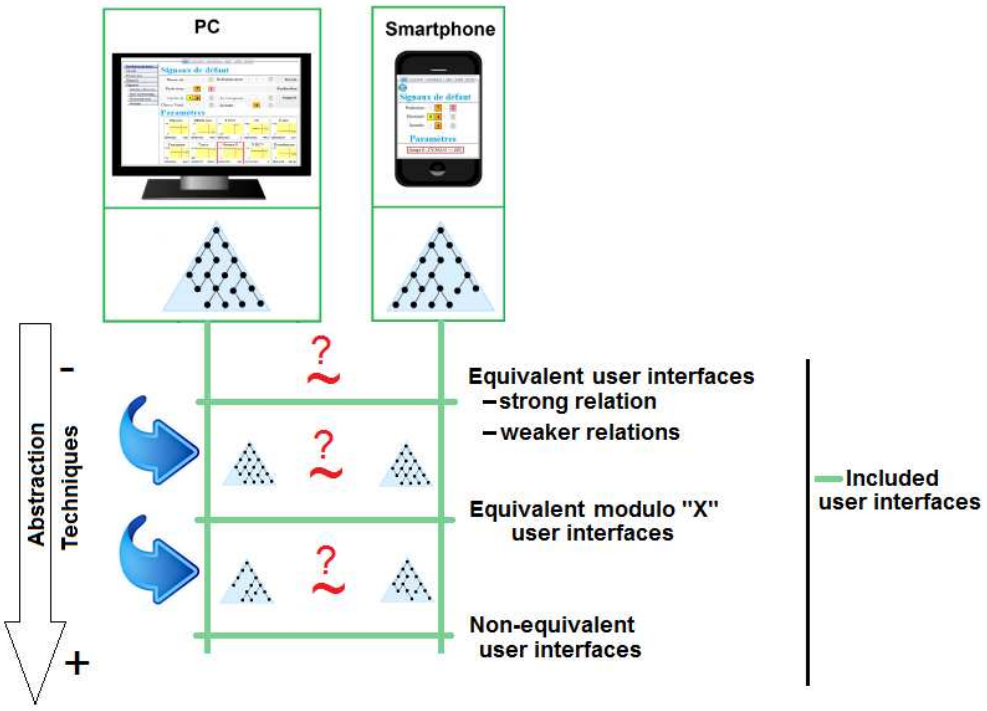


Figure 121: Different levels of equivalence between UI models

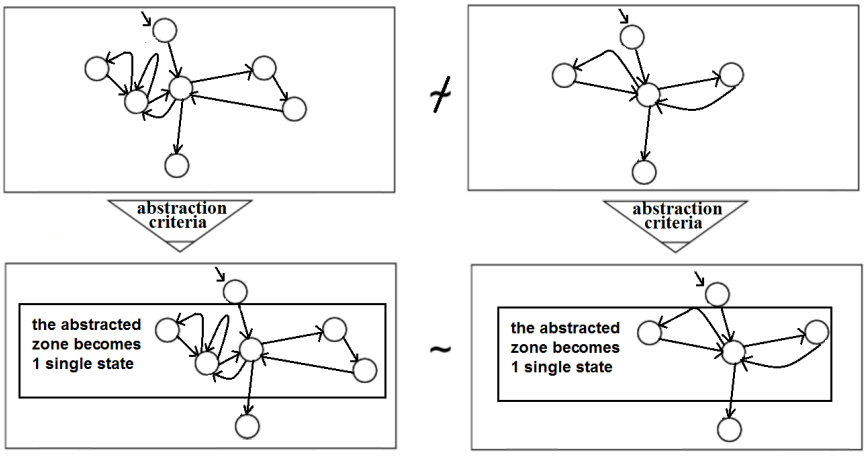


Figure 122: The abstraction problem

6.7 Summary

This chapter introduces plasticity into our global approach to verifying interactive systems. The formal model of the case study described in Chapter 3 is subject to several improvements to include plasticity. Specially, two new modules are included in the formal model, to describe two

different UI zones: parameters and signals.

We describe and compare three propositions to verify plasticity: the verification of properties by means of model checking either over the different UI versions or over the plasticity engine, and a technique to compare formal models by means of equivalence checking. We give the rationale of the chosen proposition, which is then detailed. It consists in comparing different versions of user interfaces.

Two UI aspects are considered in the comparison: the interaction capabilities and the appearance. The former concerns the ways users can interact with the UI and, conversely, how the UI reacts to this interaction. The latter concerns the elements present on the UI (where they are presented, in which color, etc.). We provide a means to representing both UI interaction capabilities and appearance in one single model derived from LTS, called ISLTS (*Interactive System LTS*).

Our approach indicates to which extent two UIs have the same interaction capabilities and appearance. Four levels of equivalence are proposed: equivalent UIs, equivalent modulo "X" UIs (which can ignore some UI divergences), non-equivalent UIs, and one UI included in another one (meaning that one UI contains at least all interaction capabilities and appearance of another one). When the UIs are not equivalent, the UI divergences are listed, which is a significant contribution of this work. Such divergences can be bypassed, to reason over the UIs disregarding them.

The formal framework supporting the technique presented in this chapter is a major contribution of this thesis. It moves towards the application of formal methods to the verification of interactive system, by covering plasticity.

Chapter 7

Validation of the Use of Formal Verification for Interactive Systems

Contents

7.1 Goals **161**
7.2 The SRI Display System **162**
7.3 Formal Model **164**
7.4 Verification Approach **167**
7.5 Properties **168**
7.6 Results and Discussion **169**
7.7 Summary **172**

7.1 Goals

In our approach, the Lnt formal specifications are manually written. Writing formal specifications is a useful exercise by itself. It helps one to familiarize with the formal specification languages and tools. However, create models by hand adds a certain overhead to the process. This is lightened by the fact that the formal specification languages we use are designed to facilitate formal modeling. This comes mainly from the fact that Lnt is an imperative language, which facilitates learning for programmers accustomed to classic programming languages. Such manual modeling requires knowledge not only in the system to be modeled, but also in the languages, tools, and formal methods to be applied. Thus, acquired experience in formal modeling is a non-negligible skill, and may impact on future applications of the proposed verification approach.

In this chapter, we aim at validating two hypotheses: whether the guidelines we proposed in Section 5.2.7 (page 105) facilitates the formal modeling of interactive systems, and whether the experience acquired in previous applications of formal methods facilitates further work. To investigate these hypotheses, we apply our verification approach to another case study in the nuclear-plant domain, developed in the Connexion Project by two industrial partners: Rolls-Royce Civil Nuclear[1] and Esterel Technologies[2]. This case study consist of a specific display system called SRI (*Système de Réfrigération Intermédiaire*) [Connexion 2014], used in nuclear-plant control rooms to display acquired and synthesized data. The system analyzed in this case study does not have plastic user interfaces. The functional core is not covered by the modeling, thus, neither it is the dialog controller. We analyze aspects of the system user

[1] `http://www.rolls-royce.com/customers/nuclear.aspx`
[2] `http://www.esterel-technologies.com/`

interfaces and the user behavior. In the sequel, we describe this case study, the modeling of the system by our approach, the verification that can be performed thanks to the model checking part of our global approach, and the flaw that is pointed out on the case study thanks to our approach.

7.2 The SRI Display System

A *displayer* is a reduce-sized equipment (i.e., around 20 x 15 cm) which embeds a programmable software. A displayer allows users to interact with a *display system*, it is constrained to provide certain functions of the display system, and it is not connected to any external system. A display system of a computerized control room typically exhibits acquired data, treated data (i.e., acquired data after numerous treatments), and data accumulated over time.

A displayer has four modes (Figure 123). When the displayer is turned off, it is considered to be in the *stopped* mode. Once it is turned on, it transits to the *initializing* mode, from which it can transit to the *functional* mode if the initialization ends well, or to the *frozen* mode if some problem occurs. From the functional mode, it can also transit to the frozen mode if some problem occurs. Finally, from any mode it can be turned off, transiting, thus, back to the stopped mode (except if it is already in the stopped mode). The user can interact with the displayer only when it is in the functional mode.

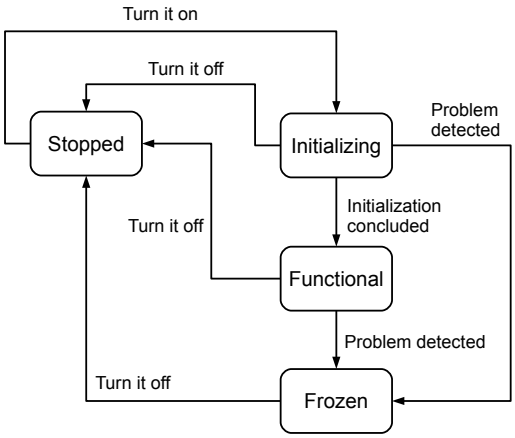


Figure 123: Displayer modes

This case study concerns part of a display system called SRI (*Système de Réfrigération Intermédiaire*), the main UI of which is illustrated in Figure 124. Such display system structures its user interface into two zones (Figure 125): a header and a main zone.

In the header, the following information is displayed and functionalities are made available:

1. *Life sign* informs whether the displayer is functional or not;

2. *Previous UI* permits the UI previously displayed to be re-accessed;

3. *Main UI* provides a shortcut to the Main UI of the system;

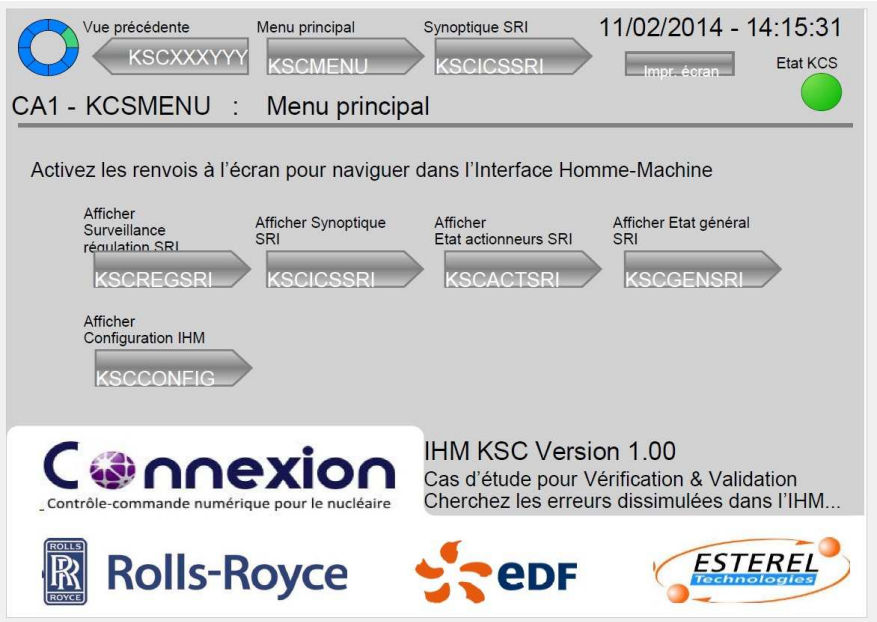


Figure 124: Main UI of the SRI system

4. *SRI mimics UI* provides a shortcut to the SRI mimics UI;
5. *Print* prints the UI currently displayed;
6. *Date* displays the current date;
7. *Time* displays the current time;
8. *Unit ID* displays the ID of the nuclear-plant unit;
9. *UI name* displays the name of the UI currently displayed; and
10. *Status* indicates the status of the displayer. Such indicator is assigned to a flashing sign to indicate a problem in the displayer, or it is assigned to a fixed sign otherwise. For each case, the status indicator has a different color.

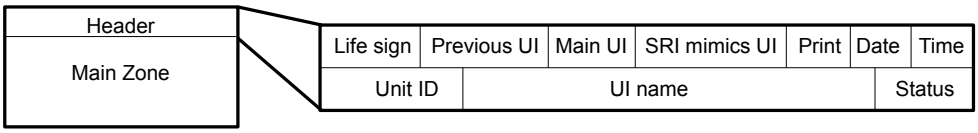


Figure 125: Organization of the UI zones and data

In the main zone, six UIs can be displayed:

1. *Main UI* (Figure 124) contains five buttons allowing one to access the other UIs;

2. *SRI mimics* displays the water circuit of the SRI system, and the departures and arrivals of condenser circuits of the `SEN` elementary system;

3. *SRI regulation curves* displays several curves representing the historical values of several SRI components;

4. *SRI status* displays the general status of the SRI system;

5. *Status of SRI actuators* displays the current status of several actuators of the SRI system; and

6. *Settings* permits the configuration of several parameters, which are needed to the displayer auto stimulation.

A displayer can also be used in simulation mode, in which the acquisition of data is simulated in the displayer by auto stimulation. In case of the SRI system, such auto stimulation is configurable by the *Settings* user interface (Figure 126).

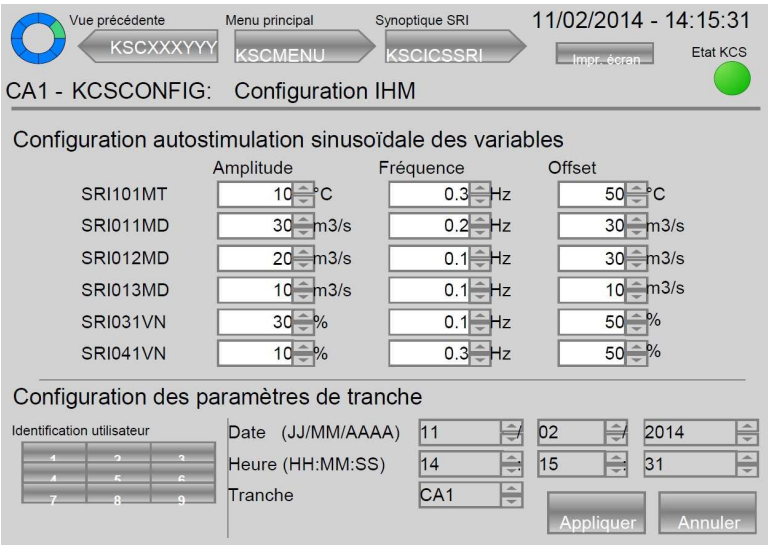


Figure 126: The *Settings* UI of the SRI system

7.3 Formal Model

The guidance we extracted from the modeling of the EDF case study was relevant to reduce the modeling time of this case study. The following guideline was proposed in Section 5.2.7 (page 105): (1) follow one architectural model (e.g., ARCH, PAC, etc.) to structure the model; (2) define the modules that will form the architectural model components, and how these modules communicate; (3) to define the UI modules, identify the UI zones with which the user can interact; (4) folding/unfolding of menus can be modeled by a set of rules which express when each menu option is available; (5) to model the functional core, pay attention to the

aspects that will be subject to verification afterwards; and finally (6) to model users, identify all actions that users can execute on the user interfaces.

To model this case study, the guidelines n. 2, 3, and 6 are reapplied: the formal model is structured before it is implemented (Figure 127); a module is created for each UI zone (i.e., *header* and *main zone* modules); and the user actions are the basis to model the *user* module. The criteria n. 1, 4, and 5 are not applicable due to the size of this case study: it neither covers the functional core, nor has a menu.

The structure of the formal model is composed of four modules:

1. *displayer modes* models the four modes in which the displayer can be, and the rules to transit from one mode to another;

2. *header* includes the functionalities related to user interactions present in the header of the UI, i.e., *Previous UI*, *Main UI*, *SRI mimics UI* and *Print*;

3. *main zone* models the navigation between the six UIs of the system; and

4. *user* models some of the possible user interactions with the system.

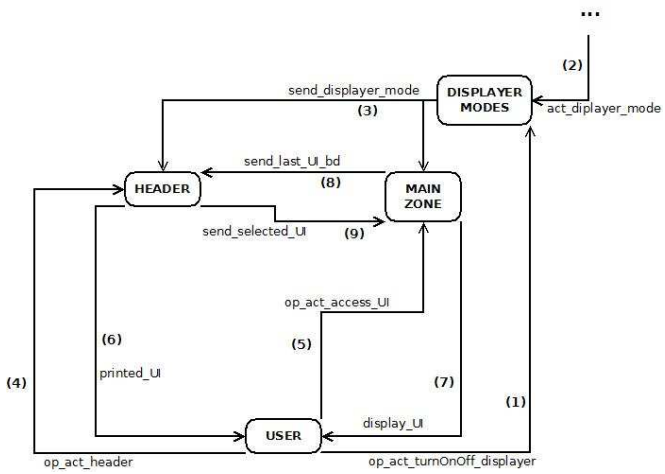


Figure 127: Formal model structure of the SRI system

These modules communicate with each other in the following way: initially, the displayer is turned off, and the only possible action is to turn it on, which is done by a communication between the *user* and the *displayer modes* modules (1). The displayer can also have its mode changed by other sources besides the user (2). For instance, when the user turns the displayer on, it changes to the *initializing* mode, and after a while, it changes by itself to the *functional* mode, with no intervention from the user. The *displayer mode* module sends its current mode to the *header* and the *main zone* modules (3), since the user can interact with those zones depending on the displayer mode. The user interacts with these zones by choosing functionalities of the header (4), or by choosing the UI she/he wants to access in the main zone (5). The *header* sends to the *user* the printed UI (6), in case the user selects the *print* functionality of the header. The *main zone* sends to the *user* the UI she/he chose (7). Finally, at each selection

of the user, the *main zone* informs the *header* the last UI chosen (8), (required to the *Previous UI* functionality of the header), and the *header* sends to the *main zone* the UI the user selected (9), in case she/he interacts with the *Previous UI*, *Main UI*, or *SRI mimics UI* functionalities of the header. This formal model is specified using the LNT language. Table 16 gives figures about the formal model.

Table 16: Summary of the formal model of the SRI system

#	File	Description	# loc
1	displayer modes	rules to change among four modes	30
2	header	selection among four functionalities	44
3	main zone	navigation between six UIs	59
4	user	interactions with the header, the main zone, and the displayer mode	58
5	main	entry point of execution of the model	59
6	library	common functions	54
		TOTAL	**304**

Table 17 shows the size of the LTS generated from the LNT formal model, using CADP. This LTS is used for the verification of properties.

Table 17: Size of the LTS of the SRI formal model

# states	# transitions
1469	2868

Figure 128 illustrates the functions users can perform by interacting with the system. With a "check" symbol we identify those that are covered by the formal model of the system, and with a "not OK" symbol those that are not.

The following functionalities of the display system are covered by the modeling:

1. to turn on the displayer;
2. to turn off the displayer;
3. to display the *Settings* UI;
4. to display the *Main UI*. This UI is fully modeled, since it gives access to the other UIs;
5. to display the *SRI regulation curves* UI;
6. to display the *SRI mimics* UI;
7. to display the *Status of SRI actuators* UI;
8. to display the *SRI status* UI; and
9. to print the UI currently displayed.

And the following functionalities/aspects of the display system are **not** covered by the modeling:

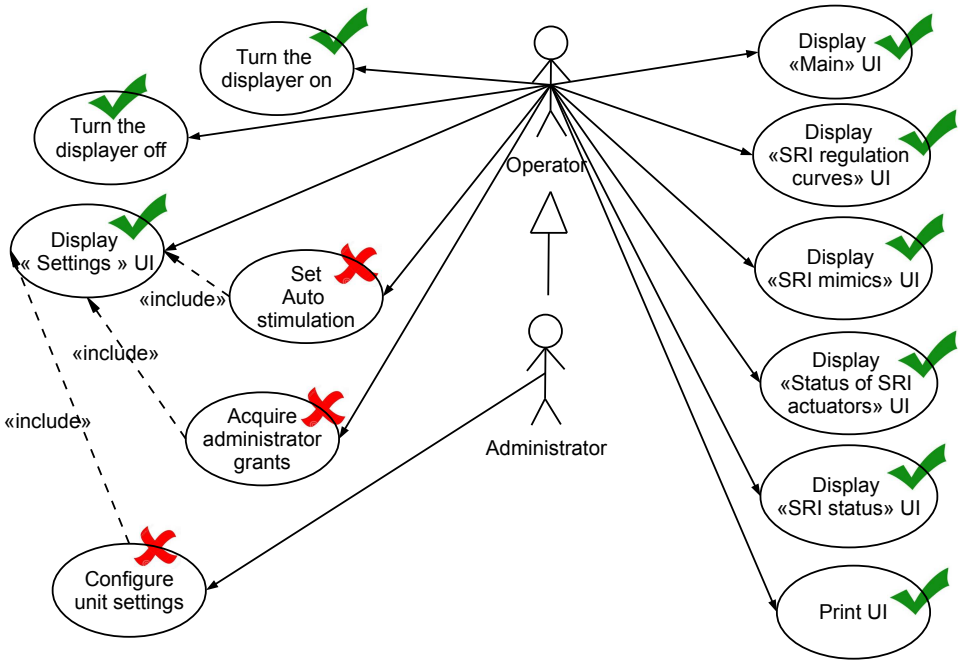


Figure 128: Functionalities of the displayer. In green, the ones covered by the formal model. In red, otherwise

1. to set the displayer to auto stimulation operation mode;
2. to manage the user profiles. Two profiles are available in the system: operator and administrator, but we bypass such information in the model;
3. to configure other settings of the displayer, such as date and time (Figure 126);
4. the details of five UIs of the SRI system, i.e., *SRI mimics*, *SRI regulation curves*, *SRI status*, *Status of SRI actuators*, and *Settings*;
5. the information that is displayed in the header of the UIs (e.g., date and time);
6. some complementary functionalities of the displayer, such as auto test and auto stimulation of variables; and
7. all the visual aspects of the user interfaces are not modeled either (e.g., the curves, mimics, and graphical objects).

7.4 Verification Approach

In Chapter 4 we describe our global approach to verifying interactive systems (Figure 63). In the SRI case study, we apply the model checking part of the approach, which consists in verifying a set of properties over a formal model of the system. In Chapter 2 we argue that one way to

verify systems with respect to their specification is by extracting properties from the requirement document of the system, and to use model checking to verify whether the system is specified according to the specifications or not. We follow this strategy to identify the required properties in this case study. Figure 129 illustrates a derivation of our model checking-based approach, in which the source of the identified properties is the requirement document.

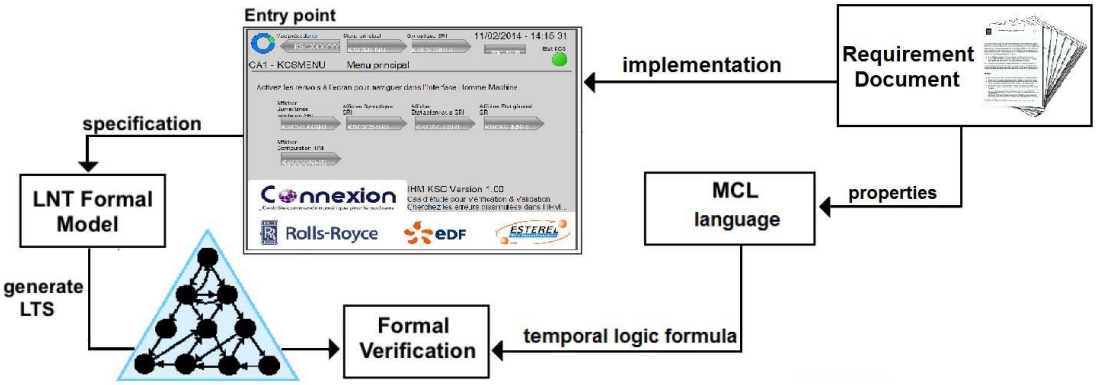


Figure 129: Verification approach of the SRI system

All steps of our model checking approach are found here: the formal modeling of the system using the LNT language, the transformation into an LTS representation, the specification of properties using the MCL language, and the verification of such properties over the model using the CADP toolbox. The specificity here is the requirement document: besides being the basis to create the real system, it is also from this document that the properties to be verified are extracted.

7.5 Properties

The SRI system documentation contains around 285 requirements to specify, among others, the appearance, interaction capabilities, and navigation of the user interfaces. We formalized some requirements related to UI interaction capabilities and navigation. From the documentation, 48 requirements are extracted and expressed as MCL properties. *Usability* and *functional* properties are identified. These requirements are concealed for confidentiality reasons. We illustrate an example of requirement that is satisfied by the system model and one requirement that is not. Consider the following requirement:

Property 1): "[A23_1_FCT_MISE_HORS_TENSION_LIEN] Turning off the displayer should make unavailable all other functions of the displayer."

We express this requirement in MCL as a *safety*[3] property in the following way:

[3](cf. the definition of *safety* properties in Section 5.2.8, page 106)

$$
\begin{array}{l}
[\ true^* \ . \\
\ \ "OP_ACT_MODE_DISPLAYER\ !TURN_IT_OFF" \\
\ \ ('.^*ACT_.^*')^* \ . \\
\ \ "OP_ACT_MODE_DISPLAYER\ !TURN_IT_ON" \\
]\ false
\end{array}
$$

This formula expresses that we want to avoid the following sequence:

starting from any state,
the displayer is turned off,
then actions can be performed on the displayer ...
... before the displayer is turned on again.

Another example of requirement follows:

Property 2): "[A23_1_VUE_ENCHAINEMENT] The UI must allow at least the navigation of UIs of Figure 130:"

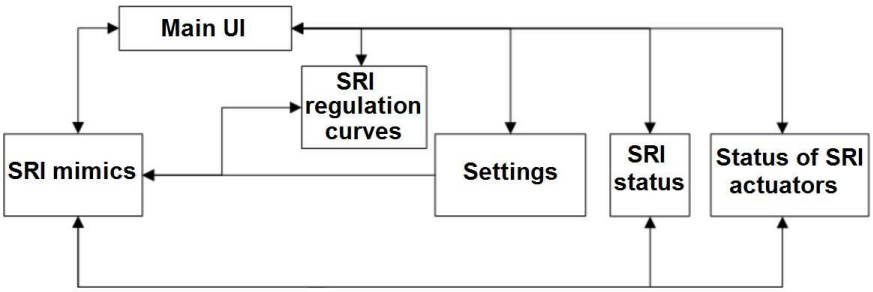


Figure 130: Graph of UI navigation of the SRI system

In Figure 131, the MCL property that expresses this graph of UI navigation describes the UIs that can be accessed from which UI of the system, either by the five buttons available in the *Main UI* or by the buttons available in the header of the system (Figure 124).

7.6 Results and Discussion

The property 1 is verified over the system model, which indicates that the system follows this requirement as expected. The property 2, however, is not satisfied over the system model. The UI navigation graph illustrated in Figure 130 requires a direct access between the *Settings* UI and the *SRI Status* UI and vice versa. We express such direct access is the MCL property. By analyzing the counter-example produced by the formal verification (Figure 132), we observe that this direct access is not possible in the model of the system. Abstracting the internal actions of the formal model needed to the exchange of information between the modules, the

```
[ true* . "AFFICHER_VUE !PRINCIPAL"]            < ('OP_ACT_ACCEDER_VUE.*' and ("OP_ACT_ACCEDER_VUE !SYNOPTIQUE_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !COURBES_REGUL_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !PARAMETRAGE" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_GENERAL_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_ACTIONNEURS_SRI")) > true
and
[ true* . "AFFICHER_VUE !SYNOPTIQUE_SRI"]       < ('OP_ACT_ACCEDER_VUE.*' and ("OP_ACT_ACCEDER_VUE !COURBES_REGUL_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_GENERAL_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_ACTIONNEURS_SRI")) > true
and
[ true* . "AFFICHER_VUE !COURBES_REGUL_SRI"]    < ('OP_ACT_ACCEDER_VUE.*' and ("OP_ACT_ACCEDER_VUE !PARAMETRAGE" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_GENERAL_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_ACTIONNEURS_SRI")) > true
and
[ true* . "AFFICHER_VUE !PARAMETRAGE"]          < ('OP_ACT_ACCEDER_VUE.*' and ("OP_ACT_ACCEDER_VUE !COURBES_REGUL_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_GENERAL_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_ACTIONNEURS_SRI")) > true
and
[ true* . "AFFICHER_VUE !ETAT_GENERAL_SRI"]     < ('OP_ACT_ACCEDER_VUE.*' and ("OP_ACT_ACCEDER_VUE !COURBES_REGUL_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !PARAMETRAGE" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_ACTIONNEURS_SRI")) > true
and
[ true* . "AFFICHER_VUE !ETAT_ACTIONNEURS_SRI"] < ('OP_ACT_ACCEDER_VUE.*' and ("OP_ACT_ACCEDER_VUE !COURBES_REGUL_SRI" or
                                                                               "OP_ACT_ACCEDER_VUE !PARAMETRAGE" or
                                                                               "OP_ACT_ACCEDER_VUE !ETAT_GENERAL_SRI")) > true
and
[ true* ] < true* . "OP_ACT_BD !VUE_PRINCIPALE" > true
and
[ true* ] < true* . "OP_ACT_BD !VUE_SYNOPTIQUE_SRI" > true
```

Figure 131: A property of the SRI system in MCL

counter-example gives the following sequence of actions that leads to a state where the property is false: from the initial state (1), the displayer is turned on (2), it transits to the *initializing* mode (3), and then to the *functional* mode (4), the *Main UI* is displayed (7), from which the button giving access to *Settings* UI is clicked (8), and this UI is displayed (10). At this state, there is no action allowing the *SRI Status* UI to be accessed, thus, the property is not satisfied.

1	<initial state>
2	"OP_ACT_TENSION_AFFICHEUR !SOUS_TENSION"
3	"ENVOYER_MODE_AFFICHEUR !INIT"
4	"ACT_MODE_AFFICHEUR_NOMINAL"
5	"ENVOYER_MODE_AFFICHEUR !NOMINAL"
6	"ENVOYER_DERNIERE_VUE_BD !PRINCIPAL"
7	"AFFICHER_VUE !PRINCIPAL"
8	"OP_ACT_ACCEDER_VUE !PARAMETRAGE"
9	"ENVOYER_DERNIERE_VUE_BD !PRINCIPAL"
10	"AFFICHER_VUE !PARAMETRAGE"
11	<goal state>

Figure 132: Counter-example of the non-satisfied property (in French)

Looking at the real system, we can also observe the flaw pointed by the formal verification of the system model. A screenshot of *Settings* UI in Figure 133 shows that indeed it is not possible to directly access the *SRI Status* UI. The UI contains several buttons, but none of them provide direct access to the aforementioned UI.

A further analysis with the designers of the system indicates that the *Settings* UI is not supposed to provide direct access to *SRI Status* UI, and that the joint arrows of the Figure 130 of the requirement document are ambiguous: they suggest several UI navigations by direct accesses that are not implemented in the real system.

In this case study, 48 requirements (among the 285 requirements of the SRI documentation)

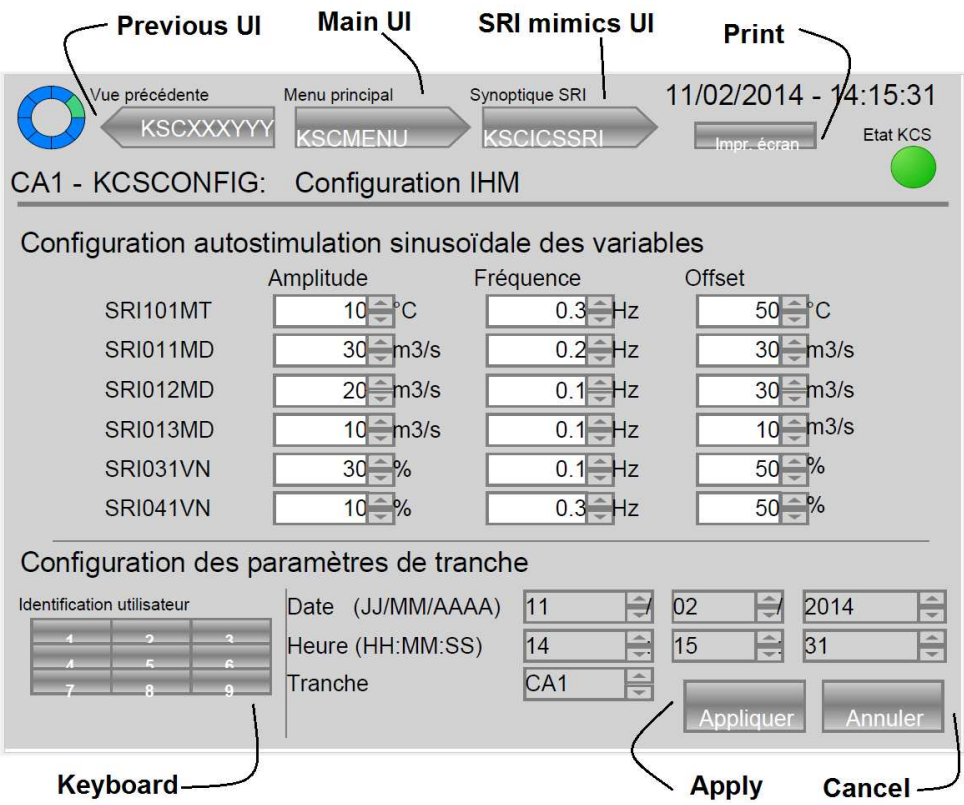


Figure 133: The *Settings* UI does not provide direct access to the *SRI Status* UI

of the system are formalized and verified. All pass, except the one about UI navigation.

This case study has shown that formal methods can be used to verify whether or not a system is implemented according to the requirement documents, helping to find system flaws originated from ambiguous and imprecise requirements. The benefit is twofold: it helps to find system flaws, and to improve the precision of the requirement document.

The goals of this case study were: to verify whether the techniques proposed in this thesis and the experience acquired in previous applications of formal methods facilitate further applications of formal methods to industrial systems. Following the guidance we extracted from the modeling of the EDF case study (cf. Section 5.2.7 on page 105), the modeling of the SRI system was considerably less time-consuming and effort-consuming than the main case study of this thesis described in Chapter 3. Together, the modeling of the system in LNT and the formalization of properties in MCL took one week of work. Some properties are similar to each other, so the usage of MCL *safety*, *liveness*, and *fairness* patterns of properties[4] helped the development to be faster.

This case study corroborates the usefulness of our approach in the verification of systems with different scales: it can also be applied to smaller systems such as the SRI system and it still provides interesting analyses. It also shows that, even when the formal model does not

[4](cf. Section 5.2.8 on page 106)

cover the whole system, interesting results can still be obtained using formal methods.

Furthermore, the support of the approach in the verification of systems with respect to the requirements is an important contribution in the context of critical systems. This kind of systems are constrained to numerous and complex requirements, which are sometimes ambiguous and imprecise. The rigorousness provided by formal methods to verify these requirements improves system quality, and the requirements themselves.

7.7 Summary

To investigate the initial hypotheses of this chapter (i.e., whether the guidelines proposed in Section 5.2.7, page 105, and acquired experience in formal methods facilitate further work), we apply our verification approach to another case study in the nuclear-plant domain: an elementary display system called SRI (*Système de Réfrigération Intermédiaire*). Only part of our global verification approach is used: the property-verification part. Specifically, this part of the approach is used to verify whether the system is implemented according to its requirement document.

In this case study, 48 (among 285) requirements are formalized as properties. The property verification over the system model is performed using the CADP toolbox, and the properties are all satisfied except one. This permitted a flaw to be identified in the system documentation: the navigation between the UIs of the real system is not implemented as specified in the requirement document.

This case study lead us to conclude that both initial hypotheses are valid. Our guidelines facilitated the formal modeling of this case study, and the time and effort required to the formal modeling of this case study was considerably inferior to the main case study of this thesis. The acquired experience not only in the tools that are used, but also in the formal modeling of interactive systems in general was of relevant help.

Chapter 8

Conclusion and Perspectives

Contents

8.1 Summary of Contributions **173**
8.2 Perspectives **175**
8.2.1 Short-term Perspectives 176
8.2.2 Mid-term Perspectives 176
8.2.3 Long-term Perspectives 177

The challenge of this thesis was to verify safety-critical interactive systems with plastic user interfaces (UIs). Our main contributions are reminded in the next subsection.

8.1 Summary of Contributions

In order to answer the initial research question, which was "*how to improve quality of safety-critical interactive systems with plastic user interfaces, in a way which permits a rigorous verification of the system and is scalable for industrial applications?*", we centered our work around five main axes: (i) a global approach to verifying interactive systems; (ii) the application of the approach to the nuclear-plant domain; (iii) the validation of the ADACS-N^{TM} part implementing the EDF system; (iv) the formalization of a technique to verify plastic user interfaces; and (v) the demonstration that our approach, together with a set of guidelines and the experience gained in formal methods, reduce the time and effort spent in new case studies.

Our *first contribution* is a global verification approach that integrates two formal techniques to assess the quality of interactive systems: *model checking*, for the verification of desired properties, and *equivalence checking*, for the verification of consistency between different versions of a system. This global approach is generic: it allows non-plastic interactive systems to be verified by the model-checking part, and it moves towards the verification of systems, by covering also the verification of plastic interactive systems using equivalence checking. Our approach allows aspects of the system such as the user interfaces, the functional core, and the users' behavior to be analyzed. Besides, our approach allows usability and functional properties of the interactive systems to be verified. In the context of safety-critical systems, this is particularly relevant, in order to demonstrate that the system guarantees ergonomic properties and safety requirements. The global approach is supported by powerful and mature tools and languages (the Cadp toolbox, the Lnt, the MCL language, and the Svl languages).

Our *second contribution* is the demonstration of the applicability of our approach to safety-critical systems, specifically, to the nuclear power plant domain. For this, we precisely described the first part of the global approach: the use of model checking for the verification of desired properties. A system prototyped by EDF in the context of the Connexion Project has been studied. This system aims at providing a global overview of a reactor in a nuclear power

plant. We modeled this system, and a set of usability and functional properties have been verified. Several insights on how to formally model interactive systems have been gained from the modeling of this case study (cf. Section 5.2.7 on page 105), and can be reused in the future. This work tackles the specificities of nuclear plants identified in Section 1.5, page 7: (1) it allows exhaustive reasoning of the a system model; (2) constraints in the user interfaces can be verified; (3) the variety of user profiles can be taken into account (by our proposition to verify plasticity); and finally (4) safety requirements of such systems can be verified.

Our *third contribution* is an application of our approach to validate part of ADACS-N^{TM}, a system integrating all functions of a nuclear control room. Specifically, we analyzed part of ADACS-N^{TM} corresponding to the EDF system. We used our approach to analyze several ADACS-N^{TM} traces, so as to give clues about whether the system implements the specifications correctly or not. This analysis permitted several improvements on the formal model (c.f. Subsection 5.4.1 on page 114) and on the industrial system (cf. Subsection 5.4.2 on page 116). Besides, a parser allowing ADACS-N^{TM} to be connected to the formal model has been developed, which can inspire future connections of formal specifications to other industrial systems. This study has shown that a coupling between the formal model and an industrial system can be very fruitful. First, such connection can be achieved by several means. In Section 5.3.2 (page 110) we proposed three means, namely analysis of traces, test case generation and co-simulation. This illustrates that the use of formal models goes beyond the verification of properties. This is relevant in the industrial context, in which few cases of use of formal methods have been reported [Miller 2009]. Secondly, in order to connect to a real system, the formal model should describe the concerned parts of the system in the same way they were implemented in the system. Such requirement forces a better understanding of the modeled system, which can only have positive effects in both the model and the implementation.

Our *fourth contribution* consists of a fine-grained description of the second part of our global approach: the verification of consistency between different versions of a system aiming at verifying plastic interactive systems. We use equivalence checking to compare the models of multiple versions of a plastic user interface (UI) with each other, in the light of different behavioral equivalence and pre-order relations. Two UI aspects are covered by the analysis: the interaction capabilities and the appearance. This coverage suits well the problematic of plastic user interfaces, in which the UI behavior and appearance adapt according to changes in the context of use, while preserving usability [Thevenin & Coutaz 1999]. We proposed a formal framework in which four levels of equivalence between UIs are provided: equivalent UIs, equivalent modulo "X" UIs (in which some UI functionalities are ignored), non-equivalent UIs, and included UIs. The formalization of a variety of equivalence levels allows a wide-ranging analysis: from UIs that are very similar to UIs that are very divergent. Besides, once two UIs diverge too much, our approach still permits to show that one includes the other. Our approach is applicable to any plastic UI, and it is even more legitimate in safety-critical systems, since problems in the UIs of such systems have strong implications.

Our *fifth contribution* is the demonstration that our approach, together with a set of guidelines and the experience gained in formal methods, reduces the time and effort spent in new case studies. This has been confirmed by the application of our approach to another case study in the Connexion Project, named SRI system ("Système de Réfrigération Intermédiaire") [Connexion 2014] (cf. Chapter 7). The insights identified in the modeling of the EDF system guided the modeling of the SRI system, which reduced its modeling time. In this case study, our approach has been used to verify whether the SRI system was specified according to the requirement

documents, and a flaw has been identified in the documentation. Besides, in this case study we demonstrated the usefulness of our approach in the verification of systems with different scales: it can also be applied to smaller system such as the SRI system and still provides interesting results.

Table 18 places our approach with respect to five characteristics we believe a verification approach of interactive systems should have: (i) our modeling *covers* aspects of the users, the user interfaces, and the functional core of the system; (ii) we model and verify *plastic UIs*; (iii) the verification covers usability and functional *properties*; (iv) the approach is *applicable* to the nuclear plant domain; and (v) the approach may *scale* well. The table also places our approach with respect to the different approaches presented in Chapter 2.

Further case studies are needed, though, to confirm the scalability of the approach. The three case studies described in this thesis (i.e., the modeling and verification of the EDF system, the analysis of the ADACS-N^{TM} industrial system, and the modeling and verification of part of the SRI system) are relatively representative, but they were not fully verified. They demonstrate, though, the feasibility of the approach. Besides, together with the powerful tool-support of the approach, these case studies demonstrate its potential scalability.

A limitation of the approach is that, so far, all *usability* properties that have been demonstrated are reachability properties (according to the framework of ergonomic properties proposed in [Abowd *et al.* 1992]). Static properties related to the appearance of the user interfaces, for instance, are not covered by the model-checking part of the approach (even though they are covered by the equivalence-checking part).

Table 18: Summary and comparison with the state of the art

			Authors *et al.*																					
			Abowd	Paterno	Markopoulos	Duke and Harrison	Campos	D'Ausbourg	Bumbulis	Palanque	Dix	Bowen and Reeves	Aït-Ameur	Loer and Harrison	Thimbleby	Miller	Knight	Jung	Bauersfeld	Chinnapongse	Degani	Rushby	Combéfis	Oliveira
1	Modeling coverage	users	■	■	□	■	■	■	□	■	□	■	■	■	□	□	■	■	□	■	■	■	■	■
		UIs	■	■	■	■	■	■	■	■	■	■	■	■	■	□	■	■	■	■	■	■	■	■
		core	■	■	□	■	■	■	□	■	■	■	■	■	■	■	■	□	□	□	■	■	■	■
2	Plastic UIs		□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	■
3	Kind of properties	usability	■	■	■	■	■	■	□	■	■	■	■	■	■	□	□	□	□	□	□	□	□	■
		functional	■	■	■	□	■	■	■	■	□	□	■	□	■	■	□	□	□	□	□	□	□	■
4	Nuclear-plant domain		□	□	□	□	□	□	□	□	□	□	□	□	□	□	■	□	□	□	□	□	□	■
5	Scalability		□	■	▒	▒	■	■	□	■	▒	▒	■	■	■	■	■	▒	▒	▒	□	■	■	■

■ Yes ▒ No evidence □ No

8.2 Perspectives

Our long-term goal is to propose a complete and integrated environment for the verification of plastic industrial interactive systems. Our contributions so far rely upon a global approach allowing properties to be verified over the system model (which can be applied to industrial interactive systems) and several versions of the system to be compared to each other (which can be applied to plastic interactive system). A complete and integrated verification environment

would provide designers with both possibilities integrated, to better assess the quality of the verified system. For this, the following work is needed in the short, mid, and long term.

8.2.1 Short-term Perspectives

In the short term, a number of improvements identified during the work of this thesis are needed, to take into account aspects partially tackled or not tackled by the proposed approach. So far, our approach allows usability and functional properties of the system to be verified, and a few properties were identified and used as a proof of concept for our approach. Other properties could be identified and formalized, for instance, to cover the *user interface appearance*. The challenge here would be to verify the UI appearance by means of properties expressed in temporal logics, since this formalism is mainly used to express temporal evolutions of the system. Perhaps with the representation of the UI appearance in the ISLTS actions (cf. Section 6.5.1 on page 144), the verification of the UI appearance by means of temporal formulas could also be possible. Furthermore, as similar properties often have the same form, *patterns of properties* specific to interactive systems could be provided, to facilitate the formalization by non-experts.

In its current state, our approach provides three abstraction techniques to support the comparison of plastic interactive systems, namely generalization, omission and elimination. *Other abstraction techniques* could be identified and formalized. A larger range of abstraction techniques would allow *new levels of equivalence* to be demonstrated between two versions of a user interface. For instance, the equivalence between two UIs could be graduated in percentage. The challenge here would be to propose a catalog of abstraction techniques allowing that, and to investigate further possibilities in the formal languages and tools to implement them.

Further *investigation of advanced techniques provided by* CADP to support scalability is needed, such as compositional verification [Garavel *et al.* 2015].

8.2.2 Mid-term Perspectives

In the mid term, a semi-automatic *generation of the formal model* would facilitate the modeling. Such generation could comprise the extraction of information from other models. The main question here is from where the formal model could be extracted. To generate a formal model from another system artifact, such artifact should be non-ambiguous, precise, and complete enough to describe the system user interfaces, the functional core, and the user behavior. A semi-automatic generation could be envisioned, for instance, from CTT (*Concur Task Trees*) task models [Paternò *et al.* 1997]. A skeleton of the formal model could be created, and afterwards manually completed by the designers.

Further investigations could be conducted of the *modeling of users' behaviors*. So far, we cover users actions, and we simplify the modeling by considering that users will always react as expected, and immediately, to discrepancies highlighted by the system. The model could also cover user errors, or a non-immediate reaction from users, for instance. A more complete modeling of the user behavior would enlarge the coverage of the integrated environment.

For the verification of plastic interactive systems, this thesis proposes three possibilities: (i) model checking the different UI versions, (ii) model checking the plasticity engine, and (iii) comparing the UI versions using equivalence checking. Propositions (i) and (ii) were partially explored in this thesis and need deeper investigations. *Model checking the plasticity engine* is particularly challenging. The formalization of numerous transformation rules and

their combination tend to become complex. Nonetheless, a bi-directional approach could be envisioned: a formal model created from an existing plasticity engine, or a plasticity engine of which the code is generated from a proven formal model. Improving propositions (i) and (ii) to verify plasticity would provide designers with three different ways to verify plastic interactive systems. The full formalization of the three approaches would improve the verification of plasticity of our framework, which so far is mainly based on comparison using equivalence checking.

In order to ease the application of our approach to industrial systems, alternatives to the use of a formal model to verify industrial systems could be further investigated. In this thesis, we described three propositions, namely (a) analysis of traces, (b) test case generation, and (c) co-simulation. Similarly to plasticity verification, propositions (b) and (c) were partially explored in this thesis and need deeper investigations. A *formalization of the three approaches* would improve the verification of industrial systems in our framework, which so far is mainly based on model checking of properties and analysis of traces. For instance, for co-simulation, this formalization consists in a language that could serve as bridge between the formal model and industrial systems. At the end of these investigations, designers would be provided with three techniques, and could choose the one that suits better their needs.

Testing and formal methods are complementary ways to assess and increase the quality of interactive systems. *Supporting testing techniques*, such as the generation of test cases from formal models, would enrich our approach, helping the approach to continuously find applications in industrial contexts.

8.2.3 Long-term Perspectives

In the long term, the contributions presented in this thesis can be seen as the starting point for a *full verification environment for interactive systems*. Coupling such a framework with a toolbox (such as CADP) implementing formal verification techniques would benefit from the maturity of the verification tools. Such a framework would provide designers with a catalog of techniques to verify interactive systems (safety-critical or not, plastic or not). For a better integration in industrial contexts, the three propositions (a), (b), and (c) would be proposed. For verifying plastic interactive systems, the propositions (i), (ii) and (iii) would be proposed.

From a research point of view, heuristics are needed to decide which technique is more appropriate to verify a given interactive system. For instance, once a single version of the system is available, model checking of properties would be recommended. For plastic interactive systems, once multiple versions of the system are available, the environment could suggest designers either compare the versions by equivalence checking, or model check each version separately, or compose both techniques.

From a technological point of view, the implementation of such a framework would be a layer above the verification toolbox, and would invoke to tools provided by the toolbox. Such implementation would facilitate the usage of our approach, proving users with a *verification environment* that focus on interactive systems. Such an environment would integrate features implementing short-term and mid-term perspectives, such as the patterns of properties and the semi-automatic generation of the formal models. These gradual enhancements in our approach would *improve its scalability*.

Appendix A
Reactor Parameters

This appendix names some of the reactor parameters we model in the main case study of this thesis: the EDF system described in Chapter 3, page 69.

Table 19: Some reactor parameters of the EDF case study

#	**Acronym**	**Name**
1	Pth moy	Puissance thermique moyenne
2	DPAX moy	Déformé de flux
3	Groupe R	Position des grappes
4	P primaire	Pression primaire
5	T moy	Température moyenne
6	Cb	Concentration en bore
7	N GV1	Niveau de remplissage des GV
8	N GV2	Niveau de remplissage des GV
9	N GV3	Niveau de remplissage des GV
10	N GV4	Niveau de remplissage des GV
11	P CNI	Puissance chaînes neutroniques intermédiaires
12	N ASG	Niveau ASG
13	T ASG	Température ASG
14	N RCV	Borication dilution
15	P condenseur	Pression condenseur
16	P elec	Puissance électrique
17	R1	Position de groupe R1
18	R2	Position de groupe R2
19	G21	G21
20	G22	G22
21	G1	G1
22	RPN010MA	Mesures de la puissance neutronique
23	RPN020MA	Mesures de la puissance neutronique
24	RPN030MA	Mesures de la puissance neutronique
25	RPN040MA	Mesures de la puissance neutronique
26	ParamX	ParamX
27	ParamY	ParamY
28	ParamZ	ParamZ
29	ParamW	ParamW

Appendix B

An Excerpt of a LNT Specification

This appendix contains some Lnt modules from the specification of the EDF case study (Chapter 3, page 69). It is structured into four sections: the first section gives a short introduction of the Lnt specification language; the following ones describe three module we implemented using Lnt.

A Short Introduction of the LNT Specification Language

Lnt is a formal language for describing formal specifications of a system. It is built around the definition of *modules*. Each module can embody one *process* and several *functions*. A process denotes a behavior of the system. It can be parameterized by a list of formal *gates*, a list of formal *variables*, and a list of formal *exceptions*. Communication between processes of different modules is enabled by means of a *rendezvous* mechanism through the gates.

A function is a routine used for code factorization. It can have zero, one, or more arguments and can return a single result [Champelovier *et al.* 2011]. In Lnt, functions can be recursive. Beside, Lnt also supports *polymorphism*. Hence, it is possible to define several functions that have the same name and the same signature but with different argument types. At runtime, polymorphism permits to choose the right function to call according to the argument types.

In this thesis, we use LNT for describing the specification of the EDF case study. Figure 70 on page 97 illustrates the module structure of this specification. In the following, we chose to describe three main modules: the *Reactor* module (`reacteur.lnt`), the *Scenario* module (`scenarios.lnt`), and the *Menu* module (`vue.lnt`).

The Reactor Module

The *Reactor* module implements the behavior of some reactor parameters. Listing B.1 describes this module. It simulates five anomaly scenarios over 29 reactor parameters in an infinite loop. For this, we defined the process `reacteur_p` and several functions. In the process, 29 variables (one for each reactor parameter) are initially declared (lines 5–9), and initialized with their corresponding value using the `initialisation_params`[1] function (lines 19–25).

Each scenario affects one reactor parameter, starting by the parameter `Pth_moy` (line 27). A *loop* (lines 31–179) permits to iterate over all the scenarios and parameters. At each instant of the current scenario, a function called `generer_probleme` (e.g., lines 34, 39) affects one parameter with a new value, and possibly with an anomaly, depending on the scenario. The values of all the 29 parameters are then made available by through the `voir_params_reacteur` gate of the process (lines 180–184). This provides the output of this process.

The function `generer_probleme` (lines 191–232) varies the five scenarios in a cycle, the instants of the current scenario, and the affected reactor parameter. For instance, the scenario

[1] The implementation of this function is not described here.

1, (*threshold overflow* – “dépassement haut”) has seven instants (defined in the function `duree`, line 311); during seven loop iterations, this anomaly scenario will change the value of the current reactor parameter, and once the scenario had finished, another scenario will be chosen, as well as another reactor parameter (lines 220–224). We defined one `generer_probleme` function for each type of reactor parameters (nat, int, and real). Those are polymorphic, i.e., the resolution of the reactor type permits to choose which function should be called at runtime .

```
module reacteur (scenarios) is

process reacteur_p [voir_params_reacteur: TParam] is

  var --29 Params
    Pth_moy_v, DPAXmoy_v, Cb_v, GroupeR_v, NRCV_v, Pcondenseur_v, Pelec_v,
    NGV1_v, NGV2_v, NGV3_v, NGV4_v, TASG_v, NASG_v, R1_v, R2_v, G1_v, G21_v,
    G22_v, RPN010MA_v, RPN020MA_v, RPN030MA_v, RPN040MA_v, ParamX_v, ParamY_v,
    ParamZ_v, ParamW_v, PCNI_v, Pprimaire_v, Tmoy_v: TParam,

    newValue_int:          int,
    newValue_nat:          nat,
    newValue_real:         real,
    defaut_v:              TDefaut,
    variable_a_probleme:   TNomParam,
    scenario:              nat,
    instant:               nat in

    -- initialisations: chaque variable P à sa valeur moyenne
    initialisation_params(?Pth_moy_v, ?DPAXmoy_v, ?Cb_v, ?GroupeR_v, ?NRCV_v,
                ?Pcondenseur_v, ?Pelec_v, ?NGV1_v, ?NGV2_v, ?NGV3_v,
                ?NGV4_v, ?TASG_v, ?NASG_v, ?R1_v, ?R2_v, ?G1_v,
                ?G21_v, ?G22_v, ?RPN010MA_v, ?RPN020MA_v, ?RPN030MA_v,
                ?RPN040MA_v, ?ParamX_v, ?ParamY_v, ?ParamZ_v, ?ParamW_v,
                ?PCNI_v, ?Pprimaire_v, ?Tmoy_v);

    variable_a_probleme := Pth_moy;
    scenario            := 1;
    instant             := 1;

    loop r in
      case variable_a_probleme in
        Pth_moy        ->
            eval newValue_nat:=generer_probleme(min_Pth_moy, max_Pth_moy,
                 moy_Pth_moy, !?variable_a_probleme, !?instant, !?scenario,
                 ?defaut_v);
            Pth_moy_v := TPth_moy (newValue_nat, defaut_v)
        | DPAXmoy      ->
            eval newValue_int:=generer_probleme(min_DPAXmoy, max_DPAXmoy,
                 moy_DPAXmoy, !?variable_a_probleme, !?instant, !?scenario,
                 ?defaut_v);
            DPAXmoy_v := TDPAXmoy (newValue_int, defaut_v)
        | GroupeR      ->
            eval newValue_nat:=generer_probleme(min_GroupeR, max_GroupeR,
                 moy_GroupeR, !?variable_a_probleme, !?instant, !?scenario,
                 ?defaut_v);
            GroupeR_v := TGroupeR (newValue_nat, defaut_v)
        | Pprimaire    ->
            eval newValue_real:=generer_probleme(min_Pprimaire, max_Pprimaire,
                  moy_Pprimaire, !?variable_a_probleme, !?instant,
```

```
51                  !?scenario, ?defaut_v);
52              Pprimaire_v := TPprimaire (newValue_real, defaut_v)
53          | Tmoy        ->
54              eval newValue_real:=generer_probleme(min_Tmoy, max_Tmoy,
55                   moy_Tmoy, !?variable_a_probleme, !?instant, !?scenario,
56                  ?defaut_v);
57              Tmoy_v := TTmoy (newValue_real, defaut_v)
58          | Cb          ->
59              eval newValue_nat:=generer_probleme(min_Cb, max_Cb,
60                  moy_Cb, !?variable_a_probleme, !?instant, !?scenario,
61                  ?defaut_v);
62              Cb_v := TCb (newValue_nat, defaut_v)
63          | NGV1        ->
64              eval newValue_int:=generer_probleme(min_NGV1, max_NGV1,
65                  moy_NGV1, !?variable_a_probleme, !?instant, !?scenario,
66                  ?defaut_v);
67              NGV1_v := TNGV1 (newValue_int, defaut_v)
68          | NGV2        ->
69              eval newValue_int:=generer_probleme(min_NGV2, max_NGV2,
70                  moy_NGV2, !?variable_a_probleme, !?instant, !?scenario,
71                  ?defaut_v);
72              NGV2_v := TNGV2 (newValue_int, defaut_v)
73          | NGV3        ->
74              eval newValue_int:=generer_probleme(min_NGV3, max_NGV3,
75                  moy_NGV3, !?variable_a_probleme, !?instant, !?scenario,
76                  ?defaut_v);
77              NGV3_v := TNGV3 (newValue_int, defaut_v)
78          | NGV4        ->
79              eval newValue_int:=generer_probleme(min_NGV4, max_NGV4,
80                  moy_NGV4, !?variable_a_probleme, !?instant, !?scenario,
81                  ?defaut_v);
82              NGV4_v := TNGV4 (newValue_int, defaut_v)
83          | PCNI        ->
84              eval newValue_real:=generer_probleme(min_PCNI, max_PCNI,
85                   moy_PCNI, !?variable_a_probleme, !?instant, !?scenario,
86                  ?defaut_v);
87              PCNI_v := TPCNI (newValue_real, defaut_v)
88          | NASG        ->
89              eval newValue_nat:=generer_probleme(min_NASG, max_NASG,
90                  moy_NASG, !?variable_a_probleme, !?instant, !?scenario,
91                  ?defaut_v);
92              NASG_v := TNASG (newValue_nat, defaut_v)
93          | TASG        ->
94              eval newValue_nat:=generer_probleme(min_TASG, max_TASG,
95                  moy_TASG, !?variable_a_probleme, !?instant, !?scenario,
96                  ?defaut_v);
97              TASG_v := TTASG (newValue_nat, defaut_v)
98          | NRCV        ->
99              eval newValue_nat:=generer_probleme(min_NRCV, max_NRCV,
100                 moy_NRCV, !?variable_a_probleme, !?instant, !?scenario,
101                 ?defaut_v);
102             NRCV_v := TNRCV (newValue_nat, defaut_v)
103         | Pcondenseur ->
104             eval newValue_nat:=generer_probleme(min_Pcondenseur,
105                 max_Pcondenseur, moy_Pcondenseur, !?variable_a_probleme,
106                 !?instant, !?scenario, ?defaut_v);
107             Pcondenseur_v := TPcondenseur (newValue_nat, defaut_v)
108         | Pelec       ->
```

```
109                 eval newValue_nat:=generer_probleme(min_Pelec, max_Pelec,
110                      moy_Pelec, !?variable_a_probleme, !?instant, !?scenario,
111                      ?defaut_v);
112                 Pelec_v := TPelec (newValue_nat, defaut_v)
113             | R1          ->
114                 eval newValue_nat:=generer_probleme(min_R1, max_R1,
115                      moy_R1, !?variable_a_probleme, !?instant, !?scenario,
116                      ?defaut_v);
117                 R1_v := TR1 (newValue_nat, defaut_v)
118             | R2          ->
119                 eval newValue_nat:=generer_probleme(min_R2, max_R2,
120                      moy_R2, !?variable_a_probleme, !?instant, !?scenario,
121                      ?defaut_v);
122                 R2_v := TR2 (newValue_nat, defaut_v)
123             | G21         ->
124                 eval newValue_nat:=generer_probleme(min_G21, max_G21,
125                      moy_G21, !?variable_a_probleme, !?instant, !?scenario,
126                      ?defaut_v);
127                 G21_v := TG21 (newValue_nat, defaut_v)
128             | G22         ->
129                 eval newValue_nat:=generer_probleme(min_G22, max_G22,
130                      moy_G22, !?variable_a_probleme, !?instant, !?scenario,
131                      ?defaut_v);
132                 G22_v := TG22 (newValue_nat, defaut_v)
133             | G1          ->
134                 eval newValue_nat:=generer_probleme(min_G1, max_G1,
135                      moy_G1, !?variable_a_probleme, !?instant, !?scenario,
136                      ?defaut_v);
137                 G1_v := TG1 (newValue_nat, defaut_v)
138             | RPN010MA    ->
139                 eval newValue_nat:=generer_probleme(min_RPN010MA, max_RPN010MA,
140                      moy_RPN010MA, !?variable_a_probleme, !?instant, !?scenario,
141                      ?defaut_v);
142                 RPN010MA_v := TRPN010MA (newValue_nat, defaut_v)
143             | RPN020MA    ->
144                 eval newValue_nat:=generer_probleme(min_RPN020MA, max_RPN020MA,
145                      moy_RPN020MA, !?variable_a_probleme, !?instant, !?scenario,
146                      ?defaut_v);
147                 RPN020MA_v := TRPN020MA (newValue_nat, defaut_v)
148             | RPN030MA    ->
149                 eval newValue_nat:=generer_probleme(min_RPN030MA, max_RPN030MA,
150                      moy_RPN030MA, !?variable_a_probleme, !?instant, !?scenario,
151                      ?defaut_v);
152                 RPN030MA_v := TRPN030MA (newValue_nat, defaut_v)
153             | RPN040MA    ->
154                 eval newValue_nat:=generer_probleme(min_RPN040MA, max_RPN040MA,
155                      moy_RPN040MA, !?variable_a_probleme, !?instant, !?scenario,
156                      ?defaut_v);
157                 RPN040MA_v := TRPN040MA (newValue_nat, defaut_v)
158             | ParamX      ->
159                 eval newValue_nat:=generer_probleme(min_ParamX, max_ParamX,
160                      moy_ParamX, !?variable_a_probleme, !?instant, !?scenario,
161                      ?defaut_v);
162                 ParamX_v := TParamX (newValue_nat, defaut_v)
163             | ParamY      ->
164                 eval newValue_nat:=generer_probleme(min_ParamY, max_ParamY,
165                      moy_ParamY, !?variable_a_probleme, !?instant, !?scenario,
166                      ?defaut_v);
```

```
167             ParamY_v := TParamY (newValue_nat, defaut_v)
168         | ParamZ       ->
169             eval newValue_nat:=generer_probleme(min_ParamZ, max_ParamZ,
170                  moy_ParamZ, !?variable_a_probleme, !?instant, !?scenario,
171                  ?defaut_v);
172             ParamZ_v := TParamZ (newValue_nat, defaut_v)
173         | ParamW       ->
174             eval newValue_nat:=generer_probleme(min_ParamW, max_ParamW,
175                  moy_ParamW, !?variable_a_probleme, !?instant, !?scenario,
176                  ?defaut_v);
177             ParamW_v := TParamW (newValue_nat, defaut_v)
178         | nil -> stop
179       end case;
180       voir_params_reacteur(Pth_moy_v, DPAXmoy_v, GroupeR_v, Pprimaire_v, Tmoy_v,
181               Cb_v, NGV1_v, NGV2_v, NGV3_v, NGV4_v, PCNI_v, NASG_v,
182               TASG_v, NRCV_v, Pcondenseur_v, Pelec_v, R1_v, R2_v,
183               G21_v, G22_v, G1_v, RPN010MA_v, RPN020MA_v, RPN030MA_v,
184               RPN040MA_v, ParamX_v, ParamY_v, ParamZ_v, ParamW_v)
185
186     end loop
187   end var
188 end process
189
190 ------------------------------------------------------------------------------
191 function generer_probleme (in min_p: nat,
192                            in max_p: nat,
193                            in moy_p: nat,
194                            in out variable_a_probleme: TNomParam,
195                            in out instant: nat,
196                            in out scenario: nat,
197                            out defaut: TDefaut):nat is
198
199   var newvalue_nat: nat, a_defaut: bool, scen_aux:nat in
200
201     a_defaut := true;
202     scen_aux := scenario; --backup de la valeur actuelle du scénario
203
204     if (min_p == 0) and (scenario == 2) then
205       scenario := 1
206     end if;
207
208     case scenario in
209       1 -> newValue_nat :=scenario1(instant, max_p, moy_p, a_defaut, ?defaut)
210     | 2 -> newValue_nat :=scenario2(instant, min_p, moy_p, a_defaut, ?defaut)
211     | 3 -> newValue_nat :=scenario3(instant, min_p,max_p,moy_p,a_defaut,?defaut)
212     | 4 -> newValue_nat :=scenario4(instant, max_p, moy_p, a_defaut, ?defaut)
213     | 5 -> newValue_nat :=scenario5(instant, max_p, moy_p, a_defaut, ?defaut)
214     | any  ->
215             newValue_nat := 0;
216             defaut       := nil
217     end case;
218
219     scenario := scen_aux;
220     if instant == duree (scenario) then
221       -- on a dépassé la durée du scénario
222       scenario := successeur (scenario);
223       instant  := 1;
224       variable_a_probleme := next_variable_a_probleme(variable_a_probleme)
```

```
225     else
226       instant := instant + 1
227     end if;
228
229     return newValue_nat
230
231   end var
232 end function
233
234 ------------------------------------------------------------------------------
235 function generer_probleme (in min_p: int,
236                            in max_p: int,
237                            in moy_p: int,
238                            in out variable_a_probleme: TNomParam,
239                            in out instant: nat,
240                            in out scenario: nat,
241                            out defaut: TDefaut):int is
242
243   var newvalue_int: int, a_defaut: bool in
244
245     a_defaut:=true;
246     case scenario in
247       1 -> newValue_int :=scenario1(instant, max_p, moy_p, a_defaut, ?defaut)
248     | 2 -> newValue_int :=scenario2(instant, min_p, moy_p, a_defaut, ?defaut)
249     | 3 -> newValue_int :=scenario3(instant, min_p,max_p,moy_p,a_defaut,?defaut)
250     | 4 -> newValue_int :=scenario4(instant, max_p, moy_p, a_defaut, ?defaut)
251     | 5 -> newValue_int :=scenario5(instant, max_p, moy_p, a_defaut, ?defaut)
252     | any  ->
253             newValue_int := 0;
254             defaut       := nil
255     end case;
256
257     if instant == duree (scenario) then
258       -- on a dépassé la durée du scénario
259       scenario := successeur (scenario);
260       instant  := 1;
261       variable_a_probleme := next_variable_a_probleme(variable_a_probleme)
262     else
263       instant := instant + 1
264     end if;
265
266     return newValue_int
267
268   end var
269 end function
270
271 ------------------------------------------------------------------------------
272 function generer_probleme (in min_p: real,
273                            in max_p: real,
274                            in moy_p: real,
275                            in out variable_a_probleme: TNomParam,
276                            in out instant: nat,
277                            in out scenario: nat,
278                            out defaut: TDefaut):real is
279
280   var newvalue_real: real, a_defaut: bool in
281
282     a_defaut:=true;
```

```
283     case scenario in
284       1 ->newValue_real :=scenario1(instant, max_p, moy_p, a_defaut, ?defaut)
285     | 2 ->newValue_real :=scenario2(instant, min_p, moy_p, a_defaut, ?defaut)
286     | 3 ->newValue_real :=scenario3(instant, min_p,max_p,moy_p,a_defaut,?defaut)
287     | 4 ->newValue_real :=scenario4(instant, max_p, moy_p, a_defaut, ?defaut)
288     | 5 ->newValue_real :=scenario5(instant, max_p, moy_p, a_defaut, ?defaut)
289     | any  ->
290           newValue_real := 0.0;
291           defaut        := nil
292     end case;
293
294     if instant == duree (scenario) then
295       -- on a dépassé la durée du scénario
296       scenario := successeur (scenario);
297       instant  := 1;
298       variable_a_probleme := next_variable_a_probleme(variable_a_probleme)
299     else
300       instant := instant + 1
301     end if;
302
303     return newValue_real
304
305   end var
306 end function
307
308 ----------------------------------------------------------------------------
309 function duree (scenario:nat):nat is
310   case scenario in
311       1   -> return 7
312     | 2   -> return 7
313     | 3   -> return 8
314     | 4   -> return 7
315     | 5   -> return 8
316     | any -> return 0
317   end case
318 end function
319
320 ----------------------------------------------------------------------------
321 function successeur (S:nat):nat is
322   if S == 5 then
323     return 1
324   else
325     return S+1
326   end if
327 end function
328
329 ----------------------------------------------------------------------------
330 function next_variable_a_probleme (V:TNomParam):TNomParam is
331        case V in
332             Pth_moy      -> return DPAXmoy
333           | DPAXmoy      -> return GroupeR
334           | GroupeR      -> return Pprimaire
335           | Pprimaire    -> return Tmoy
336           | Tmoy         -> return Cb
337           | Cb           -> return NGV1
338           | NGV1         -> return NGV2
339           | NGV2         -> return NGV3
340           | NGV3         -> return NGV4
```

```
341             | NGV4          -> return PCNI
342             | PCNI          -> return NASG
343             | NASG          -> return TASG
344             | TASG          -> return NRCV
345             | NRCV          -> return Pcondenseur
346             | Pcondenseur -> return Pelec
347             | Pelec         -> return R1
348             | R1            -> return R2
349             | R2            -> return G21
350             | G21           -> return G22
351             | G22           -> return G1
352             | G1            -> return RPN010MA
353             | RPN010MA      -> return RPN020MA
354             | RPN020MA      -> return RPN030MA
355             | RPN030MA      -> return RPN040MA
356             | RPN040MA      -> return ParamX
357             | ParamX        -> return ParamY
358             | ParamY        -> return ParamZ
359             | ParamZ        -> return ParamW
360             | ParamW        -> return Pth_moy
361             | any           -> return Pth_moy
362         end case
363 end function
364
365 end module
```

Listing B.1: Reactor.lnt ("reacteur.lnt")

The Scenarios Module

The *Scenarios* module describes the five anomaly scenarios simulated in the EDF case study: *threshold overflow*, *threshold underflow*, *gradient excess*, *invalid measurement*, and *loss of redundancy*. This module is illustrated in listing B.2. It does not contain any process. Instead, it defines several functions for each scenario. Precisely, fifteen functions are defined: three functions for each of the five anomaly scenarios, covering the three reactor parameter types (nat, int, and real).

Each scenario is composed of a given number of instants. For instance, the scenario 1 has seven instants (lines 8–19). At each instant, the function generates a value according to formulas defined in the scenario, and depends on some attributes of the reactor parameter affected by the scenario, such as its maximum threshold (e.g., line 9). Anomalies on the reactor parameter can be generated at a given instant of the scenario, for instance, once the generated value exceeds the parameter maximum threshold (lines 11–16).

```
1  module scenarios(bibliotheque) is
2
3  -- scenario1 = dépassement haut, type = int
4  function scenario1(in instant:nat, in max_p:int, in moy_p:int, in a_defaut:bool,
5                     out defaut:TDefaut):int is
6    defaut := nil;
7    case instant in
8        1   -> return moy_p
9      | 2   -> return div_tmp ((max_p+moy_p), 2)
10     | 3   -> return max_p
```

```
11      | 4   -> if a_defaut then
12                defaut := depassementHaut;
13                return max_p + 1
14              else
15                return max_p
16              end if
17      | 5   -> return max_p
18      | 6   -> return div_tmp ((max_p+moy_p), 2)
19      | 7   -> return moy_p
20      | any -> return 0
21    end case
22  end function
23
24  -- scenario1 = dépassement haut, type = nat
25  function scenario1(in instant:nat, in max_p:nat, in moy_p:nat, in a_defaut:bool,
26                     out defaut:TDefaut):nat is
27    defaut := nil;
28    case instant in
29        1   -> return moy_p
30      | 2   -> return ((max_p+moy_p) div 2)
31      | 3   -> return max_p
32      | 4   ->  if a_defaut then
33                defaut := depassementHaut;
34                return max_p + 1
35              else
36                return max_p
37              end if
38      | 5   -> return max_p
39      | 6   -> return ((max_p+moy_p) div 2)
40      | 7   -> return moy_p
41      | any -> return 0
42    end case
43  end function
44
45  -- scenario1 = dépassement haut, type = real
46  function scenario1(in instant:nat,in max_p:real,in moy_p:real,in a_defaut:bool,
47                     out defaut:TDefaut):real is
48    defaut := nil;
49    case instant in
50        1   -> return moy_p
51      | 2   -> return ((max_p+moy_p) / 2.0)
52      | 3   -> return max_p
53      | 4   -> if a_defaut then
54                defaut := depassementHaut;
55                return max_p + 1.0
56              else
57                return max_p
58              end if
59      | 5   -> return max_p
60      | 6   -> return ((max_p+moy_p) / 2.0)
61      | 7   -> return moy_p
62      | any -> return 0.0
63    end case
64  end function
65
66  ------------------------------------------------------------------------------
67  -- scenario2 = dépassement bas, type = int
68  function scenario2(in instant:nat, in min_p:int, in moy_p:int, in a_defaut:bool,
```

```
                    out defaut:TDefaut):int is
  defaut := nil;
  case instant in
      1   -> return moy_p
    | 2   -> return div_tmp ((min_p+moy_p), 2)
    | 3   -> return min_p
    | 4   -> if a_defaut then
               defaut := depassementBas;
               return min_p - 1
             else
               return min_p
             end if
    | 5   -> return min_p
    | 6   -> return div_tmp ((min_p+moy_p), 2)
    | 7   -> return moy_p
    | any -> return 0
  end case
end function

-- scenario2 = dépassement bas, type = nat
function scenario2(in instant:nat, in min_p:nat, in moy_p:nat, in a_defaut:bool,
                    out defaut:TDefaut):nat is
  defaut := nil;
  case instant in
      1   -> return moy_p
    | 2   -> return ((min_p+moy_p) div 2)
    | 3   -> return min_p
    | 4   -> if a_defaut then
               defaut := depassementBas;
               return min_p - 1
             else
               return min_p
             end if
    | 5   -> return min_p
    | 6   -> return ((min_p+moy_p) div 2)
    | 7   -> return moy_p
    | any -> return 0
  end case
end function

-- scenario2 = dépassement bas, type = real
function scenario2(in instant:nat,in min_p:real,in moy_p:real,in a_defaut:bool,
                    out defaut:TDefaut):real is
  defaut := nil;
  case instant in
      1   -> return moy_p
    | 2   -> return ((min_p+moy_p) / 2.0)
    | 3   -> return min_p
    | 4   -> if a_defaut then
               defaut := depassementBas;
               return min_p - 1.0
             else
               return min_p
             end if
    | 5   -> return min_p
    | 6   -> return ((min_p+moy_p) / 2.0)
    | 7   -> return moy_p
    | any -> return 0.0
```

```
127   end case
128 end function
129
130 ------------------------------------------------------------------------------
131 -- scenario3 = dépassement gradient, type = int
132 function scenario3(in instant:nat, in min_p:int, in max_p:int, in moy_p:int,
133                    in a_defaut:bool, out defaut:TDefaut):int is
134   defaut := nil;
135   case instant in
136       1   -> return moy_p
137     | 2   -> return moy_p - div_tmp ((moy_p-min_p), 6)
138     | 3   -> return moy_p - div_tmp ((moy_p-min_p), 4)
139     | 4   -> return moy_p - div_tmp ((moy_p-min_p), 2)
140     | 5   -> return min_p
141     | 6   -> if a_defaut then
142                 defaut := depassementGradient;
143                 return max_p
144               else
145                 return min_p
146               end if
147     | 7   -> return moy_p + div_tmp ((max_p-moy_p), 2)
148     | 8   -> return moy_p
149     | any -> return 0
150   end case
151 end function
152
153 -- scenario3 = dépassement gradient, type = nat
154 function scenario3(in instant:nat, in min_p:nat, in max_p:nat, in moy_p:nat,
155                    in a_defaut:bool, out defaut:TDefaut):nat is
156   defaut := nil;
157   case instant in
158       1   -> return moy_p
159     | 2   -> return moy_p - ((moy_p-min_p) div 6)
160     | 3   -> return moy_p - ((moy_p-min_p) div 4)
161     | 4   -> return moy_p - ((moy_p-min_p) div 2)
162     | 5   -> return min_p
163     | 6   -> if a_defaut then
164                 defaut := depassementGradient;
165                 return max_p
166               else
167                 return min_p
168               end if
169     | 7   -> return moy_p + ((max_p-moy_p) div 2)
170     | 8   -> return moy_p
171     | any -> return 0
172   end case
173 end function
174
175 -- scenario3 = dépassement gradient, type = real
176 function scenario3(in instant:nat, in min_p:real, in max_p:real, in moy_p:real,
177                    in a_defaut:bool, out defaut:TDefaut):real is
178   defaut := nil;
179   case instant in
180       1   -> return moy_p
181     | 2   -> return moy_p - ((moy_p-min_p) / 6.0)
182     | 3   -> return moy_p - ((moy_p-min_p) / 4.0)
183     | 4   -> return moy_p - ((moy_p-min_p) / 2.0)
184     | 5   -> return min_p
```

```
    | 6   -> if a_defaut then
               defaut := depassementGradient;
               return max_p
             else
               return min_p
             end if
    | 7   -> return moy_p + ((max_p-moy_p) / 2.0)
    | 8   -> return moy_p
    | any -> return 0.0
  end case
end function

-------------------------------------------------------------------------------
-- scenario4 = perte de redondance, type = int
function scenario4(in instant:nat, in max_p:int, in moy_p:int, in a_defaut:bool,
                   out defaut:TDefaut):int is
  defaut := nil;
  case instant in
      1   -> return moy_p
    | 2   -> return moy_p + div_tmp ((max_p-moy_p) , 6)
    | 3   -> return moy_p + div_tmp ((max_p-moy_p) , 4)
    | 4   -> return moy_p + div_tmp ((max_p-moy_p) , 2)
    | 5   -> return max_p
    | 6   -> if a_defaut then
               defaut := perteRedondance;
               return moy_p
             else
               return max_p
             end if
    | 7   -> return moy_p
    | any -> return 0
  end case
end function

-- scenario4 = perte de redondance, type = nat
function scenario4(in instant:nat, in max_p:nat, in moy_p:nat, in a_defaut:bool,
                   out defaut:TDefaut):nat is
  defaut := nil;
  case instant in
      1   -> return moy_p
    | 2   -> return moy_p + ((max_p-moy_p) div 6)
    | 3   -> return moy_p + ((max_p-moy_p) div 4)
    | 4   -> return moy_p + ((max_p-moy_p) div 2)
    | 5   -> return max_p
    | 6   -> if a_defaut then
               defaut := perteRedondance;
               return moy_p
             else
               return max_p
             end if
    | 7   -> return moy_p
    | any -> return 0
  end case
end function

-- scenario4 = perte de redondance, type = real
function scenario4(in instant:nat,in max_p:real,in moy_p:real,in a_defaut:bool,
                   out defaut:TDefaut):real is
```

```
243   defaut := nil;
244   case instant in
245       1   -> return moy_p
246     | 2   -> return moy_p + ((max_p-moy_p) / 6.0)
247     | 3   -> return moy_p + ((max_p-moy_p) / 4.0)
248     | 4   -> return moy_p + ((max_p-moy_p) / 2.0)
249     | 5   -> return max_p
250     | 6   -> if a_defaut then
251                 defaut := perteRedondance;
252                 return moy_p
253               else
254                 return max_p
255               end if
256     | 7   -> return moy_p
257     | any -> return 0.0
258   end case
259 end function
260
261 ------------------------------------------------------------------------------
262 -- scenario5 = invalidité de mesure, type = int
263 function scenario5(in instant:nat, in max_p:int, in moy_p:int, in a_defaut:bool,
264                    out defaut:TDefaut):int is
265   defaut := nil;
266   case instant in
267       1   -> return moy_p
268     | 2   -> return moy_p + div_tmp ((max_p-moy_p), 4)
269     | 3   -> return moy_p + div_tmp ((max_p-moy_p), 2)
270     | 4   -> return moy_p + div_tmp ((max_p-moy_p), 4)
271     | 5   -> return moy_p
272     | 6   -> return moy_p - div_tmp ((max_p-moy_p), 4)
273     | 7   -> if a_defaut then
274                 defaut := invaliditeMesure;
275                 return 0
276               else
277                 return moy_p - div_tmp ((max_p-moy_p), 4)
278               end if
279     | 8   -> return moy_p
280     | any -> return 0
281   end case
282 end function
283
284 -- scenario5 = invalidité de mesure, type = nat
285 function scenario5(in instant:nat, in max_p:nat, in moy_p:nat, in a_defaut:bool,
286                    out defaut:TDefaut):nat is
287   defaut := nil;
288   case instant in
289       1   -> return moy_p
290     | 2   -> return moy_p + ((max_p-moy_p) div 4)
291     | 3   -> return moy_p + ((max_p-moy_p) div 2)
292     | 4   -> return moy_p + ((max_p-moy_p) div 4)
293     | 5   -> return moy_p
294     | 6   -> return moy_p - ((max_p-moy_p) div 4)
295     | 7   -> if a_defaut then
296                 defaut := invaliditeMesure;
297                 return 0
298               else
299                 return moy_p - ((max_p-moy_p) div 4)
300               end if
```

```
301     | 8   -> return moy_p
302     | any -> return 0
303   end case
304 end function
305
306 -- scenario5 = invalidité de mesure, type = real
307 function scenario5(in instant:nat,in max_p:real,in moy_p:real,in a_defaut:bool,
308                    out defaut:TDefaut):real is
309   defaut := nil;
310   case instant in
311       1   -> return moy_p
312     | 2   -> return moy_p + ((max_p-moy_p) / 4.0)
313     | 3   -> return moy_p + ((max_p-moy_p) / 2.0)
314     | 4   -> return moy_p + ((max_p-moy_p) / 4.0)
315     | 5   -> return moy_p
316     | 6   -> return moy_p - ((max_p-moy_p) / 4.0)
317     | 7   -> if a_defaut then
318                 defaut := invaliditeMesure;
319                 return 0.0
320               else
321                 return moy_p - ((max_p-moy_p) / 4.0)
322               end if
323     | 8   -> return moy_p
324     | any -> return 0.0
325   end case
326 end function
327
328 end module
```

Listing B.2: Scenarios.lnt ("scenarios.lnt")

The Menu Module

The *Menu* module describes the hierarchical menu (cf. Section 5.2.2-*b)* on page 98) of the user interfaces defined in the EDF case study. This module is illustrated in Listing B.3. It is composed of a unique process (line 3) and a unique function (line 243). The `vue_p` process is parameterized with several gates (lines 3–47), allowing the *User* module (not described here) to communicate with this module (cf. Figure 70 on page 97). Each gate denotes a menu option. Communication between both the *User* and the *Menu* modules permits to simulate the selection by the user of a menu option on the user interface.

The `visible` function (lines 243–255) implements the behavior of the menu, regarding the menu options available to the user according to his or her last choice. This function takes two parameters: the *menu option* to test, and the *last option menu* chosen. It returns a *boolean* value that indicates whether the menu option should be made available to the user, according to the last option menu that was chosen. This function is recursive and permits to pull parent menus up. Hence, it supports single-level menus as well as hierarchical ones.

```
1 module vue(bibliotheque) is
2
3 process vue_p[vue_synthese_globale,
4               vue_surete,
5               vue_surete_reactivite,
6               vue_surete_reactivite_concentbore,
```

```
7                 vue_surete_reactivite_posgrappes,
8                 vue_surete_reactivite_bordilution,
9                 vue_surete_reactivite_pilreacteur,
10                vue_surete_refroidissement,
11                vue_surete_refroidissement_inv,
12                vue_surete_refroidissement_circ,
13                vue_surete_refroidissement_app,
14                vue_surete_refroidissement_sfroid,
15                vue_surete_confinement,
16                vue_production,
17                vue_production_R,
18                vue_production_R_RRI,
19                vue_production_R_RRA,
20                vue_production_R_RCP,
21                vue_production_R_RPN,
22                vue_production_R_RGL,
23                vue_production_R_RCV,
24                vue_production_R_REA,
25                vue_production_R_RPE,
26                vue_production_A,
27                vue_production_V,
28                vue_production_G,
29                vue_production_C,
30                vue_support,
31                vue_support_electricite,
32                vue_support_electricite_LGA,
33                vue_support_electricite_LGB,
34                vue_support_electricite_LGD,
35                vue_support_electricite_LGC,
36                vue_support_electricite_LHA,
37                vue_support_electricite_LHB,
38                vue_support_electricite_LBA,
39                vue_support_aircomprime,
40                vue_support_ic,
41                vue_support_incendie,
42                vue_support_climventilation,
43                vue_signaux,
44                vue_signaux_alertesalarmes,
45                vue_signaux_nonconformites,
46                vue_signaux_discordances,
47                vue_signaux_inhibes: None,
48                voir_vue: TVue] is
49
50  var vue_v, derniere_vue: TVue in
51
52    vue_v := synthese_globale;
53
54    loop v in
55      select
56        derniere_vue := vue_v;
57        vue_v := synthese_globale;
58        vue_synthese_globale where visible(vue_v, derniere_vue)
59        []
60        derniere_vue := vue_v;
61        vue_v := surete;
62        vue_surete where visible(vue_v, derniere_vue)
63          []
64          derniere_vue := vue_v;
```

```
65          vue_v := surete_reactivite;
66          vue_surete_reactivite where visible(vue_v, derniere_vue)
67            []
68            derniere_vue := vue_v;
69            vue_v := surete_reactivite_concentrationbore;
70            vue_surete_reactivite_concentbore where visible(vue_v, derniere_vue)
71            []
72            derniere_vue := vue_v;
73            vue_v := surete_reactivite_positiongrappes;
74            vue_surete_reactivite_posgrappes where visible(vue_v, derniere_vue)
75            []
76            derniere_vue := vue_v;
77            vue_v := surete_reactivite_boricationdilution;
78            vue_surete_reactivite_bordilution where visible(vue_v, derniere_vue)
79            []
80            derniere_vue := vue_v;
81            vue_v := surete_reactivite_pilotagereacteur;
82            vue_surete_reactivite_pilreacteur where visible(vue_v, derniere_vue)
83          []
84          derniere_vue := vue_v;
85          vue_v := surete_refroidissement;
86          vue_surete_refroidissement where visible(vue_v, derniere_vue)
87            []
88            derniere_vue := vue_v;
89            vue_v := surete_refroidissement_inventaire;
90            vue_surete_refroidissement_inv where visible(vue_v, derniere_vue)
91            []
92            derniere_vue := vue_v;
93            vue_v := surete_refroidissement_circulation;
94            vue_surete_refroidissement_circ where visible(vue_v, derniere_vue)
95            []
96            derniere_vue := vue_v;
97            vue_v := surete_refroidissement_appoint;
98            vue_surete_refroidissement_app where visible(vue_v, derniere_vue)
99            []
100           derniere_vue := vue_v;
101           vue_v := surete_refroidissement_sourcesfroides;
102           vue_surete_refroidissement_sfroid where visible(vue_v, derniere_vue)
103         []
104         derniere_vue := vue_v;
105         vue_v := surete_confinement;
106         vue_surete_confinement where visible(vue_v, derniere_vue)
107       []
108       derniere_vue := vue_v;
109       vue_v := production;
110       vue_production where visible(vue_v, derniere_vue)
111         []
112         derniere_vue := vue_v;
113         vue_v := production_R;
114         vue_production_R where visible(vue_v, derniere_vue)
115           []
116           derniere_vue := vue_v;
117           vue_v := production_R_RRI;
118           vue_production_R_RRI where visible(vue_v, derniere_vue)
119           []
120           derniere_vue := vue_v;
121           vue_v := production_R_RRA;
122           vue_production_R_RRA where visible(vue_v, derniere_vue)
```

```
123             []
124             derniere_vue := vue_v;
125             vue_v := production_R_RCP;
126             vue_production_R_RCP where visible(vue_v, derniere_vue)
127             []
128             derniere_vue := vue_v;
129             vue_v := production_R_RPN;
130             vue_production_R_RPN where visible(vue_v, derniere_vue)
131             []
132             derniere_vue := vue_v;
133             vue_v := production_R_RGL;
134             vue_production_R_RGL where visible(vue_v, derniere_vue)
135             []
136             derniere_vue := vue_v;
137             vue_v := production_R_RCV;
138             vue_production_R_RCV where visible(vue_v, derniere_vue)
139             []
140             derniere_vue := vue_v;
141             vue_v := production_R_REA;
142             vue_production_R_REA where visible(vue_v, derniere_vue)
143             []
144             derniere_vue := vue_v;
145             vue_v := production_R_RPE;
146             vue_production_R_RPE where visible(vue_v, derniere_vue)
147           []
148           derniere_vue := vue_v;
149           vue_v := production_A;
150           vue_production_A where visible(vue_v, derniere_vue)
151           []
152           derniere_vue := vue_v;
153           vue_v := production_V;
154           vue_production_V where visible(vue_v, derniere_vue)
155           []
156           derniere_vue := vue_v;
157           vue_v := production_G;
158           vue_production_G where visible(vue_v, derniere_vue)
159           []
160           derniere_vue := vue_v;
161           vue_v := production_C;
162           vue_production_C where visible(vue_v, derniere_vue)
163         []
164         derniere_vue := vue_v;
165         vue_v := support;
166         vue_support where visible(vue_v, derniere_vue)
167           []
168           derniere_vue := vue_v;
169           vue_v := support_electricite;
170           vue_support_electricite where visible(vue_v, derniere_vue)
171             []
172             derniere_vue := vue_v;
173             vue_v := support_electricite_LGA;
174             vue_support_electricite_LGA where visible(vue_v, derniere_vue)
175             []
176             derniere_vue := vue_v;
177             vue_v := support_electricite_LGB;
178             vue_support_electricite_LGB where visible(vue_v, derniere_vue)
179             []
180             derniere_vue := vue_v;
```

```
181             vue_v := support_electricite_LGD;
182             vue_support_electricite_LGD where visible(vue_v, derniere_vue)
183             []
184             derniere_vue := vue_v;
185             vue_v := support_electricite_LGC;
186             vue_support_electricite_LGC where visible(vue_v, derniere_vue)
187             []
188             derniere_vue := vue_v;
189             vue_v := support_electricite_LHA;
190             vue_support_electricite_LHA where visible(vue_v, derniere_vue)
191             []
192             derniere_vue := vue_v;
193             vue_v := support_electricite_LHB;
194             vue_support_electricite_LHB where visible(vue_v, derniere_vue)
195             []
196             derniere_vue := vue_v;
197             vue_v := support_electricite_LBA;
198             vue_support_electricite_LBA where visible(vue_v, derniere_vue)
199           []
200           derniere_vue := vue_v;
201           vue_v := support_aircomprime;
202           vue_support_aircomprime where visible(vue_v, derniere_vue)
203           []
204           derniere_vue := vue_v;
205           vue_v := support_ic;
206           vue_support_ic where visible(vue_v, derniere_vue)
207           []
208           derniere_vue := vue_v;
209           vue_v := support_incendie;
210           vue_support_incendie where visible(vue_v, derniere_vue)
211           []
212           derniere_vue := vue_v;
213           vue_v := support_climventilation;
214           vue_support_climventilation where visible(vue_v, derniere_vue)
215         []
216         derniere_vue := vue_v;
217         vue_v := signaux;
218         vue_signaux where visible(vue_v, derniere_vue)
219           []
220           derniere_vue := vue_v;
221           vue_v := signaux_alertesalarmes;
222           vue_signaux_alertesalarmes where visible(vue_v, derniere_vue)
223           []
224           derniere_vue := vue_v;
225           vue_v := signaux_nonconformites;
226           vue_signaux_nonconformites where visible(vue_v, derniere_vue)
227           []
228           derniere_vue := vue_v;
229           vue_v := signaux_discordances;
230           vue_signaux_discordances where visible(vue_v, derniere_vue)
231           []
232           derniere_vue := vue_v;
233           vue_v := signaux_inhibes;
234           vue_signaux_inhibes where visible(vue_v, derniere_vue)
235         []
236         voir_vue(vue_v)
237       end select
238     end loop
```

```
239   end var
240 end process
241
242
243 function visible(option_menu, derniere_option_choisie:TVue):bool is
244   if (option_menu == nil) then
245     return false
246   ---- affiche les fils de l option ----
247   elsif (parent(option_menu) == derniere_option_choisie) then
248       return true
249   ---- affiche toutes les générations de parents -----
250   elsif (derniere_option_choisie != nil) then
251     return visible(option_menu, parent(derniere_option_choisie))
252   else
253     return false
254   end if
255 end function
256
257 end module
```

Listing B.3: Menu.lnt (“vue.lnt”)

Appendix C
MCL Properties

This appendix describes the nine properties identified in the EDF case study in Section 5.2.8 (page 106), what they aim to verify, and how they are formalized in the MCL language. A summary of the properties can be found in Table 7.

1. "From any UI, one can always go directly to the main UI (i.e., without passing through any other UI).":

   $[\ true^*\ ]$
   $\langle\ (\ not\ UIs\ )^*\ .\ 'GLOBAL_SYNTHESIS'\ \rangle\ true$

   This property ensures that, in all user interfaces, there is always the possibility to come back to the main UI (called *global synthesis*, Figure 50) with one single user interaction, i.e., without the need to access intermediate UIs before.

2. "A UI is only accessible along the hierarchy of UIs.":

   $macro\ HIERARCHICAL_ACCESS\ (V1,\ V2)\ =$
   $\quad [\ true^*\ .\ (\ UIs\ and\ (not\ V1)\ )\ .\ (\ not\ UIs^*\ )\ .\ V2\ ]\ false$
   $end\ macro$

   $HIERARCHICAL_ACCESS\ (\ 'SAFETY',\ 'SAFETY_REACTIVITY'\ )$
   and
   $HIERARCHICAL_ACCESS\ (\ 'SAFETY',\ 'SAFETY_COOLING'\ )$
   and
   ...
   (it continues for all pairs of UIs)

   where "UIs" is another macro containing a formula with disjunctive clauses of all UIs of the system. This property is written as a macro, and at each execution, to V1 and V2 are assigned pairs of UIs, where V1 is a UI through which V2 can be accessed. This formula describes the rules to accessing a UI by the menu, described in Subsection 5.2.2 - b) (page 98). It is a *safety* property, thus, it expresses that it is never the case that a given user interface V2 is accessed thought a sequence of UIs where a UI V1 is not in.

3. "One can always come back directly to the parent UI (i.e., without passing through any

other UI before).":

$macro\ BACKWARDS_NAVIGATION\ (V1,\ V2)\ =$
$\ [\ true^*\ .\ V1\ ]\ \langle\ (not\ UIs)^*\ .\ V2\ \rangle\ true$
$end\ macro$

$BACKWARDS_NAVIGATION\ (\ 'SAFETY_REACTIVITY',\ 'SAFETY'\ )$
and
$BACKWARDS_NAVIGATION\ (\ 'SAFETY_COOLING',\ 'SAFETY'\ )$
...
(it continues for all pairs of UIs)

Knowing that, in the hierarchy of UIs (cf. Subsection 5.2.2 - b) on page 98), the $SAFETY$ UI ($V2$ UI) is the parent UI of the $SAFETY_REACTIVITY$ UI ($V1$ UI), this property states that once $V1$ had been accessed, it is always possible to come back to $V2$ without passing through any other UI before (i.e., $(not\ UIs)^*$).

4. "From any state one can always reach any UI.":

$macro\ ACCESSIBLE_UIS\ (V)\ =$
$\ [\ true^*\ ]\ \langle\ true^*\ .\ V\ \rangle\ true$
$end\ macro$

$ACCESSIBLE_UIS\ (\ 'GLOBAL_SYNTHESIS'\ )$
and
$ACCESSIBLE_UIS\ (\ 'SAFETY'\ )$
...
(it continues for all UIs)

This property states that from any state, all UIs will be eventually accessed.

5. "There is no deadlock in the system.":

$[\ true^*\ ]\ \langle\ true\ \rangle\ true$

This formula states that every state has at least one successor. It means that, in every state of the system, an evolution in time is always possible.

6. "The UIs that display the signal details should be always accessible independently of the

evolution of the reactor parameters.”:

$macro\ ACCESSIBILITY_SIGNAL_UIS\ (S)\ =$
$\ [\ true^*\ ]\ \langle\ (not\ 'VOIR_PARAMS\ .^{*\prime})^*\ .\ S\ \rangle\ true$
$end\ macro$

$ACCESSIBILITY_SIGNALS_UIS\ (\ 'VUE_SIGNAUX_ALERTES'\ )$
and
$ACCESSIBILITY_SIGNALS_UIS\ (\ 'VUE_SIGNAUX_IHNIBES'\ )$
...
(it continues for all signal UIs)

This property verifies that the access to the UIs that display the details of the signals are independent from the evolution of the reactor parameters. Due to the fact that, even though signals are generated when parameters have anomalies (cf. Subsection 5.2.3 - b) on page 103), some signals are independent from that. Thus, the signal UIs should be accessible independently of the evolution of the reactor parameters.

7. “Starting from any state, the reactor generates **all five** anomalies on each parameter.”:

$macro\ FAILURES_COVERAGE\ (P)\ =$
$\ [\ true^*\ ]$
$\ \langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ DEP_HAUT.^{*\prime}\ \rangle\ true$
$\ and$
$\ \langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ DEP_BAS.^{*\prime}\ \rangle\ true$
$\ and$
$\ \langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ DEP_GRAD.^{*\prime}\ \rangle\ true$
$\ and$
$\ \langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ INV_MESURE.^{*\prime}\ \rangle\ true$
$\ and$
$\ \langle\ true^*\ .\ 'VOIR_PARAMS.^*\ !'\ \#\ P\ \#\ '(.\backslash\{1,7\backslash\}\ ,\ PERTE_RED.^{*\prime}\ \rangle\ true$
$end\ macro$

$FAILURES_COVERAGE\ (\ 'TPTH_MOY'\ )$
and
$FAILURES_COVERAGE\ (\ 'TDPAXMOY'\ )$
...
(it continues for all reactor parameters)

This property is expressed as a macro (i.e., $FAILURES_COVERAGE$), and at each execution, it verifies that the five anomaly scenarios (i.e., *threshold overflow* – “dépassement haut”, *threshold underflow* – “dépassement bas”, *gradient excess* – “dépassement

gradient", *invalid measurement* – "invalidité de mesure", and *loss of redundancy* – "perte de redondance") are simulated over each reactor parameter (e.g., "Pth moy" parameter).

8. "Starting from any state, the reactor generates **at least** one anomaly on each parameter.":

```
macro ONE_FAILURE (P) =
 [ true* ]
 ⟨ true* .
   (
    'VOIR_PARAMS.* !' # P # '(.\{1,7\}, DEP_HAUT.*'
    or
    'VOIR_PARAMS.* !' # P # '(.\{1,7\}, DEP_BAS.*'
    or
    'VOIR_PARAMS.* !' # P # '(.\{1,7\}, DEP_GRAD.*'
    or
    'VOIR_PARAMS.* !' # P # '(.\{1,7\}, INV_MESURE.*'
    or
    'VOIR_PARAMS.* !' # P # '(.\{1,7\}, PERTE_RED.*'
   )
 ⟩ true
end macro

ONE_FAILURE ( 'TPTH_MOY' )
and
ONE_FAILURE ( 'TDPAXMOY' )
...
(it continues for all reactor parameters)
```

Depending on the verification goals, one can verify whether the reactor generates on each parameter either **all five** anomalies (Property n. 7) or **at least** one anomaly (Property n. 8).

9. "Starting from any state, the reactor generates **at least** one anomaly on each parameter

belonging to a specific plant status.”:

```
macro PLANT_STATUS_FAILURE (PS, P) =
  [ true* . 'FIXER_ET !' # PS . ( not 'FIXER_ET .*')* ]
  ⟨ ( not 'FIXER_ET .*' )*.
    (
     'VOIR_PARAMS.* !' # P # '(.\{1,7\}, DEP_HAUT.*'
     or
     'VOIR_PARAMS.* !' # P # '(.\{1,7\}, DEP_BAS.*'
     or
     'VOIR_PARAMS.* !' # P # '(.\{1,7\}, DEP_GRAD.*'
     or
     'VOIR_PARAMS.* !' # P # '(.\{1,7\}, INV_MESURE.*'
     or
     'VOIR_PARAMS.* !' # P # '(.\{1,7\}, PERTE_RED.*'
    )
  ⟩ true
end macro

PLANT_STATUS_FAILURE ( 'RP', 'TPTH_MOY' )
and
PLANT_STATUS_FAILURE ( 'RP', 'TDPAXMOY' )
...
(it continues for all pairs of plant status and reactor parameters)
```

As explained in Section 3.3.1 (page 72), at the top of the user interfaces six plant status can be chosen, and the reactor parameters that are displayed on the UIs change according to the plant status. This property also verifies the coverage of the functional core model, by checking whether at least one anomaly is simulated over each parameter monitored in each plant status.

Appendix D
SVL Scripts

This appendix presents the SVL scripts created to implement the three abstraction techniques introduced in this thesis, named *generalization*, *omission* and *elimination*, in order to support the comparisons between the user interfaces of the EDF system described in Chapter 6. The UIs of this case study are adapted according to changes in their context of use, and five UI versions are proposed (cf. Section 3.4.2 on page 78): running on a PC, on a Smartphone, on a Tablet, in Training mode, and in Expert mode. The five versions of the UIs are compared to each other, as described in Section 6.6, page 151, and the use of the three aforementioned abstract techniques supports the process.

PC Version × Smartphone Version

The Svl script below illustrates the transformations done in the ISLTS representations of the PC version of the UI (in line 15, the `main_v46_b.bcg` file stores the LTS), and of the Smartphone version of the UI (in line 31, the `main_v55_b.bcg` file). The `rename` Svl operator implements the *generalization* abstraction technique (lines 3 and 19). The transformed ISLTS are then compared with each other in the light of the branching bisimulation relation (lines 1 and 17). If both LTS are branching equivalent, the script returns *true*, otherwise, a counter-example is generated and stored in the `diag.bcg` file (line 1). Here, the PC version and Smartphone version are branching equivalent.

```
1  "diag.bcg" = branching comparison generation of
2
3  total rename "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), FLECHEJAUNEHAUT).*)" ->
       "AFFICH_PARS (FLECHEJAUNEHAUT in \1)",
4                "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), FLECHEJAUNEBAS).*)"  ->
       "AFFICH_PARS (FLECHEJAUNEBAS in \1)",
5                "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), CADREMAGENTA).*)"    ->
       "AFFICH_PARS (CADREMAGENTA in \1)",
6                "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), CADREROUGE).*)"      ->
       "AFFICH_PARS (CADREROUGE in \1)",
7                "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), BARREORANGE).*)"     ->
       "AFFICH_PARS (BARREORANGE in \1)",
8                "AFFICH_PARS.*"                                                  ->
       "AFFICH_PARS (VIDE)",
9
10               "AFFICH_SIGN !AFFICH_SIGN_.*_RP (.*(\(.*\), CARREJAUNE.*)"       ->
       "AFFICH_SIGN (CARREJAUNE in \1)",
11               "AFFICH_SIGN !AFFICH_SIGN_.*_RP (.*(\(.*\),.*, CARREORANGE.*)"   ->
       "AFFICH_SIGN (CARREORANGE in \1)",
12               "AFFICH_SIGN !AFFICH_SIGN_.*_RP (.*(\(.*\),.*, NCROUGE.*)"       ->
       "AFFICH_SIGN (NCROUGE in \1)",
13               "AFFICH_SIGN !AFFICH_SIGN_.*_RP (.*(\(.*\),.*, CARREROUGE.*)"    ->
       "AFFICH_SIGN (CARREROUGE in \1)",
```

```
14              "AFFICH_SIGN.*"                                                  ->
     "AFFICH_SIGN (VIDE)"
15   in "main_v46_b.bcg"
16
17  ==
18
19 total rename "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), FLECHEJAUNEHAUT).*)" ->
     "AFFICH_PARS (FLECHEJAUNEHAUT in \1)",
20              "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), FLECHEJAUNEBAS).*)"  ->
     "AFFICH_PARS (FLECHEJAUNEBAS in \1)",
21              "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), CADREMAGENTA).*)"    ->
     "AFFICH_PARS (CADREMAGENTA in \1)",
22              "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), CADREROUGE).*)"      ->
     "AFFICH_PARS (CADREROUGE in \1)",
23              "AFFICH_PARS !AFFICH_PARS_.*_RP (.*(\(.*\), BARREORANGE).*)"     ->
     "AFFICH_PARS (BARREORANGE in \1)",
24              "AFFICH_PARS.*"                                                  ->
     "AFFICH_PARS (VIDE)",
25
26              "AFFICH_SIGN !AFFICH_UN_SIGNAL (.*(\(.*\), CARREJAUNE.*)"       ->
    "AFFICH_SIGN (CARREJAUNE in \1)",
27              "AFFICH_SIGN !AFFICH_UN_SIGNAL (.*(\(.*\),.*, CARREORANGE.*)"   ->
    "AFFICH_SIGN (CARREORANGE in \1)",
28              "AFFICH_SIGN !AFFICH_UN_SIGNAL (.*(\(.*\),.*, NCROUGE.*)"       ->
    "AFFICH_SIGN (NCROUGE in \1)",
29              "AFFICH_SIGN !AFFICH_UN_SIGNAL (.*(\(.*\),.*, CARREROUGE.*)"    ->
    "AFFICH_SIGN (CARREROUGE in \1)",
30              "AFFICH_SIGN.*"                                                  ->
    "AFFICH_SIGN (VIDE)"
31   in "main_v55_b.bcg"
```

Expert Version × Training Version

The SVL script below illustrates the nested transformations done in the ISLTS of the Expert mode version of the UI (in line 28, the `main_v67_b.bcg` file stores the LTS), and in the ISLTS of the Training mode version of the UI (in line 52, the `main_v66_b.bcg` file). The `rename` SVL operator implementing the *generalization* abstraction technique is also present in this script (e.g., lines 14 and 37). The `cut` SVL operator implements the *elimination* abstract technique (line 51), eliminating the references to the breadcrumb trail from the Training mode ISLTS (cf. Figure 116). The transformed ISLTS are then compared with each other in the light of the strong bisimulation relation (lines 1 and 31), showing that the Training model and the Expert model are strong equivalent modulo the breadcrumb trail.

```
1 "diag12.bcg" = strong comparison generation of
2
3
4 (* ################### RENOMMAGES DE LA VERSION EXPERT ################### *)
5     single rename
6       "TSIGNAL_COURBE (REACTIVITE, \(.*\) TSIGNAL_COURBE (CONFINEMENT, \(.*\)),
    TSIGNAL_COURBE (PROD, \(.*\)), TSIGNAL_COURBE (ELECTRICITE, \(.*\)
    TSIGNAL_COURBE (CLIM_VENTIL, \(.*\)))" ->
7       "CATSIGNAUX (TSIGNAL_COURBE (REACTIVITE, \1 TSIGNAL_COURBE (CONFINEMENT,
    \2)), CATSIGNAUX (TSIGNAL_COURBE (PROD, \3)), CATSIGNAUX (TSIGNAL_COURBE (
    ELECTRICITE, \4 TSIGNAL_COURBE (CLIM_VENTIL, \5)))"
```

```
8       in generation of
9
10          single rename "AFFICHER_SIGNAUX !AFFICHER_SIGNAUX_\(.*\)_RP ("      ->
        "AFFICHER_SIGNAUX !AFFICHER_SIGNAUX_\1_RP (ZONETITRE, ",
11                        "AFFICHER_PARAMS !AFFICHER_PARAMS_\(.*\)_RP ("        ->
        "AFFICHER_PARAMS !AFFICHER_PARAMS_\1_RP (ZONETITRE, "
12          in generation of
13
14                multiple rename "SIGNAL_VIDE" -> "SYMBOLE_SANS_DEFAUT"
15                in "main_b_v67p.bcg"
16
17
18  ==
19
20
21  (* ################## RENOMMAGES DE LA VERSION FORMATION ################## *)
22      multiple rename "CATSIGNAUX_[A-Z0-9_]* (" -> "CATSIGNAUX ("
23      in generation of
24
25              single rename "ZONE_SIGNAUX_DE_DEFAUT"  ->  "ZONETITRE",
26                            "ZONE_PARAMETRES"         ->  "ZONETITRE"
27              in generation of
28
29                multiple rename "SIGNAL_GRIS" ->  "SYMBOLE_SANS_DEFAUT"
30                in generation of
31
32                        partial cut "ENVOYER_FILARIANE", "FIL_.*"
33                        in "main_b_v66p.bcg"
```

PC Version × Tablet Version

The SVL script below illustrates the inclusion verification between the ISLTS of the PC version of the UI (line 3), and the ISLTS of the Tablet version of the UI (line 7). Here, no abstraction technique is needed to transform the ISLTS, and both are compared with each other in the light of the strong pre-order relation (lines 1 and 5), showing that the PC version includes the Tablet version.

```
1 "diag.bcg" = strong comparison
2
3 "main_vs69p_b_PC.bcg"
4
5 >=
6
7 "main_vs68p_b_Tablet.bcg"
```

Bibliography

[1012-2004 2005] IEEE Std 1012-2004. *IEEE Std 1012 - 2004 IEEE Standard for Software Verification and Validation*, 2005.

[Abowd *et al.* 1992] Gregory D. Abowd, Joëlle Coutaz et Laurence Nigay. *Structuring the Space of Interactive System Properties.* In Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction, pages 113–129, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[Abowd *et al.* 1995] Gregory D Abowd, Hung-Ming Wang et Andrew F Monk. *A Formal Technique for Automated Dialogue Development.* In Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques, pages 219–226. ACM, 1995.

[Abowd 1991] Gregory D Abowd. *Formal Aspects of Human-computer Interaction.* PhD thesis, University of Oxford, 1991.

[Abrial 1996] J.-R. Abrial. The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA, 1996.

[Abrial 2006] Jean-Raymond Abrial. *Formal Methods in Industry: Achievements, Problems, Future.* In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 761–768, New York, NY, USA, 2006. ACM.

[Acharya *et al.* 2010] Chitra Acharya, Harold W. Thimbleby et Patrick Oladimeji. *Human Computer Interaction and Medical Devices.* In Proceedings of the 2010 British Computer Society Conference on Human-Computer Interaction, BCS-HCI 2010, Dundee, United Kingdom, 6-10 September 2010, pages 168–176, 2010.

[AFCEN 2012] AFCEN. *Règles de Conception et de Construction des Matériels Electriques des Centrales Nucléaires*, 2012.

[Aït-Ameur & Baron 2004] Yamine Aït-Ameur et Mickaël Baron. *Bridging the Gap Between Formal and Experimental Validation Approaches in HCI Systems Design: Use of the Event B Proof based Technique.* In International Symposium on Leveraging Applications of Formal Methods, ISoLA 2004, October 30 - November 2, 2004, Paphos, Cyprus. Preliminary proceedings, pages 74–80, 2004.

[Aït-Ameur & Baron 2006] Yamine Aït-Ameur et Mickaël Baron. *Formal and Experimental Validation Approaches in HCI Systems Design based on a Shared Event B Model.* International Journal on Software Tools for Technology Transfer, vol. 8, no. 6, pages 547–563, 2006.

[Aït-Ameur & Kamel 2004] Yamine Aït-Ameur et Nadjet Kamel. *A Generic Formal Specification of Fusion of Modalities in a Multimodal HCI.* In Building the Information Society, pages 415–420. Springer, 2004.

[Aït-Ameur *et al.* 1998a] Yamine Aït-Ameur, Patrick Girard et Francis Jambon. *A Uniform Approach for Specification and Design of Interactive Systems: the B Method.* In Design, Specification and Verification of Interactive Systems'98, Supplementary Proceedings of the Fifth International Eurographics Workshop, June 3-5, 1998, Abingdon, United Kingdom, pages 51–67, 1998.

[Aït-Ameur *et al.* 1998b] Yamine Aït-Ameur, Patrick Girard et Francis Jambon. *A Uniform Approach for the Specification and Design of Interactive Systems: The B Method.* In Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSVIS'98), Vol. Proceedings (Eds, Markopoulos, P. and Johnson, P.), Abingdon, UK, pages 333–352, 1998.

[Aït-Ameur *et al.* 1999] Yamine Aït-Ameur, Patrick Girard et Francis Jambon. *Using the B Formal Approach for Incremental Specification Design of Interactive Systems.* In Engineering for Human-Computer Interaction, pages 91–109. Springer, 1999.

[Aït-Ameur *et al.* 2003a] Yamine Aït-Ameur, Mickaël Baron et Patrick Girard. *Formal Validation of HCI User Tasks.* In Software Engineering Research and Practice, pages 732–738, 2003.

[Aït-Ameur *et al.* 2003b] Yamine Aït-Ameur, Mickaël Baron et Nadjet Kamel. *Utilisation de Techniques Formelles dans la Modélisation d'Interfaces Homme-machine. Une Expérience Comparative entre B et Promela/SPIN.* In 6th International Symposium on Programming and Systems ISPS, pages 57–66, 2003.

[Aït-Ameur *et al.* 2004] Yamine Aït-Ameur, Benoit Breholée, Patrick Girard, Laurent Guittet et Francis Jambon. *Formal Verification and Validation of Interactive Systems Specifications.* In Human Error, Safety and Systems Development, pages 61–76. Springer, 2004.

[Aït-Ameur *et al.* 2005] Yamine Aït-Ameur, Mickaël Baron et Nadjet Kamel. *Encoding a Process Algebra Using the Event B Method. Application to the Validation of User Interfaces.* In Proceedings of 2nd IEEE international symposium on leveraging applications of formal methods (ISOLA), 2005.

[Aït-Ameur *et al.* 2006] Yamine Aït-Ameur, Idir Aït-Sadoune, Jean-Marc Mota et Mickaël Baron. *Validation et Vérification Formelles de Systèmes Interactifs Multi-modaux Fondées sur la Preuve.* In Proceedings of the 18th International Conferenceof the Association Francophone d'Interaction Homme-Machine, pages 123–130. ACM, 2006.

[Aït-Ameur *et al.* 2009] Yamine Aït-Ameur, Mickaël Baron, Nadjet Kamel et Jean-Marc Mota. *Encoding a Process Algebra Using the Event B Method.* STTT, vol. 11, no. 3, pages 239–253, 2009.

[Aït-Ameur *et al.* 2010] Yamine Aït-Ameur, Idir Aït-Sadoune, Mickaël Baron et Jean-Marc Mota. *Vérification et Validation Formelles de Systèmes Interactifs Fondées sur la Preuve: Application aux Systèmes Multi-Modaux.* Journal d'Interaction Personne-Systéme (JIPS), vol. 1, no. 1, pages 1–30, 2010.

[Aït-Ameur *et al.* 2014] Yamine Aït-Ameur, J Paul Gibson et Dominique Méry. *On Implicit and Explicit Semantics: Integration Issues in Proof-based Development of Systems.* In

Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, pages 604–618. Springer, 2014.

[Aït-Ameur 2000] Yamine Aït-Ameur. *Cooperation of Formal Methods in an Engineering based Software Development Process.* In Integrated Formal Methods, Second International Conference, IFM 2000, Dagstuhl Castle, Germany, November 1-3, 2000, Proceedings, pages 136–155, 2000.

[Aït-Sadoune & Aït-Ameur 2008] Idir Aït-Sadoune et Yamine Aït-Ameur. *Animating Event B Models by Formal Data Models.* In Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings, pages 37–55, 2008.

[Alsmadi 2013] Izzat Mahmoud Alsmadi. *Using Mutation to Enhance GUI Testing Coverage.* Software, IEEE, vol. 30, no. 1, pages 67–73, 2013.

[Ameur *et al.* 2010] Yamine Ait Ameur, Frédéric Boniol et Virginie Wiels. *Toward a Wider Use of Formal Methods for Aerospace Systems Design and Verification.* International Journal on Software Tools for Technology Transfer, vol. 12, no. 1, pages 1–7, 2010.

[Baecker & Buxton 2014] Ronald M Baecker et William AS Buxton. Readings in Human-computer Interaction. Elsevier Science, 2014.

[Bass *et al.* 1991] Len Bass, Reed Little, Robert Pellegrino, Scott Reed, Robert Seacord, Sylvia Sheppard et Martha R Szezur. *The ARCH model: Seeheim Revisited.* In User Interface Developpers' Workshop, 1991.

[Bastide & Palanque 1990] Rémi Bastide et Philippe A Palanque. *Petri Net Objects for the Design, Validation and Prototyping of User-driven Interfaces.* In Interact, volume 90, pages 625–631, 1990.

[Bastide & Palanque 1995] Rémi Bastide et Philippe Palanque. *A Petri Net based Environment for the Design of Event-driven Interfaces.* In Application and Theory of Petri Nets 1995, pages 66–83. Springer, 1995.

[Bastide *et al.* 2003] Rémi Bastide, David Navarre et Philippe Palanque. *A Tool-supported Design Framework for Safety Critical Interactive Systems.* Interacting with computers, vol. 15, no. 3, pages 309–328, 2003.

[Bastide *et al.* 2004] Rémi Bastide, David Navarre, Philippe A. Palanque, Amélie Schyn et Pierre Dragicevic. *A Model-based Approach for Real-time Embedded Multimodal Systems in Military Aircrafts.* In Proceedings of the 6th International Conference on Multimodal Interfaces, ICMI 2004, State College, PA, USA, October 13-15, 2004, pages 243–250, 2004.

[Bastien & Scapin 1993] J.M. Christian Bastien et Dominique L. Scapin. *Ergonomic Criteria for the Evaluation of Human-computer Interfaces.* Rapport technique RT-0156, INRIA, Juin 1993.

[Bauersfeld & Vos 2012] Sebastian Bauersfeld et Tanja E. J. Vos. *GUITest: A Java Library for Fully Automated GUI Robustness Testing.* In IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012, pages 330–333, 2012.

[Bauersfeld & Vos 2014] Sebastian Bauersfeld et Tanja E. J. Vos. *User Interface Level Testing with TESTAR; What about More Sophisticated Action Specification and Selection?* In Post-proceedings of the Seventh Seminar on Advanced Techniques and Tools for Software Evolution, SATToSE 2014, L'Aquila, Italy, 9-11 July 2014., pages 60–78, 2014.

[Bauersfeld *et al.* 2011a] Sebastian Bauersfeld, Stefan Wappler et Joachim Wegener. *A Metaheuristic Approach to Test Sequence Generation for Applications with a GUI.* In Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings, pages 173–187, 2011.

[Bauersfeld *et al.* 2011b] Sebastian Bauersfeld, Stefan Wappler et Joachim Wegener. *An Approach to Automatic Input Sequence Generation for GUI Testing Using Ant Colony Optimization.* In 13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Companion Material Proceedings, Dublin, Ireland, July 12-16, 2011, pages 251–252, 2011.

[Bauersfeld *et al.* 2014a] Sebastian Bauersfeld, Antonio de Rojas et Tanja E. J. Vos. *Evaluating ROGUE User Testing in Industry: An Experience Report.* In IEEE 8th International Conference on Research Challenges in Information Science, RCIS 2014, Marrakech, Morocco, May 28-30, 2014, pages 1–10, 2014.

[Bauersfeld *et al.* 2014b] Sebastian Bauersfeld, Tanja E. J. Vos, Nelly Condori-Fernández, Alessandra Bagnato et Etienne Brosse. *Evaluating the TESTAR Tool in an Industrial Case Study.* In 2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014, pages 4:1–4:9, 2014.

[Bauersfeld 2013] Sebastian Bauersfeld. *GUIdiff - A Regression Testing Tool for Graphical User Interfaces.* In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 18-22, 2013, pages 499–500, 2013.

[Bertino 1985] E Bertino. *Design Issues in Interactive User Interfaces.* Interfaces in Computing, vol. 3, no. 1, pages 37 – 53, 1985.

[Bevan 2001] Nigel Bevan. *International Standards for HCI and Usability.* International journal of human-computer studies, vol. 55, no. 4, pages 533–552, 2001.

[Bhasin *et al.* 2013] Harsh Bhasin, Harish Kumar et Vikas Singh. *Orthogonal Testing Using Genetic Algorithms.* International Journal of Computer Science and Information Technology, vol. 4, no. 2, pages 374–377, 2013.

[Bin & Anbao 2012] Zhu Bin et Wang Anbao. *Functional and User Interface Model for Generating Test Cases.* In Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on, pages 605–610. IEEE, 2012.

[Bolton 2008] Matthew L. Bolton. *Modeling Human Perception Could Stevens' Power Law Be an Emergent Feature?* In SMC, pages 1073–1078. IEEE, 2008.

[Booch *et al.* 2005] Grady Booch, James Rumbaugh et Ivar Jacobson. Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series). Addison-Wesley Professional, 2005.

[Booth 1989] Paul A Booth. An Introduction to Human-computer Interaction. Psychology Press, 1989.

[Borjesson & Feldt 2012] Emil Borjesson et Robert Feldt. *Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry.* In ICST, pages 350–359. IEEE, 2012.

[Bouajjani *et al.* 1991] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodriguez et Joseph Sifakis. *Safety for Branching Time Semantics.* In Automata, Languages and Programming, pages 76–92. Springer, 1991.

[Bowen & Reeves 2005] Judy Bowen et Steve Reeves. *Including Design Guidelines in the Formal Specification of Interfaces in Z.* In ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings, pages 454–471, 2005.

[Bowen & Reeves 2006] Judy Bowen et Steve Reeves. *Formal Refinement of Informal GUI Design Artefacts.* In 17th Australian Software Engineering Conference (ASWEC 2006), 18-21 April 2006, Sydney, Australia, pages 221–230, 2006.

[Bowen & Reeves 2007a] Judy Bowen et Steve Reeves. *Formal Models for Informal GUI Designs.* Electr. Notes Theor. Comput. Sci., vol. 183, pages 57–72, 2007.

[Bowen & Reeves 2007b] Judy Bowen et Steve Reeves. *Using Formal Models to Design User Interfaces: A Case Study.* In Proceedings of the 21st British HCI Group Annual Conference on HCI 2007: HCI...but not as we know it - Volume 1, BCS HCI 2007, University of Lancaster, United Kingdom, 3-7 September 2007, pages 159–166, 2007.

[Bowen & Reeves 2008a] Judy Bowen et Steve Reeves. *Formal Models for User Interface Design Artefacts.* ISSE, vol. 4, no. 2, pages 125–141, 2008.

[Bowen & Reeves 2008b] Judy Bowen et Steve Reeves. *Refinement for User Interface Designs.* Electr. Notes Theor. Comput. Sci., vol. 208, pages 5–22, 2008.

[Bowen & Reeves 2011] Judy Bowen et Steve Reeves. *UI-driven Test-first Development of Interactive Systems.* In Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2011, Pisa, Italy, June 13-16, 2011, pages 165–174, 2011.

[Bowen & Reeves 2012] Judy Bowen et Steve Reeves. *Modelling User Manuals of Modal Medical Devices and Learning from the Experience.* In ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'12, Copenhagen, Denmark - June 25 - 28, 2012, pages 121–130, 2012.

[Bowen & Reeves 2013a] Judy Bowen et Steve Reeves. *Modelling Safety Properties of Interactive Medical Systems.* In ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'13, London, United Kingdom - June 24 - 27, 2013, pages 91–100, 2013.

[Bowen & Reeves 2013b] Judy Bowen et Steve Reeves. *UI-design Driven Model-based Testing.* ISSE, vol. 9, no. 3, pages 201–215, 2013.

[Bowen 2015] Judy Bowen. *Creating Models of Interactive Systems with the Support of Lightweight Reverse-Engineering Tools.* In Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2015, Duisburg, Germany, June 23-26, 2015, pages 110–119, 2015.

[Boyer & Moore 1983] Robert S Boyer et J Strother Moore. *Proof-Checking, Theorem Proving, and Program Verification.* Rapport technique, DTIC Document, 1983.

[Brat *et al.* 2013] Guillaume Brat, Célia Martinie et Philippe Palanque. *V&V of Lexical, Syntactic and Semantic Properties for Interactive Systems through Model Checking of Formal Description of Dialog.* In Proceedings of the 15th International Conference on Human-Computer Interaction: Human-centred Design Approaches, Methods, Tools, and Environments - Volume Part I, HCI'13, pages 290–299, Berlin, Heidelberg, 2013. Springer-Verlag.

[Bumbulis *et al.* 1995a] Peter Bumbulis, P. S. C. Alencar, D. D. Cowan et C. J. P. Lucena. *Combining Formal Techniques and Prototyping in User Interface Construction and Verification.* In In 2nd Eurographics Workshop on Design, Specification, Verification of Interactive Systems (DSV-IS'95). Springer-Verlag Lecture Notes in Computer Science, pages 7–9. Springer-Verlag, 1995.

[Bumbulis *et al.* 1995b] Peter Bumbulis, Paulo S. C. Alencar, Donald D. Cowan et Carlos José Pereira de Lucena. *A Framework for Machine-Assisted User Interface Verification.* In Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology, AMAST '95, pages 461–474, London, UK, UK, 1995. Springer-Verlag.

[Bumbulis *et al.* 1996] Peter Bumbulis, Paulo S. C. Alencar, Donald D. Cowan et Carlos José Pereira de Lucena. *Validating Properties of Component-based Graphical User Interfaces.* In Design, Specification and Verification of Interactive Systems'96, Proceedings of the Third International Eurographics Workshop, June 5-7, 1996, Namur, Belgium, pages 347–365, 1996.

[Butler & Finelli 1993] Ricky W Butler et George B Finelli. *The Infeasibility of Quantifying the Reliability of Life-critical Real-time Software.* Software Engineering, IEEE Transactions on, vol. 19, no. 1, pages 3–12, 1993.

[Calvary *et al.* 2003] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon et Jean Vanderdonckt. *A Unifying Reference Framework for Multi-Target User Interfaces.* Interacting with Computers, vol. 15, no. 3, pages 289–308, 2003.

[Calvary *et al.* 2011] Gaëlle Calvary, Audrey Serna, Joëlle Coutaz, Dominique Scapin, Florence Pontico et Marco Winckler. *Envisioning Advanced User Interfaces for E-government Applications: A Case Study.* In Practical Studies in E-Government, pages 205–228. Springer, 2011.

[Campos & Harrison 1997] José C Campos et Michael D Harrison. Formally Verifying Interactive Systems: A Review. Springer, 1997.

[Campos & Harrison 1998] José C Campos et Michael D Harrison. The Role of Verification in Interactive Systems Design. Springer, 1998.

[Campos & Harrison 2001] José C Campos et Michael D Harrison. *Model Checking Interactor Specifications.* Automated Software Engineering, vol. 8, no. 3-4, pages 275–310, 2001.

[Campos & Harrison 2007] José Creissac Campos et Michael D. Harrison. *Considering Context and Users in Interactive Systems Analysis.* In Engineering Interactive Systems - EIS 2007 Joint Working Conferences, EHCI 2007, DSV-IS 2007, HCSE 2007, Salamanca, Spain, March 22-24, 2007. Selected Papers, pages 193–209, 2007.

[Campos & Harrison 2008] J Creissac Campos et Michael D Harrison. *Systematic Analysis of Control Panel Interfaces Using Formal Tools.* In Interactive Systems. Design, Specification, and Verification, pages 72–85. Springer, 2008.

[Campos & Harrison 2009] José C. Campos et Michael D. Harrison. *Interaction Engineering Using the IVY Tool.* In Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '09, pages 35–44, New York, NY, USA, 2009. ACM.

[Campos & Harrison 2011] José Creissac Campos et Michael D. Harrison. *Modelling and Analysing the Interactive Behaviour of an Infusion Pump.* ECEASST, vol. 45, 2011.

[Campos *et al.* 2014] José Creissac Campos, Gavin J. Doherty et Michael D. Harrison. *Analysing Interactive Devices based on Information Resource Constraints.* Int. J. Hum.-Comput. Stud., vol. 72, no. 3, pages 284–297, 2014.

[Campos 1999] José Creissac Campos. Automated Deduction and Usability Reasoning. Phd thesis, Department of Computer Science, University of York, Heslington, York, 1999.

[Carl 1962] A Carl. *Petri. Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr, vol. 2, 1962. English Translation, 1966: Communication with Automata, Technical Report RADC-TR-65-377, Rome Air Dev. Center, New York.

[Carroll 2013] John M Carroll. Human computer interaction-brief intro. The Interaction Design Foundation, 2013.

[Cauchi *et al.* 2012a] Abigail Cauchi, Andy Gimblett, Harold Thimbleby, Paul Curzon et Paolo Masci. *Safer "5-key" Number Entry User Interfaces Using Differential Formal Analysis.* In Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers, BCS-HCI '12, pages 29–38, Swinton, UK, UK, 2012. British Computer Society.

[Cauchi *et al.* 2012b] Abigail Cauchi, Andy Gimblett, Harold W. Thimbleby, Paul Curzon et Paolo Masci. *Safer "5-key" Number Entry User Interfaces Using Differential Formal Analysis.* In BCS-HCI '12 Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers, BCS-HCI 2012, 12-14 September 2012, Birmingham, UK, pages 29–38, 2012.

[Cauchi *et al.* 2014] Abigail Cauchi, Patrick Oladimeji, Gerrit Niezen et Harold W. Thimbleby. *Triangulating Empirical and Analytic Techniques for Improving Number Entry User Interfaces.* In ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'14, Rome, Italy, June 17-20, 2014, pages 243–252, 2014.

[Champelovier *et al.* 2011] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe et Gideon Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 149 pages, Septembre 2011.

[Champelovier *et al.* 2014] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe et Gideon Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.1). INRIA/VASY and INRIA/CONVECS, 131 pages, Août 2014.

[Chériaux *et al.* 2012] François Chériaux, Dominique Galara et Marion Viel. *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, pages 1002–1012. Curran Associates, Inc., 2012.

[Chinnapongse *et al.* 2009] Vivien Chinnapongse, Insup Lee, Shaohui Wang et Paul L Jones. *Model-based Testing of GUI-driven Applications.* In Sunggu Lee et Priya Narasimhan, editeurs, The Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009), LNCS 5860, volume 5860 of *Lecture Notes in Computer Science*, pages 203–214, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Clarke *et al.* 1983] E. M. Clarke, E. A. Emerson et A. P. Sistla. *Automatic Verification of Finite State Concurrent System Using Temporal Logic Specifications: A Practical Approach.* In Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '83, pages 117–126, New York, NY, USA, 1983. ACM.

[Cofer *et al.* 2008] Darren Cofer, Michael Whalen et Steven Miller. *Software Model Checking for Avionics Systems.* In Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th, pages 5–D. IEEE, 2008.

[Cofer *et al.* 2012] Darren Cofer, Andrew Gacek, Steven Miller, Michael W. Whalen, Brian LaValley et Lui Sha. *Compositional Verification of Architectural Models.* In Proceedings of the 4th International Conference on NASA Formal Methods, NFM'12, pages 126–140, Berlin, Heidelberg, 2012. Springer-Verlag.

[Cofer 2010] Darren Cofer. *Model Checking: Cleared for Take off.* In Model Checking Software, pages 76–87. Springer, 2010.

[Cofer 2012] Darren Cofer. *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, pages 2–3, Berlin, Heidelberg, 2012. Springer-Verlag.

[Combéfis & Pecheur 2009] Sébastien Combéfis et Charles Pecheur. *A Bisimulation-based Approach to the Analysis of Human-computer Interaction.* In Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '09, pages 101–110, New York, NY, USA, 2009. ACM.

[Combéfis *et al.* 2011a] Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur et Michael Feary. *A Formal Framework for Design and Analysis of Human-machine Interaction.* In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Anchorage, Alaska, USA, October 9-12, 2011, pages 1801–1808, 2011.

[Combéfis *et al.* 2011b] Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur et Michael Feary. *Learning System Abstractions for Human Operators.* In Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering, MALETS '11, pages 3–10, New York, NY, USA, 2011. ACM.

[Combéfis 2013] Sébastien Combéfis. A Formal Framework for the Analysis of Human-machine Interactions, volume 459. Doctoral thesis, Université catholique de Louvain, 2013.

[Commission *et al.* 1997] Nuclear Regulatory Commission*et al. Operator Licensing Examination Standards for Power Reactors. Interim Revision 8.* Rapport technique, Nuclear Regulatory Commission, Washington, DC (United States). Div. of Reactor Controls and Human Factors. Funding organisation: Nuclear Regulatory Commission, Washington, DC (United States), 1997.

[Connexion 2012] Cluster Connexion. *Expression des Besoins par Points de Vue Métier, Définition des Études de Cas de l'Innovation [Livrable n.3.1.1 du Projet Connexion]*, 2012.

[Connexion 2013] Cluster Connexion. *Maquettes pour des IHM de supervision (cas d'étude) [Livrable n.3.1.3a du Projet Connexion]*, 2013.

[Connexion 2014] Cluster Connexion. *Cahier des Charges Démonstrateur IHM [Livrable du Projet Connexion]*, 2014.

[Cortier *et al.* 2007] Alexandre Cortier, Bruno d'Ausbourg et Yamine Aït-Ameur. *Formal Validation of Java/Swing User Interfaces with the Event B Method.* In Human-Computer Interaction. Interaction Design and Usability, pages 1062–1071. Springer, 2007.

[Coutaz & Calvary 2012] Joëlle Coutaz et Gaëlle Calvary. *HCI and Software Engineering for User Interface Plasticity.* Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications, Third Edition, pages 1195–1220, 2012.

[Coutaz *et al.* 2000] J Coutaz, C Lachenal, G Calvary et D Thevenin. *Software Architecture Adaptivity for Multisurface Interaction and Plasticity.* In Proc. of IFIP Workshop on Software Architecture Requirements for CSCW–CSCW'2000 Workshop, 2000.

[Coutaz *et al.* 2007] Joëlle Coutaz, Lionel Balme, Gaëlle Calvary, Alexandre Demeure et Jean-Sebastien Sottet. *An MDE-SOA Approach to Support Plastic User Interafces in Ambient Spaces.* In Proc. HCI International 2007, pages 152–171, 2007. Beijing, July 2007.

[Coutaz 1987] Joëlle Coutaz. *PAC, an Object Oriented Model for Dialog Design.* In Proceedings Interact, volume 87, pages 431–436, 1987.

[Crow *et al.* 2000] Judith Crow, Denis Javaux et John Rushby. *Models of Mechanised Methods that Integrate Human Factors into Automation Design.* In International Conference on Human-Computer Interaction in Aeronautics: HCI-Aero 2000. Citeseer, 2000.

[d'Ausbourg *et al.* 1996] Bruno d'Ausbourg, Guy Durrieu et Pierre Roché. *Deriving a Formal Model of an Interactive System from its UIL Description in order to Verify and Test its Behaviour.* In Design, Specification and Verification of Interactive Systems'96, Proceedings of the Third International Eurographics Workshop, June 5-7, 1996, Namur, Belgium, pages 105–122, 1996.

[d'Ausbourg *et al.* 1998] Bruno d'Ausbourg, Christel Seguin, Guy Durrieu et Pierre Roché. *Helping the Automated Validation Process of User Interfaces Systems.* In Proceedings of the 20th international conference on Software engineering, ICSE '98, pages 219–228, Washington, DC, USA, 1998. IEEE Computer Society.

[d'Ausbourg 1998] Bruno d'Ausbourg. *Using Model Checking for the Automatic Validation of User Interface Systems.* In Design, Specification and Verification of Interactive Systems'98, Proceedings of the Fifth International Eurographics Workshop, June 3-5, 1998, Abingdon, United Kingdom, pages 242–260, 1998.

[d'Ausbourg 2002] Bruno d'Ausbourg. *Synthétiser I'Intention d'un Pilote pour Définir de Nouveaux Équipements de Bord.* In Proceedings of the 14th French-speaking Conference on Human-computer Interaction (ConféRence Francophone Sur L'Interaction Homme-Machine), IHM '02, pages 145–152, New York, NY, USA, 2002. ACM.

[De Moura *et al.* 2004] Leonardo De Moura, Sam Owre, Harald Rueß, John Rushby, Natarajan Shankar, Maria Sorea et Ashish Tiwari. *SAL 2.* In Computer Aided Verification, pages 496–500. Springer Berlin Heidelberg, 2004.

[Degani & Heymann 2000] Asaf Degani et Michael Heymann. *Pilot-autopilot Interaction: A Formal Perspective.* Abbott et al.[1], pages 157–168, 2000.

[Degani & Heymann 2002] Asaf Degani et Michael Heymann. *Formal Verification of Human-automation Interaction.* Human Factors: The Journal of the Human Factors and Ergonomics Society, vol. 44, no. 1, pages 28–43, 2002.

[Degani & Heymann 2007] Asaf Degani et Michael Heymann. *Toward Automatic Generation of User Interfaces: Abstraction of Internal States and Transitions.* In Analysis, Design, and Evaluation of Human-Machine Systems, volume 10, pages 483–489, 2007.

[Degani *et al.* 1996] Asaf Degani, Michael Shafto et Alex Kirlik. *Modes in Automated Cockpits: Problems, Data Analysis and a Modelling Framework.* In ISRAEL ANNUAL CONFERENCE ON AEROSPACE SCIENCES, pages 258–266. OMANUTH PRESS LTD, 1996.

[Degani *et al.* 2000] Asaf Degani, Michael Heymann, George Meyer et Michael Shafto. *Some Formal Aspects of Human-automation Interaction.* NASA Technical Memorandum, vol. 209600, 2000.

[Degani *et al.* 2013] Asaf Degani, Michael Heymann et Michael Shafto. *Modeling and Formal Analysis of Human–machine Interaction.* The Oxford Handbook of Cognitive Engineering, page 433, 2013.

[Demeure *et al.* 2006] Alexandre Demeure, Gaëlle Calvary, Joëlle Coutaz et Jean Vanderdonckt. *Towards Run Time Plasticity Control based on a Semantic Network.* In Fifth International Workshop on Task Models and Diagrams for UI design (TAMODIA'06), pages 324–338, 2006. Hasselt, Belgium, October 23-24, 2006.

[Demeure *et al.* 2008] Alexandre Demeure, Gaëlle Calvary et Karin Coninx. *COMET(s), A Software Architecture Style and an Interactors Toolkit for Plastic User Interfaces.* In Design, Specification, and Verification, pages 225–237, 2008. 15th International Workshop, DSV-IS 2008, T.C.N. Graham & P. Palanque (Eds), Lecture Notes in Computer Science 5136, Springer Berlin / Heidelberg, Kingston, Canada, July 16-18.

[Dix *et al.* 1987] Alan J. Dix, Michael D. Harrison, Colin Runciman et Harold W. Thimbleby. *Interaction Models and the Principled Design of Interactive Systems.* In ESEC '87, 1st European Software Engineering Conference, Strasbourg, France, September 9-11, 1987, Proceedings, pages 118–126, 1987.

[Dix 1988] Alan J. Dix. *Abstract, Generic Models of Interactive Systems.* In People and Computers IV, Proceedings of Fourth Conference of the British Computer Society Human-Computer Interaction Specialist Group, University of Manchester, 5-9 September 1988, pages 63–77, 1988.

[Dix 1991] Alan John Dix. Formal Methods for Interactive Systems, volume 16. Academic Press London, 1991.

[Dix 1995] Alan J Dix. *Formal Methods: An Introduction to and Overview of the Use of Formal Methods within HCI.* Chapter, vol. 2, pages 9–43, 1995.

[Doherty *et al.* 1998] Gavin Doherty, José C. Campos et Michael D. Harrison. *Representational Reasoning and Verification.* Formal Aspects of Computing, vol. 12, pages 260–277, 1998.

[Duke & Harrison 1993] David J. Duke et Michael D. Harrison. *Abstract Interaction Objects.* In Computer Graphics Forum, volume 12, pages 25–36. Wiley Online Library, 1993.

[Duke & Harrison 1995] DJ Duke et MD Harrison. *Event Model of Human-system Interaction.* Software Engineering Journal, vol. 10, no. 1, pages 3–12, 1995.

[Duke *et al.* 1995] D. J. Duke, P. J. Barnard, J. May et D. A. Duce. *Systematic Development of the Human Interface.* In Proceedings of the Second Asia Pacific Software Engineering Conference, APSEC '95, pages 313–, Washington, DC, USA, 1995. IEEE Computer Society.

[Duke *et al.* 1999] David Duke, Bob Fields et Michael D Harrison. *A Case Study in the Specification and Analysis of Design Alternatives for a User Interface.* Formal Aspects of Computing, vol. 11, no. 2, pages 107–131, 1999.

[Elder & Knight 1995] Matthew C Elder et J Knight. *Specifying User Interfaces for Safety-critical Medical Systems.* In Proc. of the 2nd Annual International Symposium on Medical Robotics and Computer Assisted Surgery, pages 148–155, 1995.

[Faconti & Paternó 1990] Giorgio P. Faconti et Fabio Paternó. *An Approach to the Formal Specification of the Components of an Interaction*, 1990.

[Fernandez *et al.* 1996] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier et Mihaela Sighireanu. *CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox.* In Rajeev Alur et Thomas A. Henzinger, editeurs, Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA), volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer, Août 1996.

[Fields *et al.* 1995a] Bob Fields, Michael Harrison et Peter Wright. *Modelling Interactive Systems and Providing Task Relevant Information.* In Interactive Systems: Design, Specification, and Verification, pages 253–266. Springer, 1995.

[Fields *et al.* 1995b] Bob Fields, Peter Wright et Michael Harrison. *Applying Formal Methods for Human Error Tolerant Design.* In Software Engineering and Human-Computer Interaction, pages 185–195. Springer, 1995.

[Fields *et al.* 1995c] RE Fields, Peter C Wright et Michael D Harrison. *A Task Centered Approach to Analysing Human Error Tolerance Requirements.* In Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on, pages 18–26. IEEE, 1995.

[Foley & Wallace 1974] J. D. Foley et V. L. Wallace. *The Art of Natural Graphic Man-machine Conversation.* SIGGRAPH Comput. Graph., vol. 8, no. 3, pages 87–87, Septembre 1974.

[Ganneau *et al.* 2008] Vincent Ganneau, Gaëlle Calvary et Rachel Demumieux. *Learning Key Contexts of Use in the Wild for Driving Plastic User Interfaces Engineering.* In Engineering Interactive Systems 2008 (2nd Conference on Human-Centred Software Engineering (HCSE 2008) and 7th International workshop on TAsk MOdels and DIAgrams (TAMODIA 2008)), 2008. September 2008, Pisa (Italy).

[Garavel & Graf 2013] H Garavel et S Graf. *Formal Methods for Safe and Secure Computer Systems.* Federal Office for Information Security, 2013.

[Garavel & Lang 2001] Hubert Garavel et Frédéric Lang. *SVL: a Scripting Language for Compositional Verification.* In Myungchul Kim, Byoungmoon Chin, Sungwon Kang et Danhyung Lee, editeurs, Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea), pages 377–392. IFIP, Kluwer Academic Publishers, Août 2001. Full version available as INRIA Research Report RR-4223.

[Garavel *et al.* 2002] Hubert Garavel, Frédéric Lang et Radu Mateescu. *An Overview of CADP 2001.* European Association for Software Science and Technology (EASST) Newsletter, vol. 4, pages 13–24, Août 2002. Also available as INRIA Technical Report RT-0254 (December 2001).

[Garavel *et al.* 2007] Hubert Garavel, Frédéric Lang, Radu Mateescu et Wendelin Serwe. *CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes.* In Werner Damm et Holger Hermanns, editeurs, Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany), volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer, Juillet 2007.

[Garavel *et al.* 2013] Hubert Garavel, Frédéric Lang, Radu Mateescu et Wendelin Serwe. *CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes.* International Journal on Software Tools for Technology Transfer, vol. 15, no. 2, pages 89–107, 2013.

[Garavel *et al.* 2015] Hubert Garavel, Frédéric Lang et Radu Mateescu. *Compositional verification of asynchronous concurrent systems using CADP.* Acta Informatica, pages 1–56, 2015.

[Garavel 2015] Hubert Garavel. *Revisiting sequential composition in process calculi.* Journal of Logical and Algebraic Methods in Programming, page «to appear», 2015.

[Gimblett & Thimbleby 2010] Andy Gimblett et Harold W. Thimbleby. *User Interface Model Discovery: Towards a Generic Approach.* In Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2010, Berlin, Germany, June 19-23, 2010, pages 145–154, 2010.

[Gimblett & Thimbleby 2013] Andy Gimblett et Harold Thimbleby. *Applying Theorem Discovery to Automatically Find and Check Usability Heuristics.* In Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '13, pages 101–106, New York, NY, USA, 2013. ACM.

[Godefroid *et al.* 1998] Patrice Godefroid, John Kelly, Steven Miller et Frank Weil. *Transferring Formal Methods Technology to Industry.* In wift, page 128. IEEE, 1998.

[Göransson *et al.* 2003] Bengt Göransson, Jan Gulliksen et Inger Boivie. *The Usability Design Process–Integrating User-centered Systems Design in the Software Development Process.* Software Process: Improvement and Practice, vol. 8, no. 2, pages 111–131, 2003.

[Hall 1999] Anthony Hall. *Using Formal Methods to Develop an ATC Information System.* In Industrial-Strength Formal Methods in Practice, pages 207–229. Springer, 1999.

[Hallinger *et al.* 2000] Philip Hallinger, David P Crandall et David Ng Foo Seong. *Systems Thinking/Systems Changing & A Computer Simulation for Learning How to Make School Ssmarter.* Advances in Research and Theories of School Management and Educational Policy, vol. 1, no. 4, pages 15–24, 2000.

[Hamilton *et al.* 1995] David Hamilton, Richard Covington, John Kelly, Carron Kirkwood, Muffy Thomas, Alan R. Flora-Holmquist, Mark G. Staskauskas, Steven P. Miller, Mandayam K. Srivas, George Cleland et Donald MacKenzie. *Experiences in Applying Formal Methods to the Analysis of Software and System Requirements.* In Workshop on Industrial-Strength Formal Specification Techniques, WIFT 1995, Boca Raton, Florida, USA, April 5-8, 1995, pages 30–43, 1995.

[Hardin *et al.* 2009] David S. Hardin, T. Douglas Hiratzka, D. Randolph Johnson, Lucas Wagner et Michael W. Whalen. *Development of Security Software: A High Assurance Methodology.* In Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings, pages 266–285, 2009.

[Harrison & Duke 1995] Michael D Harrison et David J Duke. *A Review of Formalisms for Describing Interactive Behaviour.* In Software Engineering and Human-Computer Interaction, pages 49–75. Springer, 1995.

[Harrison *et al.* 1996] MD Harrison, R Fields et PC Wright. *The User Context and Formal Specification in Interactive System Design.* In Proceedings of the 1996 BCS-FACS conference on Formal Aspects of the Human Computer Interface, pages 5–5. British Computer Society, 1996.

[Harrison *et al.* 2013] Michael D. Harrison, Paolo Masci, José Creissac Campos et Paul Curzon. *Automated Theorem Proving for the Systematic Analysis of an Infusion Pump.* ECEASST, vol. 69, 2013.

[Harrison *et al.* 2015] Michael D. Harrison, José Creissac Campos et Paolo Masci. *Reusing Models and Properties in the Analysis of Similar Interactive Devices.* ISSE, vol. 11, no. 2, pages 95–111, 2015.

[Heymann & Degani 2002] Michael Heymann et Asaf Degani. *On the Construction of Human-automation Interfaces by Formal Abstraction.* In Abstraction, Reformulation, and Approximation, pages 99–115. Springer, 2002.

[Heymann & Degani 2007] Michael Heymann et Asaf Degani. *Formal Analysis and Automatic Generation of User Interfaces: Approach, Methodology, and an Algorithm.* Human Factors: The Journal of the Human Factors and Ergonomics Society, vol. 49, no. 2, pages 311–330, 2007.

[Hix & Hartson 1993] Deborah Hix et H. Rex Hartson. Developing User Interfaces: Ensuring Usability Through Product &Amp; Process. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[Hjort *et al.* 2009] Ulrik H Hjort, Jacob Illum, Kim G Larsen, Michael A Petersen et Arne Skou. *Model-based GUI Testing Using UPPAAL at NOVO Nordisk.* In FM 2009: Formal Methods, pages 814–818. Springer, 2009.

[Huang & Lu 2012] Y Huang et L Lu. *Apply Ant Colony to Event-flow Model for Graphical User Interface Test Case Generation.* IET software, vol. 6, no. 1, pages 50–60, 2012.

[IEC 2011] IEC. *IEC 61513:2011 – Nuclear Power Plants — Instrumentation and Control for Systems Important to Safety — General Requirements for System (Edition 2.0)*, 2011.

[Imaz & Benyon 2007] Manuel Imaz et David Benyon. Designing with Blends: Conceptual Foundations of Human-computer Interaction and Software Engineering Methods. Mit Press, 2007.

[ISO/IEC 1989] ISO/IEC. *LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva, Septembre 1989.

[ISO/IEC 2001] ISO/IEC. *Enhancements to LOTOS (E-LOTOS).* International Standard 15437:2001, International Organization for Standardization — Information Technology, Geneva, Septembre 2001.

[Jacky *et al.* 2007] Jonathan Jacky, Margus Veanes, Colin Campbell et Wolfram Schulte. Model-based Software Testing and Analysis with C#. Cambridge University Press, 1 édition, 2007.

[Jambon *et al.* 2001] Francis Jambon, Patrick Girard et Yamine Aït-Ameur. *Interactive System Safety and Usability Enforced with the Development Process.* In Engineering for Human-Computer Interaction, 8th IFIP International Conference, EHCI 2001, Toronto, Canada, May 11-13, 2001, Revised Papers, pages 39–56, 2001.

[Jung *et al.* 2012] Hyunjun Jung, Sukhoon Lee et Doo-Kwon Baik. *An Image Comparing-based GUI Software Testing Automation System.* In SERP, pages 318–322, 2012.

[Knight & Brilliant 1997] John C Knight et Susan S Brilliant. *Preliminary Evaluation of a Formal Approach to User Interface Specification.* In ZUM'97: The Z Formal Specification Notation, pages 329–346. Springer, 1997.

[Knight & Kienzle 1992] John C. Knight et Darrell M. Kienzle. *Preliminary Experience Using Z to Specify a Safety-critical System.* In Z User Workshop, London, UK, 14-15 December 1992, Proceedings, pages 109–118, 1992.

[Knight *et al.* 1999] John C. Knight, P. Thomas Fletcher et Brian R. Hicks. *Tool Support for Production Use of Formal Techniques.* In FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume II, page 1854, 1999.

[Knight 1998] John C Knight. *Challenges in the Utilization of Formal Methods.* In Formal Techniques in Real-Time and Fault-Tolerant Systems, pages 1–17. Springer, 1998.

[Leveson 1995] Nancy G Leveson. Safeware: System Safety and Computers. ACM, 1995.

[Lex & Powej 1997] Burton L Lex et Bechtel Powej. *Nuclear Power Plant Control and Instrumentation*, 1997.

[Li *et al.* 2015] Karen Yunqiu Li, Patrick Oladimeji et Harold W. Thimbleby. *Exploring the Effect of Pre-operational Priming Intervention on Number Entry Errors.* In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Republic of Korea, April 18-23, 2015, pages 1335–1344, 2015.

[Loer & Harrison 2000] Karsten Loer et Michael D. Harrison. *Formal Interactive Systems Analysis and Usability Inspection Methods: Two Incompatible Worlds?* In DSV-IS, pages 169–190, 2000.

[Loer & Harrison 2002] Karsten Loer et Michael Harrison. *Towards Usable and Relevant Model Checking Techniques for the Analysis of Dependable Interactive Systems.* In Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on, pages 223–226. IEEE, 2002.

[Loer & Harrison 2006] Karsten Loer et Michael D Harrison. *An Integrated Framework for the Analysis of Dependable Interactive Systems (IFADIS): Its Tool Support and Evaluation.* Automated Software Engineering, vol. 13, no. 4, pages 469–496, 2006.

[Long 1989] John Long. Cognitive Ergonomics and Human-computer Interaction, volume 1. Cambridge University Press, 1989.

[Lu & Huang 2012] Lu Lu et Ying Huang. *Automated GUI Test Case Generation.* In Computer Science & Service System (CSSS), 2012 International Conference on, pages 582–585. IEEE, 2012.

[Lutz 2000] Robyn R Lutz. *Software Engineering for Safety: a Roadmap.* In Proceedings of the Conference on The Future of Software Engineering, pages 213–226. ACM, 2000.

[Madani & Parissis 2009] Laya Madani et Ioannis Parissis. *Automatically Testing Interactive Applications Using Extended Task Trees.* The Journal of Logic and Algebraic Programming, vol. 78, no. 6, pages 454–471, 2009.

[Mariani *et al.* 2011] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli et Mauro Santoro. *AutoBlackTest: A Tool for Automatic Black-box Testing.* In Proceedings of the 33rd International Conference on Software Engineering, pages 1013–1015. ACM, 2011.

[Markopoulos *et al.* 1996] Panos Markopoulos, Jon Rowson et Peter Johnson. *Dialogue Modelling in the Framework of an Interactor Model.* In Pre-conference Proceedings. Design Specification and Verification of Interactive Systems. Namur, Belgium, volume 44, 1996.

[Markopoulos *et al.* 1998] Panos Markopoulos, Peter Johnson et Jon Rowson. *Formal Architectural Abstractions for Interactive Software.* Int. J. Hum.-Comput. Stud., vol. 49, no. 5, pages 675–715, Novembre 1998.

[Markopoulos 1995] Panos Markopoulos. On the Expression of Interaction Properties within an Interactor Model. Springer, 1995.

[Markopoulos 1997] Panagiotis Markopoulos. *A Compositional Model for the Formal Specification of User Interface Software.* PhD thesis, Citeseer, 1997.

[Martinie *et al.* 2014] Célia Martinie, David Navarre et Philippe Palanque. *A multi-formalism approach for model-based dynamic distribution of user interfaces of critical interactive systems.* International Journal of Human-Computer Studies, vol. 72, no. 1, pages 77–99, 2014.

[Masci *et al.* 2011] Paolo Masci, Rimvydas Ruksenas, Patrick Oladimeji, Abigail Cauchi, Andy Gimblett, Yunqiu Li, Paul Curzon et Harold Thimbleby. *On Formalising Interactive Number Entry on Infusion Pumps.* Electronic Communications of the EASST, vol. 45, 2011.

[Masci *et al.* 2013a] Paolo Masci, Anaheed Ayoub, Paul Curzon, Michael D. Harrison, Insup Lee et Harold Thimbleby. *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, pages 81–90, New York, NY, USA, 2013. ACM.

[Masci *et al.* 2013b] Paolo Masci, Anaheed Ayoub, Paul Curzon, Insup Lee, Oleg Sokolsky et Harold Thimbleby. *Model-based Development of the Generic PCA Infusion Pump User Interface Prototype in PVS.* In Proceedings of the 32Nd International Conference on Computer Safety, Reliability, and Security - Volume 8153, SAFECOMP 2013, pages 228–240, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[Masci *et al.* 2014a] Paolo Masci, Yi Zhang, Paul Jones, Paul Curzon et Harold Thimbleby. *Formal Verification of Medical Device User Interfaces Using PVS.* In Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411, pages 200–214, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[Masci *et al.* 2014b] Paolo Masci, Yi Zhang, Paul L. Jones, Patrick Oladimeji, Enrico D'Urso, Cinzia Bernardeschi, Paul Curzon et Harold Thimbleby. *Combining PVSio with Stateflow.* In NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings, pages 209–214, 2014.

[Masci *et al.* 2015] Paolo Masci, Rimvydas Ruksenas, Patrick Oladimeji, Abigail Cauchi, Andy Gimblett, Karen Yunqiu Li, Paul Curzon et Harold W. Thimbleby. *The Benefits of Formalising Design Guidelines: A Case Study on the Predictability of Drug Infusion Pumps.* ISSE, vol. 11, no. 2, pages 73–93, 2015.

[Mateescu & Oudot 2008] Radu Mateescu et Emilie Oudot. *Bisimulator 2.0: An On-the-Fly Equivalence Checker based on Boolean Equation Systems.* In Proceedings of the 6th ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2008 (Anaheim, CA, USA), pages 73–74. IEEE Computer Society Press, Juin 2008.

[Mateescu & Thivolle 2008] Radu Mateescu et Damien Thivolle. *A Model Checking Language for Concurrent Value-Passing Systems.* In Jorge Cuellar et Tom Maibaum, editeurs, FM 2008, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164, Turku, Finlande, 2008. Springer Verlag.

[Memon *et al.* 2003] Atif Memon, Ishan Banerjee et Adithya Nagarajan. *GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing.* In null, page 260. IEEE, 2003.

[Miller *et al.* 2006] Steven P. Miller, Alan C. Tribble, Michael W. Whalen et Mats Per Erik Heimdahl. *Proving the Shalls.* STTT, vol. 8, no. 4-5, pages 303–319, 2006.

[Miller *et al.* 2010] Steven P Miller, Michael W Whalen et Darren D Cofer. *Software Model Checking Takes off.* Communications of the ACM, vol. 53, no. 2, pages 58–64, 2010.

[Miller 2009] Steven P Miller. *Bridging the Gap Between Model-based Development and Model Checking.* In Tools and Algorithms for the Construction and Analysis of Systems, pages 443–453. Springer, 2009.

[Milner 1980] Robin Milner. A calculus of communicating systems, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mital & Pennathur 2004] Anil Mital et Arunkumar Pennathur. *Advanced Technologies and Humans in Manufacturing Workplaces: an Interdependent Relationship*. International Journal of Industrial Ergonomics, vol. 33, no. 4, pages 295 – 313, 2004.

[Miñón *et al.* 2014] Raúl Miñón, Lourdes Moreno, Paloma Martínez et Julio Abascal. *An Approach to the Integration of Accessibility Requirements into a User Interface Development Method*. Science of Computer Programming, vol. 6, pages 58 – 73, 2014. Special issue on Software Support for User Interface Description Languages (UIDL 2011).

[Moher *et al.* 1996] Thomas Moher, Victor Dirda et Remi Bastide. *A Bridging Framework for the Modeling of Devices, Users, and Interfaces*. Rapport technique, In [10, 1996.

[Mori *et al.* 2002] Giulio Mori, Fabio Paternò et Carmen Santoro. *CTTE: Support for Developing and Analyzing Task Models for Interactive System Design*. Software Engineering, IEEE Transactions on, vol. 28, no. 8, pages 797–813, 2002.

[Murugesan *et al.* 2013] Anitha Murugesan, Michael W. Whalen, Sanjai Rayadurgam et Mats Per Erik Heimdahl. *Compositional Verification of a Medical Device System*. In Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology, HILT 2013, Pittsburgh, Pennsylvania, USA, November 10-14, 2013, pages 51–64, 2013.

[Navarre *et al.* 2001] David Navarre, Philippe A. Palanque, Fabio Paternò, Carmen Santoro et Rémi Bastide. *A Tool Suite for Integrating Task and System Models through Scenarios*. In Proceedings of the 8th International Workshop on Interactive Systems: Design, Specification, and Verification-Revised Papers, DSV-IS '01, pages 88–113, London, UK, UK, 2001. Springer-Verlag.

[Navarre *et al.* 2005] David Navarre, Philippe Palanque, Rémi Bastide, Amélie Schyn, Marco Winckler, Luciana P. Nedel et Carla M. D. S. Freitas. *A Formal Description of Multimodal Interaction Techniques for Immersive Virtual Reality Applications*. In Proceedings of the 2005 IFIP TC13 International Conference on Human-Computer Interaction, INTERACT'05, pages 170–183, Berlin, Heidelberg, 2005. Springer-Verlag.

[Navarre *et al.* 2008] David Navarre, Philippe Palanque et Sandra Basnyat. *A formal approach for user interaction reconfiguration of safety critical interactive systems*. In Computer Safety, Reliability, and Security, pages 373–386. Springer, 2008.

[Navarre *et al.* 2009] David Navarre, Philippe A. Palanque, Jean-François Ladry et Eric Barboni. *ICOs: A Model-based User Interface Description Technique Dedicated to Interactive Systems Addressing Usability, Reliability and Scalability*. ACM Trans. Comput.-Hum. Interact., vol. 16, no. 4, 2009.

[Newcombe *et al.* 2015] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker et Michael Deardeuff. *How Amazon Web Services Uses Formal Methods*. Commun. ACM, vol. 58, no. 4, pages 66–73, Mars 2015.

[Nguyen *et al.* 2010] Duc Hoai Nguyen, Paul Strooper et Jorn Guy Suess. *Model-based Testing of Multiple GUI Variants Using the GUI Test Generator.* In Proceedings of the 5th Workshop on Automation of Software Test, pages 24–30. ACM, 2010.

[Nguyen *et al.* 2014] Bao N Nguyen, Bryan Robbins, Ishan Banerjee et Atif Memon. *GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software.* Automated Software Engineering, vol. 21, no. 1, pages 65–105, 2014.

[Nielsen & Landauer 1993] Jakob Nielsen et Thomas K Landauer. *A Mathematical Model of the Finding of Usability Problems.* In Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems, pages 206–213. ACM, 1993.

[Niwa *et al.* 2001] Yuji Niwa, Makoto Takahashi et Masaharu Kitamura. *The Design of Human–Machine Interface for Accident Support in Nuclear Power Plants.* Cognition, Technology & Work, vol. 3, no. 3, pages 161–176, 2001.

[Oladimeji *et al.* 2011] Patrick Oladimeji, Harold W. Thimbleby et Anna Louise Cox. *Number Entry Interfaces and Their Effects on Error Detection.* In Human-Computer Interaction - INTERACT 2011 - 13th IFIP TC 13 International Conference, Lisbon, Portugal, September 5-9, 2011, Proceedings, Part IV, pages 178–185, 2011.

[Oladimeji *et al.* 2013] Patrick Oladimeji, Harold W. Thimbleby et Anna Louise Cox. *A Performance Review of Number Entry Interfaces.* In Human-Computer Interaction - INTERACT 2013 - 14th IFIP TC 13 International Conference, Cape Town, South Africa, September 2-6, 2013, Proceedings, Part I, pages 365–382, 2013.

[Oliveira *et al.* 2014] Raquel Oliveira, Sophie Dupuy-Chessa et Gaëlle Calvary. *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, pages 235–240, New York, NY, USA, 2014. ACM.

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

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

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

[Olsen Jr 2007] Dan R Olsen Jr. *Evaluating User Interface Systems Research.* In Proceedings of the 20th annual ACM symposium on User interface software and technology, pages 251–258. ACM, 2007.

[Palanque & Bastide 1995] Philippe A Palanque et Remi Bastide. *Petri Net based Design of User-driven Interfaces Using the Interactive Cooperative Objects Formalism.* In Interactive systems: Design, specification, and verification, pages 383–400. Springer, 1995.

[Palanque *et al.* 1996] Philippe A. Palanque, Rémi Bastide et V. Sengès. *Validating Interactive System Design through the Verification of Formal Task and System Models.* In Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction, pages 189–212, London, UK, UK, 1996. Chapman & Hall, Ltd.

[Palanque *et al.* 1997] Philippe A. Palanque, Rémi Bastide et Fabio Paternò. *Formal Specification As a Tool for Objective Assessment of Safety-critical Interactive Systems.* In Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction, INTERACT '97, pages 323–330, London, UK, UK, 1997. Chapman & Hall, Ltd.

[Palanque *et al.* 1999] Philippe A. Palanque, Christelle Farenc et Rémi Bastide. *Embedding Ergonomic Rules As Generic Requirements in a Formal Development Process of Interactive Software.* In Human-Computer Interaction INTERACT '99: IFIP TC13 International Conference on Human-Computer Interaction, Edinburgh, UK, 30th August-3rd September 1999, pages 408–416, 1999.

[Park 1981] David Park. *Concurrency and Automata on Infinite Sequences.* In Proceedings of the 5th GI-Conference on Theoretical Computer Science, pages 167–183, London, UK, UK, 1981. Springer-Verlag.

[Paternò & Faconti 1992] Fabio Paternò et G Faconti. *On the Use of LOTOS to Describe Graphical Interaction.* People and computers, pages 155–155, 1992.

[Paternó & Mezzanotte 1994] Fabio Paternó et Menica Mezzanotte. *Analysing MATIS by Interactors and ACTL.* The AMODEUS Project—ESPRIT BRA 7040, Report SM/WP 36, 1994.

[Paternò & Mezzanotte 1996] Fabio Paternò et M. Mezzanotte. *Formal Verification of Undesired Behaviours in the CERD Case Study.* In Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction, pages 213–226, London, UK, UK, 1996. Chapman & Hall, Ltd.

[Paternò & Santoro 2001] Fabio Paternò et Carmen Santoro. *Integrating Model Checking and HCI Tools to Help Designers Verify User Interface Properties.* In Proceedings of the 7th International Conference on Design, Specification, and Verification of Interactive Systems, DSV-IS'00, pages 135–150, Berlin, Heidelberg, 2001. Springer-Verlag.

[Paternò & Santoro 2003] Fabio Paternò et Carmen Santoro. *Support for Reasoning about Interactive Systems through Human-computer Interaction Designers' Representations.* Comput. J., vol. 46, no. 4, pages 340–357, 2003.

[Paternò *et al.* 1997] Fabio Paternò, Cristiano Mancini et Silvia Meniconi. *ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models.* In Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction, INTERACT '97, pages 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd.

[Paterno 1993] Fabio Paterno. *A Formal Specification of Appearance and Behaviour of Visual Environments.* Software Engineering Journal, vol. 8, no. 3, pages 154–164, 1993.

[Paternó 1994] F. Paternó. *A Theory of User-interaction Objects.* Journal of Visual Languages & Computing, vol. 5, no. 3, pages 227 – 249, 1994.

[Paternó 1997] F. Paternó. *Formal Reasoning about Dialogue Properties with Automatic Support.* Interacting with Computers, vol. 9, no. 2, pages 173–196, 1997.

[Peterson 1981] James Lyle Peterson. Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Preece *et al.* 1994] Jenny Preece, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland et Tom Carey. Human-computer Interaction. Addison-Wesley Longman Ltd., 1994.

[Queille & Sifakis 1982] Jean-Pierre Queille et Joseph Sifakis. *Specification and Verification of Concurrent Systems in CESAR.* In Proceedings of the 5th Colloquium on International Symposium on Programming, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[Queille & Sifakis 1983] Jean-Pierre Queille et Joseph Sifakis. *Fairness and Related Properties in Transition Systems – A Temporal Logic to Deal with Fairness.* Acta Informatica, vol. 19, no. 3, pages 195–220, 1983.

[Rushby & von Henke 1993] John M. Rushby et Friedrich W. von Henke. *Formal Verification of Algorithms for Critical Systems.* IEEE Trans. Software Eng., vol. 19, no. 1, pages 13–23, 1993.

[Rushby *et al.* 1999] John Rushby, Judith Crow et Everett Palmer. *An Automated Method to Detect Potential Mode Confusions.* In Digital Avionics Systems Conference, 1999. Proceedings. 18th, volume 1, pages 4–B. IEEE, 1999.

[Rushby 2001] John M. Rushby. *Analyzing Cockpit Interfaces Using Formal Methods.* Electr. Notes Theor. Comput. Sci., vol. 43, pages 1–14, 2001.

[Rushby 2002] John Rushby. *Using Model Checking to Help Discover Mode Confusions and Other Automation Surprises.* Rel. Eng. & Sys. Safety, vol. 75, no. 2, pages 167–177, 2002.

[Serna *et al.* 2010] Audrey Serna, Gaëlle Calvary et Dominique Scapin. *How Assessing Plasticity Design Choices Can Improve UI Quality: A Case Study.* In Proceeding of the second ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010), pages 29–34, Berlin, Germany, 2010. ACM Press. June 19-23, 2010.

[Shiffman *et al.* 2004] Smadar Shiffman, Asaf Degani et Michael Heymann. *Ui Verify – A Web-based Tool for Verification and Automatic Generation of User Interfaces.* interfaces, vol. 3, page 4, 2004.

[Sighireanu *et al.* 2004] Mihaela Sighireanu, Claude Chaudet, Hubert Garavel, Marc Herbert, Radu Mateescu et Bruno Vivien. *LOTOS NT User Manual.* INRIA, june, 2004.

[Silva *et al.* 2007] J. C. Silva, José Creissac Campos et João Saraiva. *Combining Formal Methods and Functional Strategies Regarding the Reverse Engineering of Interactive Applications.* In Proceedings of the 13th International Conference on Interactive Systems: Design, Specification, and Verification, DSVIS'06, pages 137–150, Berlin, Heidelberg, 2007. Springer-Verlag.

[Sottet *et al.* 2006] Jean-Sebastien Sottet, Gaëlle Calvary et Jean-Marie Favre. Mapping Model: A First Step to Ensure Usability for Sustaining User Interface Plasticity. no, 2006.

[Sottet *et al.* 2007] Jean-Sebastien Sottet, Gaëlle Calvary, Joëlle Coutaz, Jean-Marie Favre, Jean Vanderdonckt, Adrian Stanciulescu et Sophie Lepreux. *A Language Perspective on the Development of Plastic Multimodal User Interfaces.* Journal of Multimodal User Interfaces, vol. 1, no. 2, 2007.

[Sousa *et al.* 2014] Manuel Sousa, J Campos, Miriam Alves, M Harrison*et al. 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), pages 62–67, 2014.

[Spivey 1989] J. M. Spivey. The Z Notation: A Reference Manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Spring 2002] Michael B. Spring. *Interactive Systems.* `http://www.encyclopedia.com/doc/1G2-3401200080.html`, 2002. Computer Sciences. [Accessed: 2015-06-15].

[Strunk *et al.* 2005] Elisabeth A. Strunk, Xiang Yin et John C. Knight. *ECHO: A Practical Approach to Formal Verification.* In Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems, FMICS '05, pages 44–53, New York, NY, USA, 2005. ACM.

[Thevenin & Coutaz 1999] David Thevenin et Joëlle Coutaz. *Plasticity of User Interfaces: Framework and Research Agenda.* In Proceedings of INTERACT, volume 99, pages 110–117, 1999.

[Thimbleby & Gimblett 2011] Harold W. Thimbleby et Andy Gimblett. *Dependable Keyed Data Entry for Interactive Systems.* ECEASST, vol. 45, 2011.

[Thimbleby & Gow 2008] Harold Thimbleby et Jeremy Gow. *Applying Graph Theory to Interaction Design.* In Jan Gulliksen, Morton Borup Harning, Philippe Palanque, Gerrit C. Veer et Janet Wesson, editeurs, Engineering Interactive Systems, pages 501–519. Springer-Verlag, Berlin, Heidelberg, 2008.

[Thimbleby 2007a] Harold Thimbleby. *Interaction Walkthrough: Evaluation of Safety Critical Interactive Systems.* In Interactive Systems. Design, Specification, and Verification, pages 52–66. Springer, 2007.

[Thimbleby 2007b] Harold Thimbleby. *User-centered Methods Are Insufficient for Safety Critical Systems.* In Proceedings of the 3rd Human-computer Interaction and Usability Engineering of the Austrian Computer Society Conference on HCI and Usability for Medicine and Health Care, USAB'07, pages 1–20, Berlin, Heidelberg, 2007. Springer-Verlag.

[Thimbleby 2010] Harold Thimbleby. *Think! Interactive Systems Need Safety Locks.* CIT. Journal of Computing and Information Technology, vol. 18, no. 4, pages 349–360, 2010.

[Tsai *et al.* 2000] WT Tsai, X Bai, B Huang, G Devaraj et R Paul. *Automatic Test Case Generation for GUI Navigation.* In Quality Week, volume 2000, 2000.

[Tu *et al.* 2014] Huawei Tu, Patrick Oladimeji, Karen Yunqiu Li, Harold W. Thimbleby et Chris Vincent. *The Effects of Number-related Factors on Entry Performance.* In BCS-HCI 2014 Proceedings of the 28th International BCS Human Computer Interaction Conference, Southport, UK, 9-12 September 2014, 2014.

[Turchin & skiĭ 2006] P. Turchin et Rossiĭ skiĭ. History and Mathematics. History & mathematics. URSS, 2006.

[Turner 1993] Clark S Turner. *An Investigation of the Therac-25 Accidents.* COMPUTER, vol. 18, no. 9I62/93, pages 0700–001830300, 1993.

[van Glabbeek & Weijland 1996] Rob J. van Glabbeek et W. Peter Weijland. *Branching Time and Abstraction in Bisimulation Semantics.* Journal of the ACM, vol. 43, no. 3, pages 555–600, 1996.

[Vanderdonckt *et al.* 2008] Jean Vanderdonckt, Gaëlle Calvary, Joëlle Coutaz et Adrian Stanciulescu. Multimodality for Plastic User Interfaces: Models, Methods, and Principles, chapitres d'ouvrages 4, pages 61–84. Springer, 2008. D. Tzovaras (ed.), Lecture Notes in Electrical Engineering, Springer-Verlag, Berlin, 2007.

[Vanderdonckt 1994] Jean Vanderdonckt. Guide Ergonomique des Interfaces Homme-machine. Numeéro 13 de Collection "Travaux de l'Institut d'Informatique". Presses Universitaires, Namur, 1994.

[Wang & Abowd 1994] Hung-Ming Wang et Gregory Abowd. *A Tabular Interface for Automated Verification of Event-based Dialogs.* Rapport technique, DTIC Document, 1994.

[Whalen *et al.* 2008] Michael Whalen, Darren Cofer, Steven Miller, Bruce H Krogh et Walter Storm. *Integration of Formal Analysis into a Model-based Software Development Process.* In Formal Methods for Industrial Critical Systems, pages 68–84. Springer, 2008.

[White & Almezen 2000] Lee White et Husain Almezen. *Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences.* In Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on, pages 110–121. IEEE, 2000.

[Worldgrid 2011a] Atos Worldgrid. *Modernizing Data Processing for EDF Energy at Dungeness – Improving Performance and Standardizing Technology for EDF Energy*, 2011.

[Worldgrid 2011b] Atos Worldgrid. *ADACS-N^{TM}: Monitoring and Control of Nuclear Power Plants*, 2011.

[Yamine *et al.* 2005] Aït-Ameur Yamine, Aït-Sadoune Idir et Baron Mickaël. *Modélisation et Validation formelles d'IHM : LOT 1 (LISI/ENSMA).* Rapport technique, LISI/ENSMA, 2005 2005.

[Yin & Knight 2010] Xiang Yin et John C. Knight. *Formal Verification of Large Software Systems*. In Second NASA Formal Methods Symposium - NFM 2010, Washington D.C., USA, April 13-15, 2010. Proceedings, pages 192–201, 2010.

[Yin *et al.* 2008] Xiang Yin, John C. Knight, Elisabeth A. Nguyen et Westley Weimer. *Formal Verification by Reverse Synthesis*. In Computer Safety, Reliability, and Security, 27th International Conference, SAFECOMP 2008, Newcastle upon Tyne, UK, September 22-25, 2008, Proceedings, pages 305–319, 2008.

[Yin *et al.* 2009a] Xiang Yin, John Knight et Westley Weimer. *Exploiting Refactoring in Formal Verification*. In Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on, pages 53–62. IEEE, 2009.

[Yin *et al.* 2009b] Xiang Yin, John C. Knight et Westley Weimer. *Exploiting Refactoring in Formal Verification*. In Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009, pages 53–62, 2009.

[Yoshikawa 2005] Hidekazu Yoshikawa. *Human-machine Interaction in Nuclear Power Plants*. Nuclear Engineering and Technology, vol. 37, no. 2, page 151, 2005.

Abstract

Plasticity provides users with different versions of a UI. Although it enhances UI capabilities, plasticity adds complexity to the development of user interfaces: the consistency between multiple versions of a given UI should be ensured. This complexity is further increased when it comes to UIs of safety-critical systems. Safety-critical systems are systems in which a failure has severe consequences. Given the large number of possible versions of a UI, it is time-consuming and error prone to check these requirements by hand. Some automation must be provided to verify plasticity. Formal verification provides a rigorous way to perform verification, which is suitable for safety-critical systems. Our main contribution is an approach to verifying safety-critical interactive systems provided with plastic UIs using formal methods. Using a powerful tool support, our approach permits: (1) the verification of sets of properties over a model of the system. Using model checking, our approach permits the verification of properties over the system formal specification. Usability properties verify whether the system follows ergonomic properties to ensure a good usability. Validity properties verify whether the system follows the requirements that specify its expected behavior; (2) the comparison of different versions of UIs. Using equivalence checking, our approach verifies to which extent UIs present the same interaction capabilities and appearance. We can show whether two UI models are equivalent or not. When they are not equivalent, the UI divergences are listed, thus providing the possibility of leaving them out of the analysis. We also present three industrial case studies in the nuclear power plant domain to which the approach was applied.

Résumé

La plasticité fournit aux utilisateurs différentes versions d'une interface utilisateur. Bien qu'elle enrichisse les interfaces utilisateur, la plasticité complexifie leur développement: la cohérence entre plusieurs versions d'une interface donnée devrait être assurée. Cette complexité est accentuée quand il s'agit de systèmes critiques. Les systèmes critiques sont des systèmes dans lesquels une défaillance a des conséquences graves. Étant donné le grand nombre de versions possibles d'une interface utilisateur, il est coûteux de vérifier ces exigences à la main. Des automatisations doivent être alors fournies afin de vérifier la plasticité. La vérification formelle fournit un moyen d'effectuer une vérification rigoureuse, qui est adaptée pour les systèmes critiques. Notre principale contribution est une approche de vérification des systèmes interactifs critiques et plastiques à l'aide de méthodes formelles. Avec l'utilisation d'un outil performant, notre approche permet: (1) la vérification d'ensembles de propriétés sur un modèle du système. Reposant sur la technique de "model checking", notre approche permet la vérification de propriétés sur la spécification formelle du système. Les propriétés d'utilisabilité permettent de vérifier si le système suit de bonnes propriétés ergonomiques. Les propriétés de validité permettent de vérifier si le système suit les exigences qui spécifient son comportement attendu; (2) la comparaison des différentes versions du système. Reposant sur la technique "d'équivalence checking", notre approche vérifie dans quelle mesure deux interfaces utilisateur offrent les mêmes capacités d'interaction et la même apparence. Nous pouvons ainsi montrer si deux modèles d'une interface utilisateur sont équivalents ou non. Dans le cas où ils ne sont pas équivalents, les divergences de l'interface utilisateur sont listées, offrant ainsi la possibilité de les sortir de l'analyse. Nous présentons également trois études de cas industriels dans le domaine des centrales nucléaires dans lesquelles l'approche a été appliquée.