HAL
open science

Automatic testing of Lustre/SCADE programs

Virginia Papailiopoulou

To cite this version:

Virginia Papailiopoulou. Automatic testing of Lustre/SCADE programs. Computer Science [cs]. Université Joseph-Fourier - Grenoble I, 2010. English. NNT : . tel-00454409

HAL Id: tel-00454409

https://theses.hal.science/tel-00454409

Submitted on 8 Feb 2010

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
Ecole Doctorale MSTII
Mathématiques, Sciences et Technologies de l'Information, Informatique

Test automatique de programmes Lustre/SCADE

Thèse

présentée par

Virginia PAPAILIOPOULOU

en vue de l'obtention du grade de

Docteur de l'Université De Grenoble
Discipline Informatique

Directeurs: Farid Ouabdesselam et Ioannis Parissis

soutenue le 04 Février 2010 devant le jury composé de :

M. Philippe Lalanda	Professeur Université Joseph Fourier	Président
Mme. Odile Laurent	Airbus France	Examinateur
M. Yves Le Traon	Professeur Université de Luxembourg	Rapporteur
Mme. Virginie Wiels	Chercheur ONERA/Cert	Rapporteur
M. Farid Ouabdesselam	Professeur Université Joseph Fourier	Examinateur
M. Ioannis Parissis	Professeur Grenoble INP-Esisar	Examinateur

Automatic testing of LUSTRE/SCADE programs

Abstract

The design of embedded systems in safety-critical domains is a demanding task. During the last decades, formal methods and model-based approaches are widely used for the design, the analysis, the implementation and testing of major industrial applications.

The work in this thesis addresses the improvement of the testing process with a view to automating test data generation as well as its quality evaluation, in the framework of reactive synchronous systems specified in the LUSTRE/SCADE language. The proposed testing methods use two different requirement-based models of the program under test.

On the one hand, we present a testing methodology using the LUTESS tool that automatically generates test input data based exclusively on the environment description of the system under test. In particular, this environment description is derived from the system informal requirements and represents a *test model* of the program describing its functional behavior. Several test models can be built for the same program simulating certain system failures or different situations that must be tested. Test models are then used to guide the test data generation process towards meaningful and valid test input sequences with regard to the specifications.

On the other hand, among the large set of coverage criteria that have been proposed in the last years, we study the extension of a coverage assessment technique especially dedicated to LUSTRE/SCADE applications, using LUSTRUCTU, a prototype tool for coverage measurement. In this case, we are based on the *SCADE model* of the program under test built from the system requirements. The coverage criteria are defined directly on the LUSTRE/SCADE specifications and coverage is measured on the SCADE model rather than on the generated C code. The extended criteria take into account two new aspects: the use of multiple clocks in a LUSTRE program as well as integration testing as opposed to unit testing, allowing the coverage measurement of large-sized systems.

These two strategies targeting, respectively, the test data generation process and the coverage evaluation are tailored to industrial needs and could have a positive impact in effectively testing real-world applications. Case studies extracted from the avionics domain are used to demonstrate the application of these methods as well as to empirically evaluate their performance and complexity.

Test automatique de programmes LUSTRE/SCADE

Résumé

La conception et le développement des systèmes embarqués est une tâche difficile et exigeante. Pendant les dernières années, les méthodes formelles et les approches basées sur des modèles ont été largement utilisées pour la conception et le test des applications industrielles importantes. Ce travail porte sur l'amélioration du processus de test. Son objectif est d'offrir des moyens d'automatiser la génération des données de test ainsi que l'évaluation de leur qualité, dans le cadre des systèmes réactifs synchrones spécifiés en LUSTRE/SCADE. Les méthodes de test proposées s'appuient sur deux modèles différents du système sous test qui sont construits à partir des exigences fonctionnelles.

D'une part, nous présentons une méthodologie de test basée sur l'outil LUTESS qui génère automatiquement des données de test exclusivement à partir de la description de l'environnement du système sous test. En particulier, cette description de l'environnement vient des besoins informels du système, en représentant un *modèle de test* qui décrit le comportement fonctionnel du système. Plusieurs modèles de test peuvent être construits pour le même programme simulant certaines défaillances du système ou différentes situations qui doivent être examinées. Les modèles de test sont alors utilisés pour guider la procédure de génération des tests vers des séquences de test significatives par rapport aux spécifications.

D'autre part, parmi la grande gamme des critères de couverture qui ont été proposés ces dernières années, nous étudions l'extension d'une technique d'évaluation de la couverture spécialement dédiée aux applications LUSTRE/SCADE, en utilisant LUSTRUCTU, un outil de mesure de la couverture à l'état de prototype. Dans ce cas, on se base sur le *modèle SCADE* du programme sous test qui est construit à partir des exigences fonctionnelles. Les critères de couverture sont directement définis sur les spécifications LUSTRE/SCADE et la couverture est mesurée sur le modèle SCADE plutôt que sur le code C généré. Les critères étendus prennent en compte deux nouveaux aspects: l'utilisation de plusieurs horloges dans un programme LUSTRE aussi bien que le test d'intégration par opposition au test unitaire, permettant la mesure de couverture de systèmes de grande taille.

Ces deux stratégies visant, respectivement, à la procédure de génération automatique de tests et l'évaluation de leur couverture ont été conçues en fonction

des besoins industriels et pourraient avoir un impact positif sur le test efficace des applications réelles. Des études de cas extraites du domaine de l'avionique sont employées pour démontrer l'applicabilité de ces méthodes et pour évaluer leur complexité.

Acknowledgments

I would like to thank all the people who, in different ways, have helped and inspired me during the course of my PhD and without whom this dissertation would be simply impossible.

I am deeply indebted to my supervisors, Ioannis Parissis and Farid Ouabdesselam, for their kindness and willingness to help me out throughout the last three years and especially the first few months after I came to France. Their guidance and helpful advices as well as their patience and faith in me have constantly helped me feel positive and confident.

I would like to acknowledge all members of the ANR project SIESTA for the precious interactive work and the productive collaboration during our project meetings. I particularly express my appreciation to Lydie Du Bousquet for her assistance, her insightful comments, suggestions and contribution to this work.

Ms. Virginie Wiels and Mr.Yves Le Traon deserve a special thanks for having accepted to review this thesis. I am also thankful to M. Philippe Lalanda and Ms. Odile Laurent for having accepted to participate in the dissertation committee.

I would also like to thank my colleagues and friends for their support and friendship. I especially thank Laya, Besnik and Ajitha for the fruitful discussions that we shared and their inspiring advices.

My deepest gratitude goes to my mother and my sister, my family and my closest friends Fotini and Niki for always being there for me despite of the long distance separating us. Their love and continuous support have always motivated me and helped me overcome all difficulties that came along the way. I am finally grateful to Michali for his understanding and patience, specially through the hard time of writing this thesis.

"If we knew what it was we were doing, it would not be called research, would it?"

Albert Einstein

Contents

Abstract **ii**

Résumé **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Research background 1
1.2 Motivations and objectives 3
1.3 Contributions 6
1.4 Thesis outline 8

I Software testing **17**

2 Testing: basic concepts and techniques **18**

2.1 Testing process 19
2.2 Testing techniques 20
2.2.1 Black-box testing 21
2.2.2 White-box testing 22
2.3 Quality evaluation 22
2.3.1 Control-flow-based coverage 24
2.3.2 Data-flow-based coverage 27

3 Testing Lustre programs **29**

3.1 Reactive software 29
3.2 The synchronous approach 30
3.3 LUSTRE overview 31
3.3.1 Compilation 33

3.3.2 Operator network 34
3.3.3 Clocks in LUSTRE 34
3.4 Synchronous software development 36
3.5 Test data generation 39
3.5.1 LUTESS 40
3.5.2 GATeL 42
3.6 Structural coverage of LUSTRE programs 44
3.6.1 Paths and activation conditions 44
3.6.2 Coverage criteria 47
3.6.3 LUSTRUCTU 48
3.6.4 SCADE MTC 48
3.7 Requirements-based testing 49
3.8 Objectives of this thesis 50

II Extended structural test coverage criteria 52

4 Multiple clocks 53
4.1 The use of multiple clocks 54
4.2 Activation conditions for `when` and `current` 54
4.3 Example: a counter 57
4.4 SCADE MTC 60

5 Integration testing 61
5.1 Node integration 61
5.2 Path length w.r.t. temporal loops 63
5.3 Path activation conditions in integration testing 63
5.4 Extended structural coverage criteria 65
5.4.1 Integration depth 65
5.4.2 Path length 66
5.4.3 Criteria definition 67
5.5 Subsumption relations 68

6 Experimental evaluation 74
6.1 Case study: an alarm management software 74
6.1.1 Objectives 74
6.2 System description 75
6.3 Number of paths 79

6.4 Testing effort w.r.t. criteria satisfaction 83
6.5 Fault detection capability 85
6.6 Concluding remarks 88

III Test generation 89

7 Automatic test data generation with Lutess 90
7.1 Testing methodology 91
7.2 The steam-boiler controller 93
7.2.1 System specification 93
7.2.2 Modeling and testing the boiler 96
7.2.2.1 Domain definitions 96
7.2.2.2 Environment dynamics 97
7.2.2.3 Test scenarios 99
7.2.2.4 Property-based testing 101
7.2.3 Remarks on the problem size 101
7.3 The land gear controller 102
7.3.1 System specification 102
7.3.2 The control software 104
7.3.3 Modeling the land gear controller 106
7.3.3.1 Domain definitions 106
7.3.3.2 Environment dynamics 107
7.3.3.3 Test scenarios 107
7.3.3.4 Property-based testing 111
7.3.4 Evaluation aspects 112
7.3.5 Remarks on the problem size 114
7.4 Concluding remarks 115

8 Conclusions and future work 117
8.1 Conclusion 117
8.2 Perspectives 119

List of Figures

2.1 General testing procedure. 20
2.2 Program calculating the greatest common divisor (`gcd(a,b)`). . . 23
2.3 Control-flow graph. 24
2.4 An example for decision coverage. 25

3.1 Synchronous software operation 31
3.2 Example of a LUSTRE node. 32
3.3 The operator network for the node `Never`. 34
3.4 An example using two clocks (the basic clock and the flow c) and the corresponding operator network. 36
3.5 Development process in avionics. 37
3.6 Current testing process in avionics. 38
3.7 The Lutess testing environment. 40
3.8 General form of a testnode syntax. 41
3.9 Activation condition for the path $\langle a, e \rangle$. 45

4.1 Modeling the `when` and `current` operators using the `if-the-else`. 55
4.2 The node `TIME_STABLE`: a simple example with the `when` and `current` operators. 58
4.3 The operator network for the node `TIME_STABLE`. 59

5.1 Example of a complex LUSTRE program. 62
5.2 A LUSTRE node using a compound operator (global level: `NODE0` abstracted). 64
5.3 Expansion of `NODE0` in Figure 5.2. 64
5.4 Structure of a large application.. 66

5.5 Operator network for the node `WD2` (level 0) and the called nodes `WD1` and `EDGE` (level 1). 71

6.1 Call graph for the alarm management software component. . . . 76
6.2 Operator network of the alarm management system. 77
6.3 Call graph for node #2. 79
6.4 Operator network of node #2 (the called nodes are expanded). . 80
6.5 Operator network of node #2 (the called nodes are abstracted). . 80
6.6 Coverage ratio for test sequence length 1 to 1000. 85
6.7 Coverage ratio VS Mutation score (test sequence length 1-1000). 87

7.1 General form of a testnode syntax. 92
7.2 The steam boiler control system. 93
7.3 Operational modes of the steam-boiler system. 94
7.4 The landing gear system. 103

8.1 New feature in SCADE V6. 120

List of Tables

3.1 The use of the operators `when` and `current`. 35
3.2 Activation conditions for all LUSTRE operators. 46

4.1 Test cases samples for the input m. 57

6.1 Size and complexity of the alarm management system. 78
6.2 Number of paths in node #2 w.r.t. the number of cycles. 82
6.3 Number of activation conditions (M) w.r.t. the criteria power. . 82
6.4 Number of activation conditions for node #2. 82
6.5 Coverage ratio for test sequences of length 1 to 10. 83
6.6 Coverage ratio for test sequences of length 10 to 100. 84
6.7 Coverage ratio for test sequences of length 100 to 1000. 84
6.8 Set of mutant operators. 86
6.9 Coverage ratio VS Mutation score (test sequence length 1-10). . . 86
6.10 Coverage ratio VS Mutation score (test sequence length 10-100). 86
6.11 Coverage ratio VS Mutation score (test sequence length 100-1000). 87

7.1 Excerpt of a test case using all the possible invariant properties. 99
7.2 Excerpt of a test case with broken level device scenario. 101
7.3 Excerpt of a test case guided by a safety property. 101
7.4 Excerpt of a test case using only invariant properties. 108
7.5 Excerpt of a test case using the scenarios specified in Section 7.3.3.3. 109
7.6 Excerpt of a test case using scenario concerning the gear up failure specified in Section7.3.3.3. 110
7.7 Excerpt of a test case guided by a safety property. 111
7.8 Excerpt of a test case guided by a safety property and certain hypotheses on the system. 112

7.9 Excerpt of a test case with oracle violation. 113
7.10 Excerpt of a test case with oracle violation simulating a failure in the up gear. 114

Chapter 1

Introduction

In any software development project, from the phase of design through the final phase of production, software is tested and inspected in order to ensure that it satisfies the requirements and that the implementation corresponds correctly to the specification. Especially in safety-critical applications, in which a possible failure could cost human lives or severe damage to equipment or environment, dependability [34, 52] is one of the most important factors that allows a service to be reliable. This property can be reassured by means of verification and validation [66], two significant activities during software development which mainly aim at eliminating errors. Elimination of errors can be achieved, among other methods, through software testing, which mainly consists in executing the program under test over a test data set and verifying if the results match the expected ones. The topic of this thesis lies mainly in this context. More precisely, we focus on synchronous programs written in the Lustre language, a particular class of reactive systems, and we study the improvement of testing in this area with a special interest in the avionics field.

1.1 Research background

In high-risk applications in such domains as avionics, automotive and energy, systems are normally of type command and control. Such systems must be well synchronized with their external environment in order to prevent any malfunction or even failure with undesirable consequences. In fact, software response to the information supplied by the environment should be fast enough (theoreti-

cally instantaneous) so that any change in the latter can be taken into account. This property is termed as *synchronism* and characterizes synchronous reactive systems that are usually part of safety-critical applications. With a view to properly modeling and specifying this kind of systems, several synchronous languages have been proposed as such Esterel [9], Signal [36] and LUSTRE [22, 7]. In the context of this thesis, we are mainly concerned with LUSTRE.

LUSTRE is a declarative, data-flow language that contains a temporal logic of the past, allowing to refer to the past values of an expression. Variables and expressions are represented by data flows, that is, infinite sequences of values whose evaluation is determined by a discrete global clock. A LUSTRE program describes the way that input data flows are transformed into output flows at each instant of the global clock. In addition to this cyclic behavior, the deterministic nature of the language and its formal semantics make it ideal for programming reactive synchronous systems with similar features. The structure of a LUSTRE program is graphically represented by an operator network. An operator network is an oriented graph; the nodes denote the operators used in the program and the edges connecting the operators denote the transfer of data among them.

LUSTRE is the backbone of SCADE (Safety Critical Application Development Environment) [2], a graphical tool dedicated to the development of critical embedded systems and often used by industries and professionals. This design environment allows the hierarchical definition of the system components, their graphical simulation and verification as well as the automatic code generation. From SCADE functional specifications, C code is automatically generated, though this transformation (SCADE to C) is not standardized. This graphical modeling tool-suite is used mainly in the field of avionics (Airbus, DO-178B standard) and is becoming a de-facto standard in this field.

The design of embedded systems in safety-critical domains is a demanding task. During the last decades, formal methods and model-based approaches have been widely used for the design, the analysis and the implementation of major industrial applications. All along the development cycle of such systems, the verification and validation process is a crucial, let alone time and cost-consuming part. Specifically, testing is a mean of dynamic verification and validation that could reveal faults in a system and ensure that the model is in conformance with the implementation and sometimes it could provide a sense of confidence in the final product.

The aim of this thesis is to study the improvement of the testing process with

a view to automating test data generation as well as their quality evaluation, in the framework of reactive synchronous systems specified in the LUSTRE/SCADE language. This research work is conducted within the framework of the SIESTA project[1] (**A**utomatisation du **T**est de **S**ystèmes **E**mbarqués réalisés en SCADE et **Si**mulink), funded by ANR, the French National Research Foundation. The project brings together academic and industrial partners around a common issue: to optimize the development costs of SCADE applications, facing the reality of the industry and taking into account the concepts inadequately treated so far.

1.2 Motivations and objectives

In general, the testing process involves primarily the test data selection, then the execution of the program under test on the selected test data and finally the evaluation of the results in order to identify the differences between the obtained and the expected results and also make sure that the system is sufficiently tested. In this context, there are two core issues of particular interest. The first one lies in the phase of test data selection. Indeed, one of the most important parts in testing a program is the design and the creation of effective test cases. Testing a program exhaustively would be ideal because the program would be tested over all the possible states and all the errors would be theoretically detected. However, this is not feasible due to the important number of input values that the program should be subjected to. Even for small and simple programs, this number might be very big even infinite. Therefore, since complete testing is impossible, the question is to suitably select the most complete test data set with the highest probability of detecting most of the errors. Given the constraints on time and cost, this is not as easy a task as it may seem. On the other hand, the second core issue concerns the evaluation of selected test data. The absence of errors does not always mean that the program is well implemented and error-free. Actually, the principal goal of testing should be the exact opposite [41]. Hence, when a test data reveals no errors in a program, this usually suggests either that the specific test data set is not the appropriate one or that the program is not adequately tested. This is why this step is equally important since not only it provides a measure of the program coverage but also it often determines whether the testing process can stop.

[1] www.siesta-project.com

There has been extensive study and research on these two topics and numerous methods and strategies have been proposed with the main purpose of improving the program testing procedure. These solutions are usually based on certain rules, properties or hypotheses that a test case must satisfy. When such properties are formally defined they are called *criteria*. *Selection criteria* are used during the test data selection phase while *adequacy* or *stopping criteria* are used during the quality evaluation phase to decide whether the program is adequately tested. These criteria are commonly referred to as coverage metrics with regard to the program code (code coverage), the program requirements (requirements coverage) or the percentage of detected faults (fault coverage) [10, 15].

As far as testing of LUSTRE programs is concerned, several testing tools [16, 37, 57] have been developed to make test data generation more profitable and valuable. They are based either on the functional specification or the structural details of the program under test. GATeL [37] is a structural testing tool based on constraint logic programming that generates test cases by solving a problem of constraints on program variables. LUTESS [17, 49], conversely, is a functional testing tool that generates test cases on the fly according to a set of constraints imposed on the program environment. Various testing strategies are supported, as random testing [45, 51, 57], scenarios-guided testing [46, 69] or safety-property-guided testing [63]. Recently, the tool has been entirely redesigned [61] using constraint logic programming in order to handle numerical variables and make hypotheses on the program under test that could improve test data generation. Despite the progress achieved in test data generation techniques and tools, this process is not fully automated and rather frequently engineers have to manually produce test cases with respect to program requirements. Admittedly, this is a quite difficult, let alone expensive task that usually takes weeks or even months to be completed. Specifying a complete and precise testing methodology based on LUTESS that would be tailored to industrial needs could have a positive impact in effectively testing real-world applications. One of the objectives of this research work is to examine and assess the applicability and the effectiveness of such an approach on a case study from the avionics domain.

As for the quality evaluation part, various structural coverage criteria have been proposed for imperative languages. They are mostly based on the source code and they are defined on the control flow graph that represents the transfer of control throughout the program execution. Some typical examples are path

coverage, statement coverage and branch coverage [41]. The first one consists in executing at least once every path in the program control flow graph. This is almost always impossible because of the number of paths which may be infinite in case that the program contains many loops. On the contrary, statement or branch coverage require that each statement or branch, respectively, is executed at least once during testing. Such criteria may achieve lower coverage ratios but they are more practical, thus more preferable in real software testing [4]. There are also several structural criteria based on the decomposition of branch boolean expressions in decisions and conditions[2] [41, 64]. The modified condition-decision coverage criterion (MC/DC) [13] is the one used mostly, especially in safety-critical software. Moreover, there is a family of structural criteria based on the way that data flows all along the program execution [55]. These criteria consider the definitions and the uses of the variables contained in a program. A few works [39, 49] have addressed in the past the coverage of Lustre programs but these coverage metrics are not useful nor sufficiently powerful.

Nevertheless, none of the above coverage metrics can be directly applied to Lustre code. In particular, in major industrial applications in the field of avionics, testing of safety-critical parts of software is performed according to the DO-178B standard [1], which demands full MC/DC coverage by generating test cases based on requirements. This implies that coverage metrics are applied to the C code generated from Lustre/SCADE functional specification. For an application specified and designed in the Lustre/SCADE environment, C code coverage measurement does not totally correspond to the actual coverage of the system requirements due to the lack of any formal relation between those two. The compilation method from Lustre to C code is not yet standardized and Lustre/SCADE engineers find it hard to link C code coverage with Lustre functional specification coverage and thus with requirements.

To deal with this problem, especially designed structural coverage criteria for Lustre/SCADE programs have been proposed that are explicitly conformed with the synchronous paradigm [33]. Although these criteria are comparable to the existing data-flow based criteria, they are not the same. They aim at defining intermediate coverage objectives and estimating the required test effort

[2] A branch in a control flow graph represents one of the two possible paths that can be traversed according to the outcome of the decision in the node. For example, an `if-then-else` branch statement leads to one of the possible paths, `then`-path or `else`-path, according to the true or false value of the `if`-condition.

towards the final one. These criteria are based on the operator network of a LUSTRE program, the path length and the notion of the *activation condition* of a path, which informally represents the propagation of the effect of the input edge through the output edge. In fact, an activation condition is associated with each path and is expressed by a boolean expression that shows the dependencies of the path output on the path input. These criteria have been implemented in LUSTRUCTU, a prototype tool which automatically measures the structural coverage of LUSTRE programs.

However, the existing LUSTRE coverage criteria face two significant deficiencies. On the one hand, they cannot be performed on the complete operator set of LUSTRE language; they can be applied only to specifications that are defined under a unique global clock. On the other hand, they can be applied only to single nodes that do not contain additional internal nodes. Real-world embedded systems are more complex with several components and usually operate on multiple clocks. Therefore, the solution to these to problems consists in suitably adapting the criteria definition to the needs of modern critical embedded software. This constitutes the second goal of this thesis.

In SCADE, coverage is measured through the Model Test Coverage (MTC) module, in which the user can define his own criteria by defining the conditions to be activated during testing. Thus, LUSTRE coverage criteria should be integrated in SCADE in order to achieve a powerful and efficient technique of coverage measurement for LUSTRE/SCADE applications.

In sum, the purpose of this thesis is to enhance the theoretical and empirical results of the existing research work on the testing of LUSTRE/SCADE programs and adjust it to the real and current industrial aspects, with a view to providing tools and methods directly applicable to avionics applications. The proposed approach attempts to provide the means to fully automate the testing process of critical systems, combining both the activities of test data generation and quality evaluation.

1.3 Contributions

The contribution of this thesis is twofold.

1. The first part concerns the structural coverage assessment of LUSTRE programs. The existing criteria defined for the LUSTRE language are extended to fully support the SCADE language. This extension is divided in two

main aspects.

First, it consists in taking into account the use of multiple clocks in a LUSTRE program. A LUSTRE program execution is determined by the global clock, a boolean flow that always values *true* and defines the frequency of the program execution cycles. Other, slower, clocks can be defined through boolean-valued flows. They are mainly used to prevent useless operations of the program and to save computational resources by forcing some program expressions to be evaluated strictly on specific execution cycles. Thus, nested clocks may be used to restrict the operation of certain flows when this is necessary, without affecting at the same time the rest of the program variables. This fact introduces modifications to the criteria definition and more precisely, to the path activation condition definition for the two temporal operators `when` and `current`.

These extended criteria are also implemented in LUSTRUCTU and integrated in SCADE MTC.

Second, integration testing as opposed to unit testing is further studied and new criteria are defined allowing the coverage measurement of large-sized systems. This extension requires the enhancement of the activation condition definition and, consequently, that of the criteria definition. To this end, we introduce an abstract model for the composed nodes to approximately estimate their coverage. We also define an abstracted form for the activation conditions, taking into account additional parameters apart from the path length. These parameters are the path length considered within the called node and the node integration depth with regard to the different levels of the tree-like structure that the nodes of a program form. In other words, the analysis considers the length of the paths to be covered inside the called node as well as whether the called node also contains calls to other nodes. In this way, we achieve a sound abstracted coverage model for the called nodes that is well related to the coverage of the other nodes.

The use and the applicability of the proposed criteria are illustrated in a case study from the avionics, with a view to demonstrating the required testing effort in order to meet the criteria as well as assessing their fault detection capability.

2. The second part addresses the automated test data generation using LUTESS aiming at the facilitation of the testing process. Recently, a testing

methodology has been proposed that allows the automatic construction of input test generators based on formal specifications. These specifications are defined in a LUSTRE-like language that contains a set of special operators making it possible to guide the test data generation process through several testing techniques. It mainly consists in modeling the system external environment by formally expressing the properties of the latter so that the generated test data fit best a given test objective and be the most suitable with regard to the system. In this context, we study and assess the applicability and the effectiveness of this approach on a case study from the aerospace domain. We demonstrate how the methodology can be employed and evaluate the results with respect to performance and complexity issues.

1.4 Thesis outline

The document comprises three main parts.

The first part, that consists of Chapters 2 and 3, provides the state of the art in software testing. Chapter 2 introduces the primary notions and techniques concerning software testing in general, while Chapter 3 focuses on the synchronous approach and the LUSTRE language. In addition, we describe in details the current practices and methods in software testing in the domain of avionics.

In the second part, that is composed of Chapters 4, 5 and 6, we present our work on the structural coverage of LUSTRE programs. In Chapter 4, we formally define the activation conditions for the LUSTRE temporal operators `when` and `current` and in Chapter 5 we introduce the extended structural coverage criteria considering node integration. Chapter 6 presents an experiment that evaluates the effectiveness and the applicability of the proposed coverage metrics on a case study of the alarm management system of an aircraft.

Chapter 7 of the third part illustrates the LUTESS test generation method that is based on modeling the system external environment. We demonstrate the applicability and the scalability of this approach using a case study of a military aircraft.

Finally, Chapter 8 summarizes the work in this dissertation and points to directions for future work.

Introduction

Dans tous les projets de développement de logiciel, de la phase de la conception jusqu'à la phase finale de la production, le logiciel est testé et inspecté afin de s'assurer qu'il se conforme aux exigences fonctionnelles et que son implémentation correspond correctement aux spécifications. Particulièrement dans le cas des applications critiques où un échec possible pourrait coûter des vies humaines ou des dégâts graves à l'équipement ou à l'environnement, la sûreté de fonctionnement [34, 52] est l'un des facteurs les plus importants qui permet d'avoir confiance dans le service délivré par le logiciel. Cette propriété peut être assurée au moyen de vérification et validation [66], deux activités significatives pendant le développement de logiciel qui visent principalement à éliminer des fautes. L'élimination des fautes est souvent obtenue, entre d'autres méthodes, par le test de logiciel, qui consiste surtout à exécuter le programme avec des jeux de tests données et à vérifier que les résultats obtenus s'accordent à ceux attendus.

Cette thèse porte sur le test des programmes réactifs synchrones écrits en Lustre avec un intérêt particulier aux systèmes du domaine de l'avionique.

Contexte de recherche

Les parties critiques des applications dans des domaines de l'aéronautique, des transports et de l'énergie concernent souvent des systèmes du type contrôle/commande. Ce type de logiciels doit être bien synchronisé avec leur environnement externe afin d'éviter tout défaut de fonctionnement avec des conséquences indésirables. En effet, la réaction du logiciel aux informations fournies par son environnement devrait être assez rapide (théoriquement instantanée) de sorte que n'importe quel changement de ce dernier puisse être pris en compte. Cette propriété, dite de *synchronisme*, est une caractéristique fondamentale des

logiciels réactifs synchrones qui assez fréquemment font partie des applications critiques. Afin de modéliser et spécifier correctement ce type des systèmes, plusieurs langages synchrones ont été proposés comme Esterel [9], Signal [36] et Lustre [22, 7]. Dans le cadre de cette thèse, nous nous sommes intéressés au langage Lustre.

Lustre est un langage déclaratif à flots de données avec des opérateurs temporels qui permettent de se référer aux valeurs précédentes des expressions. Toutes les variables et les expressions sont représentées par des flots de données, à savoir des séquences infinies des valeurs qui sont évaluées par rapport à une horloge discrète globale. Un programme Lustre décrit comment les flots de données d'entrée sont transformés en flots des données de sortie à chaque top de l'horloge globale. En plus de ce comportement cyclique, la nature déterministe du langage et sa sémantique formelle le rendent idéal pour programmer des systèmes synchrones réactifs. La structure d'un programme est décrite de façon graphique par un réseau d'opérateurs. Un réseau d'opérateurs est un graphe orienté: les noeuds symbolisent les opérateurs du programme et les arcs qui relient les opérateurs symbolisent les données transférées entre eux.

Lustre est le noyau de SCADE (Safety Critical Application Development Environment) [2], un outil graphique dédié au développement des systèmes critiques embarqués qui est souvent utilisé par les industriels et les professionnels. Cet outil permet la définition hiérarchique des systèmes et de ses composants, leur simulation graphique, leur vérification ainsi que la génération automatique de code. A partir des spécifications formelles en SCADE, un programme C est automatiquement généré, bien que cette transformation ne soit pas standardisée. Cet environnement graphique est essentiellement utilisé pour la spécification et la modélisation des systèmes du domaine de l'avionique.

La conception des systèmes embarqués critiques est une tâche difficile. Pendant les dernières années, les méthodes formelles et les approches basées sur des modèles ont été largement utilisés pour la conception, l'analyse et l'exécution des applications industrielles majeures. Tout au long du cycle de développement des systèmes, la vérification et la validation sont des activités essentielles et coûteuses. En particulier, le test logiciel est un moyen dynamique de vérification et de validation qui peut révéler des erreurs dans un système ou bien assurer que le modèle est conforme à l'implémentation améliorant ainsi la sûreté de fonctionnement du produit final.

Le but de cette thèse est d'étudier l'amélioration des procédures de test afin d' automatiser la génération des données de test ainsi que l'évaluation de

leur qualité dans le cadre des systèmes réactifs synchrones spécifiés en LUSTRE/SCADE. Ce travail s'est en partie déroulé dans le cadre du projet ANR SIESTA[3] (**A**utomatisation du **T**est de **S**ystèmes **E**mbarqués réalisés en SCADE et **Si**mulink), qui regroupe des partenaires universitaires et industriels autour d'une problématique commune: l'optimisation du coût de développement des applications SCADE, en prenant compte de la réalité industrielle.

Problèmes et motivations

En général, la procédure du test logiciel, dans un premier temps, consiste à sélectionner les données de test, ensuite exécuter le programme sous test avec les données de test sélectionnées et finalement identifier les différences entre les résultats effectifs et ceux attendus ainsi que s'assurer que le système est suffisamment testé. Dans ce contexte, il y a deux problématiques importantes. La première d'entre elles réside sur la phase de la sélection des données de test. Effectivement, une des parties les plus importantes au test logiciel c'est la conception et la création des jeux de tests pertinents. Tester exhaustivement un programme est l'objectif idéal car, dans ce cas, le programme pourrait s'observer dans tous ses états possibles et cela permettrait de détecter, théoriquement, toutes les erreurs. Cependant, ceci n'est pas vraiment réalisable à cause du nombre infini des valeurs d'entrée pour lesquelles le programme doit être testé. Donc, vu que le test complet d'un programme est impossible, la difficulté se trouve à choisir un ensemble des jeux de tests pertinent qui est le plus complet possible et qui pourrait révéler la majorité des erreurs. Étant données les contraintes de temps et du coût, il s'agit d'une tâche lourde. La deuxième problématique concerne l'évaluation des jeux de tests sélectionnés. L'objectif principal du test étant la découverte de fautes [41], quand un jeu de test ne met en évidence aucune erreur, cela souvent suggère que le programme n'est pas suffisamment testé. C'est pourquoi évaluer la qualité des tests est une étape toute aussi importante puisque elle apporte une mesure de la couverture du programme qui pourrait aider à décider de l'arrêt du test.

La recherche sur ces deux problématiques a été bien approfondie pendant des années et de nombreuses méthodes et stratégies ont été proposées. Les solutions proposées se sont souvent basées sur certaines règles, propriétés ou hypothèses que les jeux de tests doivent satisfaire. De telles propriétés s'appellent également

[3] www.siesta-project.com

des *critères*. Les *critères de sélection* sont utilisés pendant la phase de la sélection des données de tests tandis que les *critères d'arrêt* sont utilisés pendant la phase d'évaluation afin de déterminer si le programme est suffisamment testé. De tels critères sont la mesure de la couverture du code du programme (couverture du code) ou de ces exigences fonctionnelles (couverture des exigences) ou le pourcentage des erreurs détectées (couverture des erreurs) [10, 15].

En ce qui concerne le test des programmes écrits en LUSTRE, plusieurs outils de test [16, 37, 57] ont été développés afin d'automatiser la génération de données de test, réduisant ainsi l'importance du facteur humain. Ces outils sont basés sur les spécifications fonctionnelles ou la structure du programme sous test. GATeL [37] est un outil de test structurel basé sur la programmation par contraintes; les jeux de test sont générés en résolvant un système de contraintes imposées sur les variables du programme. En revanche, LUTESS [17, 49] est un outil de test fonctionnel qui génère des séquences de test dynamiquement selon un ensemble des contraintes imposées sur l'environnement du programme sous test. Cet outil permet à l'utilisateur diverses stratégies de test, comme le test aléatoire [45, 51, 57], le test guidé par des scénarios [46, 69] ou le test guidé par des propriétés de sûreté [63]. Récemment, l'outil a été entièrement revu [61] en utilisant la programmation logique par contraintes afin de traiter convenablement des variables numériques ainsi que pour permettre de prendre en compte des hypothèses sur le programme sous test. Malgré les progrès réalisés par les techniques et les outils de génération de tests, ce processus n'est pas totalement automatique et, souvent, les ingénieurs doivent générer des jeux de tests manuellement à partir des besoins fonctionnels du système, ce qui est difficile et coûteux. Ce travail consiste, pour une part, en la définition d'une méthodologie de test complète et précise basée sur LUTESS prenant en compte les besoins des applications industrielles et s'attache à évaluer l'applicabilité et l'efficacité d'une telle approche sur une étude de cas extraite du domaine de l'avionique.

En ce qui concerne l'évaluation de la qualité des données de test, plusieurs critères de couverture structurelle ont été proposés pour les langages impératifs. Ils sont plutôt basés sur le code source du programme et ils sont définis sur le graphe de flot de contrôle associé qui représente le transfert du contrôle tout au long de l'exécution du programme. Quelques exemples typiques sont la couverture des chemins, la couverture des instructions et la couverture des branches [41]. Le premier consiste à exécuter chaque chemin dans le graphe de flot de contrôle du programme au moins une fois. Ce critère est impossible à satisfaire

dans presque tous les cas à cause du nombre des chemins qui pourrait être infini dans le cas où le programme contient des boucles. Par contre, la couverture des instructions ou celle des branches exigent que chaque instruction ou branche, respectivement, soit exécutée au moins une fois pendant le test. Ces critères sont plus réalistes et donc plus utilisés [4]. Il y a également plusieurs critères structurels qui reposent sur la décomposition des expressions booléennes en décisions et conditions[4][41, 64]. Le critère de couverture de condition/décision modifiée (MC/DC) [13] est souvent utilisé dans le test des logiciels critiques embarqués. Enfin, une famille de critères structurels basés sur le flot de données [55] prend en compte les définitions et les utilisations des variables dans le programme.

Néanmoins, aucun des critères présentés ci-dessus ne peut être directement appliqué sur le code Lustre. Particulièrement, dans le cas des applications industrielles majeures du domaine de l'avionique, le test des composants critiques du logiciel est réalisé selon la norme DO-178B [1], qui exige que les séquences de test soient générées par rapport aux exigences fonctionnelles et obtiennent une couverture complète du programme par rapport au critère MC/DC. Cela implique que les mesures de couverture s'appliquent sur le code C généré automatiquement à partir des spécifications formelles en Lustre/SCADE. Pour une application développée et spécifiée en Lustre/SCADE, la mesure de la couverture sur le code C ne correspond pas exactement à la couverture réelle des exigences fonctionnelles car il n'y a pas de relation formelle entre les deux. La méthode de compilation Lustre-en-C n'étant pas encore standardisée, il est difficile pour les ingénieurs de faire le lien entre la couverture du code C et celle des spécifications formelles associées écrites en Lustre.

Peu des travaux [39, 49] ont abordé la couverture des programmes écrits en Lustre. Récemment, des critères de couverture structurelle spécialement conçus pour les programmes écrits en Lustre/SCADE ont été définis directement sur le paradigme synchrone [33]. Ces critères sont comparables aux critères existants basés sur le flot de données et ont pour but de définir des objectifs intermédiaires de couverture et d'estimer l'effort de test jusqu'à l'objectif final. Ces critères sont définis sur le réseau d'opérateurs du programme Lustre, en fonction de la longueur des chemins. Une condition d'activation est associée à chaque chemin et elle correspond à la dépendance entre la valeur de la sortie et celle de l'entrée du chemin. Ces critères ont été implantés dans Lustructu, un

[4]Une branche dans un graphe de flot de contrôle correspond à un choix dans un noeud. Par exemple, selon la valeur assignée à la condition d'une instruction conditionnelle `if-then-else` le contrôle sera transféré à la branche `then` ou bien à la branche `else`. .

outil à l'état de prototype qui effectue la mesure automatique de la couverture structurelle des programmes LUSTRE.

Cependant, les critères structurels de modèles LUSTRE sont principalement destinés au test unitaire en boîte blanche, sur les noeuds de tailles raisonnables. Donc, dans un cadre général, il y a deux questions qui restent ouvertes. D'une part, les critères ne supportent pas tous les opérateurs de LUSTRE (opérateurs d'horloge); ils ne peuvent traiter que des spécifications définies sous une horloge globale. D'autre part, ils sont capables de traiter seulement des noeuds qui ne contiennent pas d'appels vers d'autres noeuds. Par contre, la plupart des systèmes embarqués du monde réel se composent des nombreux composants et s'opèrent sous plusieurs horloges. En conséquence, la définition des critères doit être adaptée de manière pertinente aux besoins industriels des logiciels embarqués critiques. Ceci est le deuxième objectif de cette thèse.

En conclusion, l'objectif principal de cette thèse est l'extension des techniques et des pratiques actuelles utilisées pour le test des logiciels spécifiés en LUSTRE/SCADE et les adapter aux aspects courants du monde industriel, afin de fournir des outils et des méthodes adaptés aux applications du domaine de l'avionique. L'approche proposée vise à introduire les moyens d'automatiser complètement le processus du test des logiciels critiques en prenant en compte la génération de données de test ainsi que 'évaluation de leur qualité.

Contributions de la thèse

A travers ce travail, notre contribution est double.

1. La première partie concerne l'évaluation de la couverture structurelle des programmes LUSTRE. Les critères définis dans le contexte du langage sont étendus afin de supporter l'ensemble du langage Lustre/SCADE. Cette extension est envisagée dans deux directions.
   D'une part, elle consiste à prendre en compte l'utilisation des plusieurs horloges dans un programme LUSTRE. L'exécution d' un programme LUSTRE est déterminée par une horloge globale (*horloge de base*); en pratique, cette horloge est représentée par un flot booléen qui vaut toujours *vrai* et définit la fréquence des cycles d'exécution du programme. En définissant plusieurs flots booléens on peut avoir plusieurs horloges avec des fréquences différentes. La raison principale de l'utilisation des horloges multiples est d'éviter des opérations du programme inutiles ainsi que d'économiser des

ressources en imposant à certaines expressions du programme de n'être évaluées qu'à certains cycles d'exécution précis. Les horloges multiples peuvent être imbriquées. La prise en compte de plusieurs horloges introduit des modifications à la définition des critères de couverture et en particulier à la définition de la condition d'activation des chemins composés par les deux opérateurs temporels `when` et `current`.
Les critères étendus sont également implantés dans LUSTRUCTU ainsi que dans le module MTC de SCADE.
D'autre part, le test d'intégration est aussi étudié par opposition au test unitaire et des nouveaux critères sont définis qui permettent la mesure de la couverture des systèmes de grande échelle. Cette extension nécessite de fournir de nouvelles règles pour la définition de la condition d'activation et par conséquent pour la définition des critères. Pour cela, nous proposons la construction d'une approximation de la couverture des noeuds composés, en introduisant un modèle abstrait de ces derniers. Nous définissons également une forme abstraite de la condition d'activation, en tenant compte des paramètres complémentaires à la longueur de chemins. Ces paramètres sont la longueur de chemins considérée dans les noeuds appelés et le niveau d'intégration des noeuds. En d'autres termes, l'analyse de couverture prend en compte le fait que le noeud appelé fait appels à d'autres noeuds ainsi que la longueur de chemins à couvrir dans le noeud appelé. De cette manière, on obtient une abstraction du modèle de couverture.

L'utilisation et l'applicabilité des critères proposés sont démontrées sur une étude de cas extraite du domaine de l'avionique visant à illustrer l'effort de test nécessaire pour que les critères soient satisfaits et à évaluer leur aptitude à détecter des fautes dans le programme.

2. La deuxième partie aborde l'automatisation de la génération de données de test avec l'outil LUTESS afin de faciliter le processus du test. Une méthodologie de test est proposée, permettant à construire automatiquement des générateurs des valeurs d'entrée à partir des spécifications formelles du système sous test. Ces spécifications sont définies en un langage proche de LUSTRE qui contient un ensemble d'opérateurs spécialement destinés à guider la procédure de génération de test. Cette approche consiste à modéliser l'environnement externe du système en exprimant de manière formelle ses propriétés afin que les données de test générées soient le plus pertinentes possible par rapport aux besoins fonctionnels

et éventuellement à un objectif de test donné. Dans ce contexte, nous menons une étude d'évaluation de l'applicabilité et l'efficacité de cette approche de test sur une étude de cas extraite du domaine de l'avionique. Nous montrons comment la méthodologie peut être utilisée et nous faisons une évaluation des résultats obtenus par rapport à sa complexité.

Plan du document

Ce manuscrit s'articule autour de trois parties principales.

La première partie, constituée des chapitres 2 et 3, est dédiée à la présentation de l'état de l'art dans le domaine du test des logiciels. Le chapitre 2 introduit les notions fondamentales concernant le test de logiciels en général, tandis que le chapitre 3 se concentre sur l'approche synchrone et le langage Lustre. En plus, nous détaillons les pratiques industrielles et les méthodes utilisées actuellement pour le test des logiciels aéronautiques.

Dans la deuxième partie, qui se compose des chapitres 4, 5 et 6, nous présentons nos travaux sur la couverture structurelle des programmes écrits en Lustre. Dans le chapitre 4, nous exposons la définition formelle des conditions d'activation pour les opérateurs temporels `when` et `current`. Le chapitre 5 introduit la définition des critères étendus vers le test d'intégration. Le chapitre 6 décrit les expérimentations menées afin d'évaluer l'efficacité et l'applicabilité des mesures de couverture proposées sur une étude de cas d'un contrôleur d'alarme dans un système de contrôle de vol.

Le chapitre 7 de la troisième partie illustre la méthode de génération de tests sous Lutess qui est basée sur la modélisation de l'environnement externe du système. Nous présentons l'applicabilité et la mise en échelle de cette approche à travers une expérimentation de l'outil sur une étude de cas d'une appareil militaire.

Enfin, le chapitre 8 conclut ce document par un bilan ainsi que des perspectives et questions ouvertes de ce travail.

Part I

Software testing

Chapter 2

Testing: basic concepts and techniques

According to the definition given in [41], testing is a verification and validation technique which consists in executing a program with the intent of finding errors. Hence, it involves an activity of system evaluation by checking if the system satisfies the requirements described in its documentation and finding differences between expected and actual results. The difficulty of this procedure depends on various factors, such as the system complexity, the test objective, the tester's experience or even the clarity of the given specification. Nevertheless, regardless of all these limitations, testing is a determinative and necessary activity throughout the software development cycle and consumes more than the half of the total development cost and effort.

In general, regardless of the development cycle, testing is performed in two basic steps. Firstly, a test data set is selected and the program under test is executed over this set and then the obtained program outputs are observed and compared to the expected ones. The observation of the results can be either a manual or an automated procedure.

In the following, the testing process and several testing techniques are thoroughly discussed.

Selon la définition donnée dans [41], le test logiciel est une technique de vérification et de validation qui consiste à exécuter un programme avec l'intention de trouver des erreurs. Cela implique une évaluation dynamique du système

afin de vérifier s'il satisfait les exigences décrites dans sa spécification et afin de mettre en évidence toutes les différences entre les résultats effectifs et ceux attendus. La difficulté d'une telle procédure dépend de plusieurs paramètres comme la complexité du système, l'objectif de test, le niveau d'expérience du testeur ou bien la clarté des spécifications. Cependant, malgré toutes ces contraintes, le test logiciel est une activité très importante et nécessaire tout au long du cycle de développement du système et il absorbe plus de la moitié du coût total de développement.

En général, indépendamment du cycle de développement, le test logiciel est réalisé en deux étapes. Dans un premier temps, un ensemble de données de test est sélectionné avec lequel le programme sous test est exécuté. Ensuite, les résultats obtenus sont observés et comparés à ceux attendus. L'observation des résultats peut être un processus manuel ou automatique.

Dans ce qui suit, nous présentons en détails le processus général du test logiciel ainsi que certaines techniques de test.

2.1 Testing process

An efficient procedure of detecting errors in a program should combine several testing techniques and be adjusted to the kind of errors to be found as well as to the tester's vision on testing. Especially in critical industrial applications, testing must be carefully and thoroughly deployed in order to prevent disastrous situations; testing process can be generally considered as a four-step procedure, as it is shown in Figure 2.1.

The first step consists in analyzing the informal requirements that the system under test should satisfy. This information is usually part of the system documentation or specification sheet. Based on this information, test objectives are determined and a test data set is selected from all the possible input values. The program under test is executed over this test data set and the obtained results are compared against the expected ones. This step of observing the system execution allows to identify the errors and usually premises an oracle, either a human observer or an automated program, that distinguishes a false result from a correct one. Finally, once the program is executed over a test case, the quality of the latter should be evaluated and assessed according to the number and the *importance* of faults found. In this step, it is also decided whether the program has been sufficiently tested (so testing can stop) or further

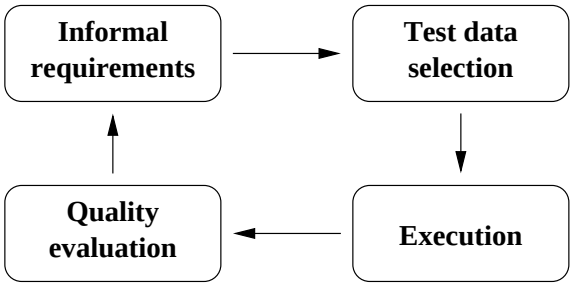


Figure 2.1: General testing procedure.

investigation is needed.

2.2 Testing techniques

Testing a program to find all of its errors is impractical and often impossible, even for trivial programs. It is extremely hard to simulate all the valid and possible conditions in which a program could be exposed to in order to execute it and observe the results. Instead, the objective of any testing technique should be to maximize the number of errors found in the minimum period of time and at a minimal cost. In other words, since complete testing is impossible, the key issue is to select a subset of all possible test cases that has the highest probability of detecting the most errors.

With regard to the stage of the development process where testing is deployed, testing techniques are identified among the three following ones. *Unit* or *module testing* focuses on the individual components of a system and serves to detect the errors introduced during the programming phase. *Integration testing* refers to the communication and interaction of these components and detects errors regarding the components interface or the program architecture. Lastly, *system testing* is conducted on the complete system and evaluates its compliance with its specified requirements.

Most common testing techniques are classified with respect to the way test data is selected. Test data selection based on the specification concerns *black-box testing* whereas code-based data selection results in *white-box testing*. Each method has distinct strengths and weaknesses and, in general, exhaustive black-box and white-box testing alone is impossible. However, in many cases, viewing the program as a black box and then taking a look inside its structure would

help maximize the yield on testing performance by maximizing the number of errors found in a finite number of test cases. That is, a combined use of both techniques that takes advantage of several of their elements can lead to a rigorous test of a program. These two techniques and their strategies are presented in details in the remainder of this chapter.

2.2.1 Black-box testing

In *black-box* or *functional* testing the program under test is considered to be a black box, that is to say only the program inputs and outputs and its function are known. The program code and structure are either unknown or ignored. Test cases are derived exclusively from program specification without regard to the program internal behavior. Therefore, this method is more capable of detecting errors because of missing or imprecise specification.

Ideally, exhaustively testing a program using every possible input would reveal all errors. However, this could not guarantee that the program is error-free, since one has to test it using not only all valid inputs, but all possible inputs, which results in an infinite number of required test cases. Thus, since exhaustive input testing is not feasible, several techniques are used to obtain a test data set as complete as possible.

Some of the most prevalent black-box testing methodologies include random testing, equivalence partitioning and boundary-value analysis. *Random testing* is often characterized as the simplest and the least effective testing methodology. It consists in randomly selecting some subset of all possible input values to test a program [26]. According to [18], random input testing is efficient as far as code coverage and fault detection aspects are concerned. *Equivalence partitioning* is based on the idea of finding the most appropriate test data set with the highest error detection probability. This method implies the division of a program input domain into a finite number of equivalence classes. If a test case in an equivalence class detects an error, all other test cases in the same class would be expected to find the same error and inversely. Hence, each test case reduces the number of other test cases that must be developed to reach a predefined test objective. However, it is difficult to equitably select the representative value of each class. This is the main reason why, in many cases, this method does not yield better results than random testing [25]. *Boundary-value analysis* [41] is a variation of equivalence partitioning and not only it explores the input values inside an equivalence class but those on, above and beneath the edges of each

class (input equivalence classes). In addition, test cases are derived directly from the specifications by considering the results as well (output equivalence classes). Boundary-value analysis, if practiced correctly, can be very effective and in several case detect errors that would go undetected if a random or ad hoc test case generation method were used.

2.2.2 White-box testing

Contrary to black-box testing, *white-box* or *structural* testing has total access to the program code and test data selection is driven only by the latter, often ignoring the program specification. White-box testing techniques are mainly based on the structural coverage of the control flow graph of the program under test; they examine to what extend test cases exercise or cover the program control flow.

In a control-flow graph each node represents a set of program statements that execute sequentially (i.e. a piece of code without any branch statement). Edges or arcs are used to represent branches in the control flow. In this case, exhaustively testing a program (similarly to black-box testing) would be equivalent to execute, via test cases, all possible paths of control flow (*path coverage*). Yet, this is not always possible nor useful. Actually, the number of paths in a program could be very large, especially in real programs that contain several loops, rendering complete path testing impractical. More importantly, even if every path is examined, there might still be errors in the program since the compliance with the specification is neglected.

There are several white-box testing methods that are principally concerned with a program structural coverage and closely connected with the yielded quality of the selected test data. The following section addresses the more widely known of these methods.

2.3 Quality evaluation

Although test data is compulsory so that a program is tested, test data generation is not sufficient for the testing process completion. Test data evaluation is also needed, in fact it is complementary to test data generation for the overall success of the testing process, in order to be able to evaluate its results and eventually determine its termination, as it is discussed in Section 2.1. To reassure the test cases quality, coverage analysis techniques are used, which can be

```
START
read(a);
read(b);
while b<>0 do
   r:=a mod b;
   a:=b;
   b:=r;
endwhile
print(a);
END
```

Figure 2.2: Program calculating the greatest common divisor (`gcd(a,b)`).

either functional, aiming at property coverage qualifying test cases according to their fault detection ability, or structural, focusing on code coverage.

Traditionally, coverage analysis requires a representative model of the program under test as well as a set of coverage criteria to be applied to this model. The coverage model consists in the set of elements (operators, instructions, branches, etc.) that are involved in the program execution. Usually, a program can be represented either by a control-flow graph or by a data-flow diagram.

A control flow graph, as it is stated in 2.2.2, denotes the order in which program instructions are executed and the paths that might be traversed through a program during its execution. Therefore, this model is better adapted to imperative programs. Each node is associated with a boolean expression corresponding to a condition that determines the transfer of control. On the contrary, in a data-flow diagram each node represents an atomic action while the edges represent the data transferred between these actions. This model is a proper graphical representation for programs written in declarative languages since it demonstrates well the flow of information through a program and explicitly describes data dependencies.

The control-flow graph of a simple program (given in Figure 2.2) that calculates the greatest common divisor of two integers a and b, is illustrated in Figure 2.3. Edges are labeled with instructions blocks and nodes correspond to decision points of the program. Instead, in the respective data-flow diagram, nodes would represent data processing in function of the input data values while edges would have been labeled with the necessary condition for each transition to be performed.

Coverage criteria are quantitative metrics that measure how well a test case has exercised a program. They are frequently referred to as *adequacy criteria* because they are used during the quality evaluation phase (see Figure 2.1) to

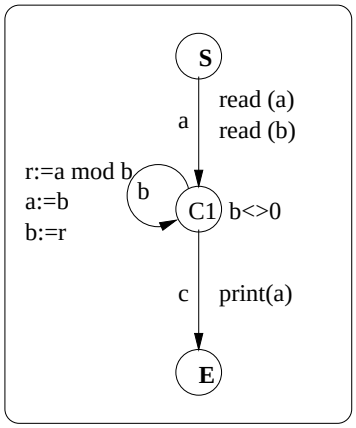


Figure 2.3: Control-flow graph.

evaluate test data quality *after* the latter has been selected, as opposed to *selection criteria* that are used to guide the test data selection phase (i.e. *before* test data is selected). The selection of a criterion depends on various parameters, as program complexity, time, cost and failure constraints. Consequently, there is a trade-off implied between the strength of a criterion and the number of required test cases for its satisfaction. The stronger the selected criterion, the more thoroughly the program is tested and the greater the number of faults potentially found. Conversely, a weaker criterion might detect fewer faults but with a lower number of test cases. The rest of this chapter is devoted to a brief state of the art in coverage criteria.

2.3.1 Control-flow-based coverage

Control-flow graphs can be used for static analysis [29, 40] as well as for defining test adequacy criteria [67].

Given that the execution of every path in a program is the stronger, albeit infeasible coverage criterion, weaker but more practical criteria are used in practice. For instance, statement coverage requires the execution of every statement in the program at least once (*statement coverage*). However, this criterion is too weak since the detection of each possible error cannot be reassured. Hence, more useful criteria offering a realistic compromise between path and statement coverage are widely used in the industrial domain.

A control flow statement execution implies a choice between two or more paths to be followed. Such a statement can be either an individual statement

```
if (cond1 && (cond2 || cond3))
     statement1;
else
     statement2;
```

Figure 2.4: An example for decision coverage.

(i.e. that cannot be broken down into simpler ones) and it is referred to as a *condition* [64] (or a *clause* [44]), or a composite statement, often involving logical operators, in which case it is referred to as a *decision*. Several proposed coverage criteria are based on the evaluation and execution of such conditions and decisions [41, 64, 65, 13], addressing mostly the coverage of boolean expressions.

Decision coverage This criterion requires that each program statement is executed at least once and each decision takes on both possible outcomes (*true* and *false*) at least once. Considering only two-way decisions, this means that both paths of a branch statement (e.g. `if-then-else`, `while`) should be traversed. This criterion is stronger than statement coverage but it is still rather weak because some conditions may be precluded and some mistakes may not be detected. Consider, for example the case in Figure 2.4. It is possible that the criterion is met without never exploring `cond3`. If both `cond1` and `cond2` are *true* then the decision is *true*. Otherwise, if `cond1` is false then the decision is also *false*. Thus, even though both branches have been examined, not every possible error can be found.

Condition coverage This criterion is stronger than the previous one since it ensures that each program statement is executed at least once and each condition in a decision takes on both possible outcomes (*true* and *false*) at least once. Hence, it requires two test cases for each condition. Even though it may seem that the condition coverage criterion subsumes the decision coverage criterion, this is not the case. It is possible that each condition is tested independently of each other and is evaluated to both *true* and *false* values without satisfying both the branches of the decision. For instance, in case of the decision `if (a || b)`, the test cases set {`a`, `b`}={*tf*, *ft*} satisfies condition coverage but do not cause the decision to become *false*.

Decision-condition coverage This criterion is a combination of decision and condition coverage. That is, it requires that each program statement is executed at least once, each decision takes on both possible outcomes at least once and each condition in a decision takes on both possible outcomes at least once. The drawback of this criterion is that some conditions might mask or block other conditions in the decision outcome. The test cases set {`a`, `b`}={*tt*, *ff*} would cover the decision `if (a || b)` according to this criterion but it does not distinguish the correct expression between `a`, `b`, `(a || b)` or `(a && b)`.

Multiple-condition coverage This criterion requires that each program statement is executed at least once and that all possible combinations of the conditions in each decision are tested at least once. Obviously, this is theoretically the strongest criterion of all and examines all possible condition outcomes in relation to the rest of the evaluated conditions. Nevertheless, it often involves a great number of required test cases, provided that for a decision composed of n conditions, the criterion is satisfied with 2^n tests. In case of a complicated program with many decisions and many compound conditions, this number could be prohibitive.

Modified condition-decision coverage (MC/DC) This criterion introduces an intermediate solution between the multiple-condition coverage criterion and the other criteria by minimizing the number of condition combinations [13, 28]. This modified criterion requires that each program statement is executed at least once, each condition in a decision takes on both possible outcomes at least once and each condition is shown to independently affect the decision outcome. Each test case exercises the evaluation of a decision with regard to the value variation of one condition at a time while all the other possible conditions remain fixed. That means that the dependencies between conditions and decisions are analyzed and only those conditions whose change is reflected on the decision are taken into account. That is, for the decision `if (a || b)`, test cases set {`a`, `b`}={*tf*, *ft*, *ff*} provide MC/DC coverage. Although a smaller number of condition combinations is needed in this case, still the number of required test cases increases linearly since for a decision with n conditions, $n+1$ test cases must be checked.

This coverage criterion is used in the DO-178B standard [1], developed in

RTCA as a means of certifying software in avionics, for the most critical software. In order to ensure the absence of failures of the standard highest level (Level A - Catastrophic), MC/DC must be fully satisfied (100% coverage).

LCSAJ This coverage metric [68] is defined directly on the program source code and stands as a variation of path coverage. A *LCSAJ* (Linear Code Sequence And Jump) is a segment of statements that execute sequentially and is terminated by a jump in the control flow. Longer sub-paths, hence stronger coverage metrics, are gradually constructed by concatenating several LCSAJs. The obtained coverage is a rather useful measurement that is more thorough than decision coverage and yet approaches path coverage. The metric unit used is the Test Effectiveness Ratio (TER_i) [68] corresponding to the number of covered LCSAJs. In this way, a criteria hierarchy is formed; statement coverage (TER_0) is the weaker criterion, branch (decision) coverage follows (TER_1), then there is the coverage of one LCSAJ (TER_2), two LCSAJs (TER_3) and so on. In general, TER_{n+1} measures the coverage of n LCSAJs.

2.3.2 Data-flow-based coverage

Several path selection criteria have been proposed based on data-flow analysis [55, 20, 14]. These criteria examine the association between the definition of a variable and its uses. They demonstrate how values are related to variables and how these relations can affect the program execution, focusing on the occurrences of variables within the program. Each variable occurrence is either a *definition* (the variable is associated with a value) or a *use* (reference to the variable). A use can be identified to be either a computation-use (*c-use*) or a predicate-use (*p-use*). The first type of use affects directly the computation of a variable or indicates the result of some earlier definition, that is a variable or an output calculation. The second type of use directly affects the control flow through the program and corresponds to the evaluation of a predicate which determines the executed path (a predicate from a conditional branch statement).

The analysis is concerned not only with the definitions and uses of variables but also with the sub-paths from definitions to statements that use those definitions. A definition of a variable x in a node n of the program data-flow graph reaches a use of x associated with a node m if and only if there is a path $p = n \cdot p_1 \cdot m$ such that the sub-path p_1 does not contain any definition of x and

there is no definition of x in m (p_1 is a *definition-clear sub-path* with regard to x).

The following path selection criteria [58, 14] represent predicates defined with regard to a set of paths P in program A. If the predicate is evaluated to a true value for every path in P, then P satisfies the criterion.

All-Defs (Definition coverage) This criterion requires that a path set contains at least one definition-clear sub-path from each definition to some use reached by that definition. That is, this criterion is satisfied if test data cover all the definitions of a variable.

All-Uses (Use coverage) This criterion requires that a path set contains at least one definition-clear sub-path from each definition to some use reached by that definition and each successor of the use. That is, this criterion is satisfied if test data cover all uses reached by a definition.

All-DU-Paths (Definition-Use coverage) This is the strongest criterion of this family; it is an extended variation of All-Uses and subsumes it. It requires that every definition-clear sub-path from each definition to some use reached by that definition is a simple cycle or is cycle-free, in order to assure a finite number of paths.

Different variations of the above criteria as well as criteria that make the distinction between computation and predicate uses can be found in [55]. Other criteria based on dependencies between variable definitions and uses are defined in [42, 35].

Chapter 3

Testing Lustre programs

This chapter is dedicated to the state of the art in software testing with regard to the synchronous approach. After introducing the essential characteristics of synchronous reactive software and the basic notions concerning the LUSTRE language, we present some methods and tools designed for test data generation and coverage assessment.

Ce chapitre est consacré à un état de l'art du test logiciel par rapport à l'approche synchrone. Après avoir introduit les caractéristiques fondamentalles des logiciels réactifs synchrones ainsi que les notions principales sur le langage LUSTRE, nous présentons des méthodes et des outils conçus pour la génération des données de test et l'évaluation de la couverture structurelle.

3.1 Reactive software

A *reactive* system [27] maintains a permanent interaction with its physical environment and continuously responds to external stimuli. A reactive program is operated on a continuous action-reaction basis and it instantly reacts to its inputs at a speed determined by its environment. Contrary to classical *transformational* systems, that compute or perform a function transforming their inputs into the outputs, and *interactive* systems, in which the environment handles the timing and synchronization issues (e.g. operating systems or data base systems), reactive systems are often deterministic since the outputs depend only on the current and past input values. Their behavior is described by the sequences of

their actions as well as the relations between the program inputs and outputs and their combination in time.

The main application domains of reactive systems include embedded systems, industrial process control systems, signal processing systems and communication protocols. *Real-time* systems are also a subclass of reactive systems with additional constraints on the response time imposed by the external environment. In general, such systems are highly critical systems and they are characterized by the parallel interaction between the system and its environment as well as the time constraints imposed by the latter regarding input frequencies and input/output response times.

3.2 The synchronous approach

The complexity of reactive systems stems mainly from the fact that the program is executed in parallel with its environment evolution. Consequently, modeling this kind of systems is not an easy task. Dependability [34, 52] and safety are important issues that must be considered all along the design and development stages. In order to ensure the soundness of reactive systems, the synchronous approach [6] was introduced. This approach considers that the software reaction is sufficiently fast so that every change in the external environment is taken into account. This hypothesis of *synchrony* leads to an abstracted deterministic program, from both temporal and functional points of view and thus it makes it possible to efficiently design and model synchronous systems. In fact, it is sufficient that software response time is lower than the minimum delay needed for a change in the environment. As soon as the order of all the events occurring both inside and outside the program is specified, time constraints describing the behavior of a synchronous program can be expressed [23].

In a synchronous program, the outputs are computed in a time determined by the duration of a cycle according to a global clock. This cycle defines an infinite sequence of instants that corresponds to the basic clock. In this sense, every synchronous program is characterized by a cyclic behavior where at each cycle, the program receives its inputs, calculates the outputs and feeds the environment with the latter. This behavioral model, in which time is divided into discrete instants, allows to precisely relate every internal action of the program with regard to its environment. At each instant t_i, the input i_i is read and processed and the output o_i is emitted simultaneously, as it is shown in Figure 3.1.

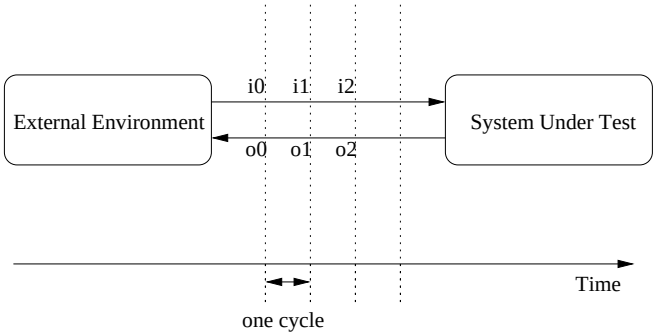


Figure 3.1: Synchronous software operation

Several programming languages have been proposed to specify and implement synchronous applications. Esterel [9] is an imperative language based on an explicit expression of the control flow while Signal [36] and LUSTRE [22, 7] are declarative languages based on data flows.

3.3 LUSTRE overview

LUSTRE [11] is a declarative data-flow language developed for the specification of reactive synchronous applications. The merging of both synchronous and data-flow paradigms, its simple graphical syntax as well as the notion of discrete time are some of the main characteristics that make the language ideal for the modeling and the design of control systems in several industrial domains, such as avionics, automotive and energy. It provides formal specification and verification facilities and ensures efficient C code generation.

Contrary to imperative languages which describe the control flow of a program, LUSTRE describes the way that the inputs are turned into the outputs. Any variable or expression is represented by an infinite sequence of values and is considered to be a function of time. Time is logical and discrete, that is there is no notion of instant duration or elapsed time between two instants. Indeed, time is represented by a global clock which is an infinite sequence of time instants (i.e. execution cycles) and determines the occurrence and the frequency of data flows. Hence, variables are associated with a global clock and take the n-th value at the n-th cycle of this clock and the hypothesis of synchrony is satisfied. This property provides a simplified way of handling time issues which are of great importance in real-time systems.

```
node Never(A: bool) returns (never_A: bool);
let
  never_A = not(A) -> not(A) and pre(never_A);
tel;
```

	c_1	c_2	c_3	c_4	...
`A`	*false*	*false*	*true*	*false*	...
`never_A`	*true*	*true*	*false*	*false*	...

Figure 3.2: Example of a LUSTRE node.

A LUSTRE program is structured into nodes. A node is a set of equations which define the node outputs as a function of its inputs. Each variable can be defined only once within a node and the order of equations is of no matter. Specifically, when an expression E is assigned to a variable X, X=E, it indicates that the respective sequences of values are identical throughout the program execution; at any cycle, X and E have the same value. This property is very useful in program transformation. Once a node is defined, it can be used inside other nodes like any other operator.

The operators supported by LUSTRE are the common arithmetic and logical operators (`+`, `-`, `*`, `/`, `and`, `or`, `not`) as well as two specific temporal operators: the *precedence* (`pre`) and the *initialization* (`->`). The `pre` operator introduces to the flow a delay of one time unit, while the `->` operator -also called *followed by* (`fby`)- allows the flow initialization. Let $X = (x_0, x_1, x_2, x_3, \ldots)$ and $(e_0, e_1, e_2, e_3, \ldots)$ be two LUSTRE expressions. Then `pre(X)` denotes the sequence $(nil, x_0, x_1, x_2, x_3, \ldots)$, where *nil* is an undefined value, while X`->`E denotes the sequence $(x_0, e_1, e_2, e_3, \ldots)$.

LUSTRE does not support loops (operators such as `for` and `while`) nor recursive calls. Consequently, the execution time of a LUSTRE program can be statically computed and the satisfaction of the hypothesis of synchrony can be checked.

A simple LUSTRE program is given in Figure 3.2, followed by an instance of its execution. This program has a single input boolean variable and a single boolean output. The output is *true* if and only if the input has never been *true* since the beginning of the program execution.

In addition, assertions can be used to specify certain assumptions regarding the system environment in case that there is some knowledge on the system specification. These assumptions are introduced through the operator `assert` followed by a boolean expression; an assertion denotes that the property enclosed

in `assert` will always be satisfied during the program execution. For instance, the following assertion states that the boolean input variables x and y can never occur simultaneously:

$$assert\, not\, (x\, and\, y)$$

Not only assertions provide the compiler with directives in order to optimize the code but they are used also in program verification.

3.3.1 Compilation

LUSTRE compiler [56] associates a sequential program with the source LUSTRE code. It generates executable code corresponding to a finite state automaton that represents an abstract execution model of the parallel LUSTRE description. The main tasks of the compiler amount to:

- definition checking: there must be one single equation that defines any local or output variable
- clock consistency
- absence of cyclic definitions: any cycle should contain at least one `pre` operator.

The translation of a LUSTRE node into an imperative program can be done by constructing an infinite loop that implements all the equations of the source code performed at any cycle of the basic clock. The compiler defines auxiliary variables and uses memory variables to translate LUSTRE temporal operators. The order of actions is specified according to the LUSTRE node structure and the dependencies between variables. Even though the speed of the produced code with this compilation method is not at all optimal, it is simpler and its size is linear to the number of LUSTRE expressions.

Alternatively, another compilation technique uses a more complex control structure and associates a finite state automaton to a LUSTRE node. Each state of this automaton represents the past actions before reaching this state. Transitions are labeled with input and output values and a state is characterized by a set of state variables. A state variable is created for each `pre` expression of the source code in order to keep its previous value. The drawback of this technique is the size of the obtained code that might be very large. However, this model is suitable for validation and verification purposes.

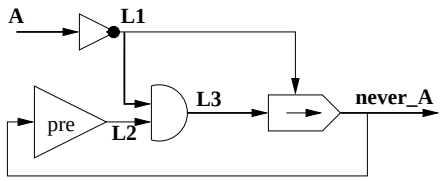


Figure 3.3: The operator network for the node `Never`.

3.3.2 Operator network

The transformation of the inputs into the outputs in a LUSTRE program is done via a set of operators. Therefore, a LUSTRE program can be represented by a directed graph, the so called *operator network*. An operator network is a graph with a set of N operators which are connected to each other by a set of $E \subseteq N \times N$ directed edges. Each operator represents a logical or a numerical computation. With regard to the corresponding LUSTRE program, an operator network has as many input edges (respectively, output edges) as the program input variables (respectively, output variables).

Figure 3.3 shows the corresponding operator network for the node of Figure 3.2. Such a graph designates the dependencies of the program outputs on the inputs.

An operator represents a data transfer from an input edge towards an output edge. There are two kinds of operators:

- the basic operators which correspond to a basic computation and
- the compound operators which correspond to the case that a node calls another node.

A basic operator is denoted as $\langle e_i, s \rangle$, where e_i, $i = 1, 2, 3, \ldots$, stands for its inputs edges and s stands for the output edge.

3.3.3 Clocks in LUSTRE

As noted earlier in this section, each variable and expression in LUSTRE denotes a flow, i.e. each infinite sequence of values is defined on a clock, which represents a sequence of time. Thus, a flow is a pair of a sequence of values and a clock.

The clock serves to indicate when a value is assigned to the flow. That means that a flow takes the n-th value of its sequence of values at the n-th time of its clock. Any program has a cyclic behavior and that cycle defines a sequence of

times, i.e. a clock, which is the *basic clock* of a program. A flow on the basic clock takes its n-th value at the n-th execution cycle of the program. Slower clocks can be defined through flows of boolean values. The clock defined by a boolean flow is the sequence of times at which the flow takes the value *true*.

Two operators affect the clock of a flow: `when` and `current`.

- `when` is used to *sample* an expression on a slower clock.
  Let E be an expression and B a boolean expression with the same clock. Then, `X=E when B` is an expression whose clock is defined by B and its values are the same as those of E's only when B is *true*. That means that the resulting flow X has not the same clock as E or, alternatively, when B is *false*, X is not defined at all.

- `current` operates on expressions with different clocks and is used to *project* an expression on the immediately faster clock.
  Let E be an expression with the clock defined by the boolean flow B which is not the basic clock. Then, `Y=current(E)` has the same clock as B and its value is the value of E at the last time that B was *true*. Note that until B is *true* for the first time, the value of Y will be *nil*.

The sampling and the projection are two complementary operations: a projection changes the clock of a flow to the clock that the flow had before its last sampling operation. Trying to project a flow that was not sampled produces an error. The input parameters of a node may be defined on distinct clocks (different from the basic one). In this case, the faster one is the basic clock of the node and all the other clocks must be in the declaration input list. Outputs also can be defined on different clocks, certainly slower that the basic one, and careful implementation is needed so that these are visible from outside of the node. Table 3.1 explains the function of the two temporal LUSTRE operators in more details.

`E`	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	...
`B`	*false*	*false*	*true*	*false*	*true*	*false*	*false*	*true*	*true*	...
`X=E when B`			$x_0 = e_2$		$x_1 = e_4$			$x_2 = e_7$	$x_3 = e_8$	...
`Y=current(X)`	$y_0 = nil$	$y_1 = nil$	$y_2 = e_2$	$y_3 = e_2$	$y_4 = e_4$	$y_5 = e_4$	$y_6 = e_4$	$y_7 = e_7$	$y_8 = e_8$	...

Table 3.1: The use of the operators `when` and `current`.

The LUSTRE node `ex2cks` [21] in Figure 3.4 shows the usage of clocks. The node receives as input the signal m. Starting from this input value when the

```
node ex2cks(m:int) returns (c:bool; y:int);
var (x:int) when c;
let
  y = if c then current(x) else pre(y)-1;
  c = true -> (pre(y)=0);
  x = m when c;
tel;
```

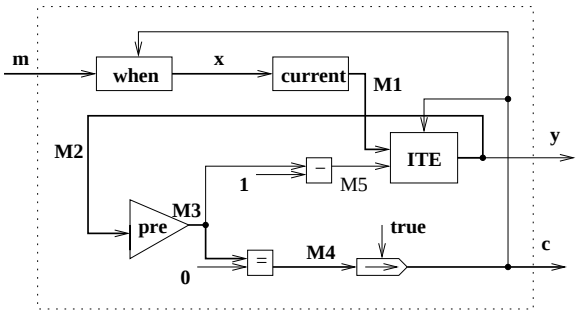


Figure 3.4: An example using two clocks (the basic clock and the flow c) and the corresponding operator network.

clock c is true, the program counts backwards until zero; from this moment, it restarts from the current input value and so on.

3.4 Synchronous software development

This section is largely inspired from the SIESTA project report produced in the context of this research [19].

For several years the design of real time embedded software for the control parts of safety-critical systems depends mainly on Simulink or SCADE[1] specifications.

Especially in the development of critical avionics systems, SCADE and Simulink environments are the two most commonly used. Following the general development procedure (Figure 3.5), SCADE models are built by manually translating the Simulink models that describe the system physical environment (mainly in UML [59]); the latter are derived from the system functional requirements. Then, SCADE automatically generates the embedded code through the

[1] www.esterel-technologies.com

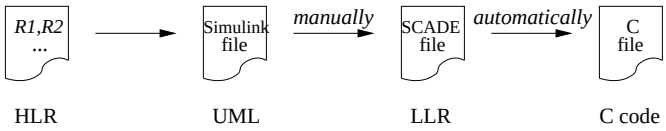


Figure 3.5: Development process in avionics.

qualified code generator (KCG). Test scenarios are built manually from the system functional requirements. Figure 3.6 illustrates the testing procedure used currently in the domain of avionics in relation to the respective development procedure in Figure 3.5; this is a more detailed version of the general testing process shown previously in Section 2.1 (Figure 2.1). Functional test coverage is verified based on the coverage of the SCADE model inputs with respect to the functional requirements. For some systems, the analysis of the system inputs coverage is combined with the analysis of the structural code coverage, in order to determine when testing can stop as well as to identify missing tests. Although these practices are well controlled and the combined use of SCADE models with a qualified code generator has eliminated the unit tests, it is still essential to further reduce the cost of test campaigns. Therefore, the conformity of models to the system functional requirements must be ensured as early as possible in the development cycle.

Within this context, several studies have been carried out and various tools have been developed with the aim of optimizing the test coverage of systems developed in SCADE. They are mostly based on the automated test generation guided by coverage criteria specially adapted to data flow languages. Nonetheless, these tools and practices need to be combined towards a systematic and generally approved testing approach that adequately bridges the existing gaps. One of the primary targets of such a testing approach is to provide the means of automatic generation of test vectors and stop criteria, by analyzing the structural coverage of system specifications. More precisely, industry and academia are interested in responding to two prevalent research questions:

1. according to the functional system requirements, how could we generate test campaigns able to ensure that requirements are satisfied?

2. at the specification level, which would be a suitable structural coverage criterion with regard to its ability to indicate that the system is adequately tested and if not, to track the uncovered parts and reason about them?

As far as programs written in Lustre and the relevant testing tools are con-

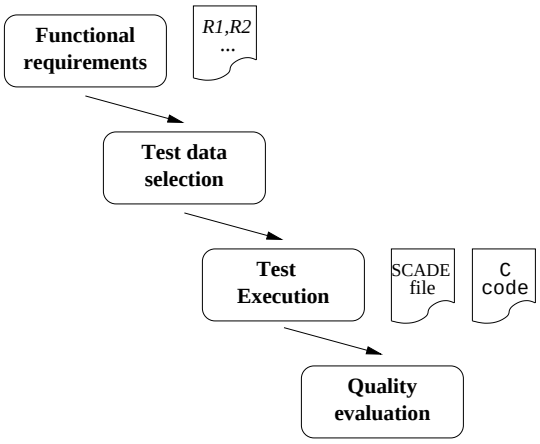


Figure 3.6: Current testing process in avionics.

cerned, the open issues lie in two directions. As for the test suites design, the problem remains open in terms of the definition of the test objectives as well as the way that test input vectors are generated. In practice, test engineers manually design and build test cases with regard to a given test objective; the test data generation process is far from being fully automated. Furthermore, assessing the completeness of a test campaign is traditionally achieved by measuring the percentage of code exercised during testing, while the precise definition of the structural coverage conditions depends on the level of software criticality. The traditional coverage criteria are defined on the control flow graph of the generated code. As a result, this concept of coverage is rather unsuitable for Simulink or SCADE specifications that handle data flows in operator networks. The main challenge is to identify within this network the set of conditions that activate different paths of program execution. Thus, regarding testing of critical LUSTRE/SCADE applications, a concrete testing methodology that would effectively combine automatic test data generation based on the system specifications with well formed coverage metrics to assess test data quality is a necessity.

A selection of testing tools, based on the LUSTRE language, that are currently used for testing real-scale industrial applications is presented in the following sections.

3.5 Test data generation

As discussed in Section 2.2, test data generation methods are mainly divided into two classes according to the information available about the system. This section focuses specifically on LUSTRE programs and again, the methods of the first class are based on black-box testing and simply require a program binary description to generate test data, whereas the second class includes the methods based on white-box testing which requires the LUSTRE code of the program under test. In general, black box testing techniques aim at detecting faults in the program under test and data generation is performed by random testing under certain constraints on the system environment or functional requirements. On the one hand, testing a system with constraints on its environment consists in reducing the set of states to be explored by taking into account only meaningful system behaviors. For example, incompatible input values correspond to a non realistic behavior and would be excluded in this case. On the other hand, functional requirements are LUSTRE boolean expressions that must always hold (also referred to as safety properties), thus testing a system against its functional requirements amounts to leading the system towards a state so that a safety property could be violated. On the opposite, test cases generated with white-box testing techniques ensure that the internal operations are performed according to the specification and all internal components have been adequately exercised.

Several test data generation tools have been proposed for programs specified in LUSTRE. LUTESS [17, 61] is a black-box testing environment which automatically transforms formal specifications into test data generators. Its main application domain is the validation of the control part of a software. GATeL [37, 38] is based on constraint logic programming but, in contrast to LUTESS, it is rather a white-box testing tool. It translates a LUSTRE program and its environment specification in an equivalent Prolog representation and then computes a test input according to precise test objectives. Contrary to LUTESS that generates test data with dynamic interaction with the system under test, GATeL interprets the LUSTRE code. Lurette [57] is another black-box testing tool similar to LUTESS and it makes possible to test LUSTRE programs with numeric inputs and outputs. A boolean abstraction of the environment constraints is first built: any constraint consisting of a relation between numeric expressions is assimilated to a single boolean variable in this abstraction. The concrete numeric expressions are handled by an ad hoc environment dedicated to linear arithmetic expressions. The generation process first assigns a value to the

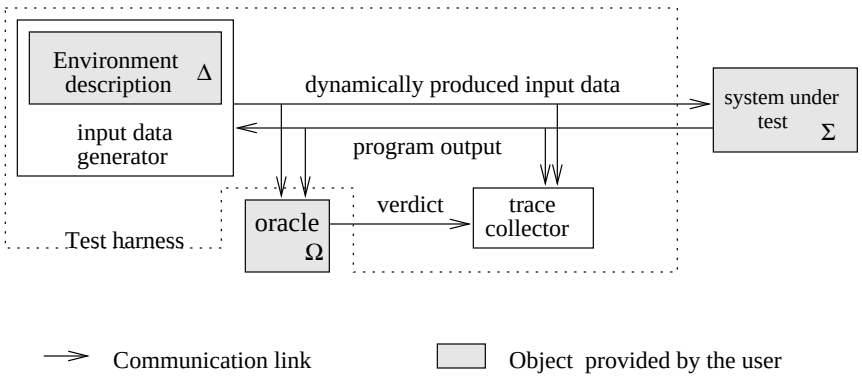


Figure 3.7: The Lutess testing environment.

variables of the boolean abstraction and, then, tries to solve the corresponding equations to determine the values of the numeric variables. However, LUTESS uses constraint logic programming instead of an ad hoc resolution environment restricted to linear expressions, as Lurette does. Moreover, the LUTESS specification language is an extension of the LUSTRE language while Lurette uses ad hoc scenario description notations. A thorough discussion on these tools as well as a wider selection of model-based test data generation tools can be found in [5].

In the following, we analyze LUTESS and GATeL, as two tools using the two different LUSTRE-based testing techniques, black-box and white-box respectively.

3.5.1 LUTESS

To perform the test operation, LUTESS requires three components: the software environment specification (Δ), the executable code of the system under test (Σ) and a test oracle (Ω) describing the system requirements, as shown in Figure 3.7. The system under test and the oracle are both synchronous executable programs.

LUTESS builds a test input generator from the test specification as well as a test harness which links the generator, the system under test and the oracle. LUTESS coordinates their execution and records the input and output sequences as well as the associated oracle verdicts thanks to the trace collector.

The test is operated on a single action-reaction cycle. The generator produces an input vector and sends it to the system under test; the later reacts

```
testnode Env(<SUT outputs>) returns (<SUT inputs>);
var <local variables>;
let
   environment(Ec_1); ... environment(Ec_n);
   prob(C_1, E_1, P_1); ... prob(C_m, E_m, P_m);
   safeprop(Sp_1, Sp_2, ...., Sp_k);
   hypothesis(H_1, H_2, ...., H_l);
   <definition of local variables>;
tel;
```

Figure 3.8: General form of a testnode syntax.

with an output vector sent back to the generator. The generator produces a new input vector and the cycle is repeated. The oracle observes the exchanged inputs and outputs to detect failures. The testing process is stopped when the user-defined test sequence length is reached.

The LUTESS generator selection algorithm performs a fair selection and chooses a valid input vector according to the environment description. In fact, any valid input vector has the same probability to be selected similarly to random testing. In addition to this generation mode (also referred to as random environment simulation), more testing strategies are supported by LUTESS: operational profiles [46], behavioral patterns [16] and safety-property guided testing [50].

The new version of LUTESS [62] uses Constraint Logic Programming (CLP) for test generation and it is able to handle numeric input and outputs. The environment description is made in an extended version of LUSTRE as a new file, the main node of which is called *testnode* and the general form of its syntax can be seen in Figure 3.8. The inputs (outputs) of a *testnode* are the outputs (inputs) of the program under test. The *testnode* is automatically transformed into a test data generator.

As a rule, a *testnode* contains four operators specifically introduced for testing purposes.

- The `environment` operator is used to specify a list of invariant properties, stated as LUSTRE expressions, that should hold at each execution step. The definition of the input domains for variables can be stated as an instant invariant as well as a temporal property. At any cycle, the test generator chooses a sequence of valid random input values to supply the software with. Therefore, the environment constraints can only depend

on the previous values of the output signals[2].

- The `prob` operator is used to define conditional probabilities, useful to guide test data selection. The expression `prob(C,E,P)` means that if the condition `C` holds then the probability of the expression `E` to be true is equal to `P`.
  There is no support for statically checking the defined probabilities. For example, a variable could be defined to hold *true* with a probability $p_1 \in (0,1)$ and *false* with a different probability $p_2 \in (0,1)$. Therefore, it may happen that the defined operational profiles cannot be satisfied, resulting in a constraint system impossible to satisfy. The specified operational profiles can be considered inconsistent and the environment specification cannot satisfy any concrete input values. In this case, there are two possible situations to be considered. If the test data must strictly agree with the probability specification, then the testing operation terminates and the tester is requested to modify the assigned probabilities. Alternatively, the generator could try to satisfy as much constraints as possible in order to generate input values, so it ignores the specification causing the inconsistency and continue the generation process. Both options are implemented in LUTESS [61].

- The `safeprop` operator is used to guide the test generation process towards situations where the safety properties could be violated. The resulting test data generator would generate data that have a high probability to reveal failures related to the safety properties.

- The `hypothesis` operator is used as a complement to `safeprop` in order to insert assumptions on the program under test which could improve the fault detection ability of the generated data. Such assumptions could be either the result of the system analysis or system properties already tested that are considered to be satisfied.

3.5.2 GATeL

Contrary to LUTESS that starts the test sequence generation from the initial state and then sequences are dynamically generated, GATeL [37] starts with

[2]Supposing that at the current instant t, the input signal *i(t)* must be issued before the software computes the output *o(t)*, then, if the environment definitions were referring to the current output *o(t)*, the generation of valid input sequences at the instant t would be impossible, since the actual value *o(t)* would be yet unknown.

the final state to be reached by the test sequence. Thus test data generation is performed before the system under test is executed (and not during its execution).

GATeL requires the LUSTRE code of the program under test, its environment description as well as a test objective. A test objective can be a safety property or a path predicate. Invariant properties that must hold in each step of the generated test sequence are stated with the `assert` directive. Properties that must be satisfied at least once are stated using the `reach` directive. Based on the system under test and the environment description, GATeL tries to find a test sequence which satisfies both properties expressed in the `assert` and `reach` statements. If such a test case is found, it is executed over the system under test and the generated output is compared with the corresponding one expected from the precomputed test case. If these two match then the test case passes, otherwise it fails.

The test sequence search is carried out backwards through a resolution procedure for constraints built from an interpretation of LUSTRE constructions over boolean variables, variables with integer intervals and a synchronization constraint on the status of each cycle. Resolution proceeds by successive eliminations of all constraints building a compatible past and backtracking some output variables.

The basic testing with GATeL supplies a single test case. In order to generate more test cases, the tool offers the possibility of domain splitting, that is decomposing the input variable domain in sub-domains. This is done by unfolding LUSTRE operators. For instance, in case of the boolean variable *condition* and the following constraint :

`if` *condition* `then` *expr1* `else` *expr2*

GATeL unfolds the `if-then-else` operator in two sub-domains, the one containing the test cases with *condition=true* and the second with *condition=false*. This results in the definition of two separate constraint systems according to the value of the *condition* variable. These sub-domains can be further split with regard to the rest of operators; at each step, the tool indicates which operators can be unfolded and the user interacts choosing which domains should be split. In this way, it is possible to ensure some structural coverage with respect to a test objective.

3.6 Structural coverage of LUSTRE programs

Since a LUSTRE program can be compiled in an imperative C program, structural coverage is usually computed on the latter, based on the corresponding control-flow graph. In this model, there can be applied every structural testing technique described in Section 2.3.1. Particularly for critical industrial applications, SCADE uses the MTC module (Model Test Coverage) in order to measure structural coverage of the SCADE model according to the DO-178B standard [1]. However, as there is no universal and standard method for the LUSTRE-to-C compilation, coverage measured on the C code does not exactly stands for the actual coverage of the original LUSTRE program. What is more, LUSTRE programs are properly represented by an operator network and common coverage techniques designed for imperative languages are not consistent with the data-flow nature of the language. Hence, coverage metrics should be applied to the operator network, so that LUSTRE programs are effectively tested and implementation faults can be detected.

In [45, 51], a preliminary study on the coverage of the operator network of LUSTRE programs has been presented which resulted in the definition of a family of structural coverage criteria especially designed for LUSTRE programs [31]. Although these criteria are comparable to the existing data-flow based criteria [55, 14], presented in Section 2.3.2, they are not the same. They aim at defining intermediate coverage objectives and estimating the required test effort towards the final one. These criteria are based on the notion of the *activation condition* of a path, which informally represents the propagation of the effect of the input edge through the output edge.

3.6.1 Paths and activation conditions

Paths in an operator network represent the possible directions of flows from the input through the output. Formally, a path is a finite sequence of edges $\langle e_0, e_1, \ldots, e_n \rangle$, such that for $\forall i \epsilon [0, n-1]$, e_{i+1} is a successor of e_i in N. A *unit path* is a path with two successive edges. For instance, in the operator network of Figure 3.3, there can be found the following paths of maximum length 7.

$p_1 = \langle A, L_1, never_A \rangle$
$p_2 = \langle A, L_1, L_3, never_A \rangle$
$p_3 = \langle A, L_1, never_A, L_2, L_3, never_A \rangle$
$p_4 = \langle A, L_1, L_3, never_A, L_2, L_3, never_A \rangle$

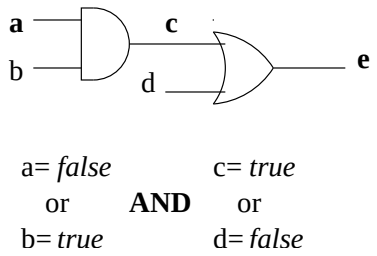


Figure 3.9: Activation condition for the path $\langle a, e\rangle$.

Obviously, one could discover infinitely many paths in an operator network depending on the number of cycles repeated in the path (i.e. the number of `pre` operators in the path). However, we only consider paths of finite length by limiting the number of cycles. That is, a path of length n is obtained by concatenating a path of length *n-1* with a unit path (of length 2). Thus, beginning from unit paths, longer paths can be built; a path is finite if it contains no cycles or if the number of cycles is limited.

A boolean LUSTRE expression is associated with each pair $\langle e, s\rangle$, denoting the condition on which the data flows from the input edge e through the output s. This condition is called *activation condition*. The evaluation of the activation condition depends on what kind of operators the paths is composed of. Informally, a path is activated if any change in its input causes a consequent change in its output. Hence, a path activation condition shows the dependencies between the path inputs and outputs. Consider, for example, the path $p = \langle a, e\rangle$ in the operator network shown in Figure 3.9. The path is decomposed in unit sub-paths, $p_1 = \langle a, c\rangle$ and $p_2 = \langle c, e\rangle$. The first sub-path is activated if a=*false*, because then the output c will be necessarily *false*, or if b=*true*, because in this case c depends exclusively on the value of a. Similarly, for the `OR` operator, the path input value is propagated through the output if c=*true*, because then e will be unconditionally *true*, or if d=*false*, because then e is only affected by c. Finally, the activation condition for path p is the conjunction of the partial conditions resulting in the boolean expression 3.1.

$$AC\,(p) = (not\,(a)\ or\,b)\ and\ (c\,or\,not\,(d)) \tag{3.1}$$

Therefore, the selection of a test set satisfying the paths activation conditions in an operator network leads to a notion for the program coverage. Since covering all the paths in an operator network could be impossible, because of

Operator	**Activation condition**
$s = NOT\,(e)$	$AC\,(e, s) = true$
$s = AND\,(a, b)$	$AC\,(a, s) = not\,(a)\ or\, b$ $AC\,(b, s) = not\,(b)\ or\, a$
$s = OR\,(a, b)$	$AC\,(a, s) = a\, or\, not\,(b)$ $AC\,(b, s) = b\, or\, not\,(a)$
$s = ITE\,(c, a, b)$	$AC\,(c, s) = true$ $AC\,(a, s) = c$ $AC\,(b, s) = not\,(c)$
relational operator	$AC\,(e, s) = true$
$s = FBY\,(a, b)$	$AC\,(a, s) = true$ -> $false$ $AC\,(b, s) = false$ -> $true$
$s = PRE\,(e)$	$AC\,(e, s) = false$ -> $pre\,(true)$

Table 3.2: Activation conditions for all LUSTRE operators.

their potentially infinite number and length, in this approach, coverage is defined with regard to a given path length.

Table 3.2 summarizes the formal expressions of the activation conditions for most of the LUSTRE operators, as these were defined in [33]. In this table[3], each operator `op`, with the input e and the output s, is paired with the respective activation condition $AC\,(e, s)$ for the unit path $\langle e, s\rangle$. Noted that some operators may define several paths through their output, so the activation conditions are listed according to the path inputs.

Let us consider the path $p_2 = \langle A, L_1, L_3, never_A\rangle$ in the corresponding operator network for the node `Never` (Figure 3.3). The condition under which that path is activated is represented by a boolean expression showing the propagation of the input A through the output $never_A$. To calculate its activation condition, we progressively apply the rules for the activation conditions of the corresponding operators according to Table 3.2. Starting from the end of the path, we reach the beginning, moving one step at a time along the unit paths. Therefore, the necessary steps would be the following:

$AC\,(p_2) = false$ -> $AC\,(p')$, where $p' = \langle A, L_1, L_3\rangle$

$AC\,(p') = not\,(L_1)\ or\, L_2\, and\, AC\,(p'') = A\, or\, pre\,(never_A)\ and\, AC\,(p'')$,

[3]In the general case (path of length n), the path p containing the `pre` operator is activated if its prefix p' is activated at the previous cycle of execution, that is $AC\,(p) = false$ -> $pre\,(AC\,(p'))$. Similarly in the case of the initialization operator `fby`, the given activation conditions are respectively generalized in the forms: $AC\,(p) = AC\,(p')$ -> $false$ (i.e. the path p is activated if its prefix p' is activated at the initial cycle of execution) and $AC\,(p) = false$ -> $AC\,(p')$ (i.e. the path p is activated if its prefix p' is always activated except for the initial cycle of execution).

where $p'' = \langle A, L_1 \rangle$
$AC\,(p'') = true$

After backward substitutions, the boolean expression for the activation condition of the selected path is:

$$AC\,(p_2) = false \text{ -> } A\,or\,pre\,(never_A).$$

In practice, in order for the path output to be dependent on the input, either the input has to be *true* at the current execution cycle or the output at the previous cycle has to be *true*; at the first cycle of the execution, the path is not activated.

3.6.2 Coverage criteria

The proposed coverage criteria [31] are specifically defined for LUSTRE programs and they are specified on the operator network according to the length of the paths and the input variable values.

Let $\mathcal{T}$ be the set of test sets (input vectors) and $P_n = \{p | length(p) \leq n\}$ the set of all paths in the operator network whose length is smaller or equal to n. Hence, the following families of criteria are defined for a given and finite order $n \geq 2$. The input of a path p is denoted as $in\,(p)$ whereas a path edge is denoted as e.

1. **Basic Coverage Criterion (BC).** This criterion is satisfied if there is a set of test input sequences, $\mathcal{T}$, that activates at least once the set P_n. Formally, $\forall p \in P_n$, $\exists t \in \mathcal{T}$: $AC\,(p) = true$. The aim of this criterion is basically to ensure that all dependencies between inputs and outputs have been exercised at least once. In case a path is not activated, certain errors such as a missing or misplaced operator could not be detected.

2. **Elementary Conditions Criterion (ECC)**. In order that an input sequence satisfies this criterion, it is required that the path p is activated for both input values, *true* and *false* (taking into account that only boolean variables are considered). Formally, $\forall p \in P_n$, $\exists t \in \mathcal{T}$: $in\,(p) \wedge AC\,(p) = true$ and $not\,(in\,(p)) \wedge AC\,(p) = true$. This criterion is stronger than the previous one in the sense that it also takes into account the impact that the input value variations have on the path output.

3. **Multiple Conditions Criterion (MCC).** In this criterion, the path output depends on all combinations of the path edges, also including the internal ones. A test input sequence is satisfied if and only if the path activation condition is satisfied for each edge value along the path. Formally, $\forall p \in P_n$, $\forall e \in p$, $\exists t \in \mathcal{T}$: $e \wedge AC\,(p) = true$ and $not\,(e) \wedge AC\,(p) = true$.

The above criteria form a hierarchical relation: MCC satisfies all the conditions that ECC does, which also subsumes BC. Note also that the MCC criterion is close to the constraints imposed by MC/DC; in fact, it is less powerful than MC/DC but more powerful than DC (Decision Coverage).

3.6.3 Lustructu

Lustructu [30] is an academic tool that integrates the above criteria and automatically measures the structural coverage of Lustre/SCADE programs. It requires three inputs: the Lustre program under test, the required path length and the maximum number of loops in a path and finally the criterion to satisfy. The tool analyzes the program and constructs its operator network. It then finds the paths that satisfy the input parameters and extracts the conditions that a test input sequence must satisfy in order to meet the given criterion. This information is recorded in a separate Lustre file, the so called *coverage node*. This node receives as inputs those of the program under test and computes the coverage ratio at the output. The program outputs become the node local variables. For each path of length lower or equal to the value indicated in the input, its activation condition and the accumulated coverage ratio are calculated. These coverage nodes are compiled and executed (like any other regular Lustre program) over a given test data set and the total coverage ratio is computed.

3.6.4 SCADE MTC

In SCADE, coverage is measured through the Model Test Coverage (MTC) module, in which the user can define his own criteria by defining the conditions to be activated during testing. Indeed, MTC measures the coverage of low-level requirements (LLR coverage), with regard to the demands and objectives of DO-178B standard, by assessing how thoroughly the SCADE model (i.e. system specification) has been exercised. In particular, each elementary SCADE operator is associated with a set of features concerning the possible behaviors of

the operator. Therefore, structural coverage of the SCADE model is determined by the activation ratio of the features of each operator.

In order to compute coverage for a given test cases set, model coverage is performed in four steps. During the *instrumentation* phase, the initial SCADE model is instrumented and the criteria that test cases must satisfy are defined. Default instrumented libraries applying the MC/DC or the DC coverage criteria are provided for SCADE predefined operators. In addition, user-instrumented libraries can be created to support several user-defined coverage criteria. Once the instrumented model is created, the *acquisition* phase follows; this phase produces and records which ones of the SCADE elements have been covered. The *analysis* phase provides a complete view of the coverage results by analyzing the covered and the uncovered parts and taking corrective actions. Finally, the *reporting* phase produces customizable reports containing information about the coverage criteria, results and records.

3.7 Requirements-based testing

In [53], an interesting validation approach is proposed that defines objective coverage metrics for measuring requirements coverage, that is providing a means of precisely knowing whether or not system requirements are sufficiently tested. The author uses LTL semantics (Linear Temporal Logic) to formally express high-level requirements. Three coverage criteria are then defined directly on the formalized requirements in order to determine how well a test suite has exercised the requirements set. In addition, these criteria can be used to automatically generate test cases sets for requirements testing [54].

These requirements-based criteria are not limited to the use of LTL notation for the requirement formalization; any formal notation may be used to translate informal requirements into formal properties. Therefore, they provide a new direction to the structural coverage of LUSTRE/SCADE programs. Since LUSTRE is a rather powerful formal language with well-defined temporal logic of the past, it could be used in place of LTL to formalize requirements; in this way, the structural coverage criteria defined in Section 3.6 could also be used to measure requirements coverage. This aspect may be helpful in the area of critical avionics software testing to fulfill the rigorous requirements of DO-178B standard that targets at full requirements coverage in addition to structural code coverage.

3.8 Objectives of this thesis

Motivated by the current practices and needs of testing in the avionics, that are thoroughly discussed in Section 3.4, the outer goal of this thesis is concerned not only with the improvement of the validation and the testing of critical real-time embedded systems but also with the simplification of its cost. Therefore, we attempt to provide contributions that answer to both research questions addressed in Section 3.4.

The existing structural coverage criteria cannot effectively deal with real-world complex systems that demand thorough and preferably automated testing. One of the basic difficulties in the testing process is the lack of a formal relation between the test objective and the system formal specification. Indeed, the tester must ensure that the generated test cases cover the system functional requirements and the test objective that the latter define. Coverage metrics applied to the C code, generated by SCADE without a standard compilation method, do not exactly correspond to the actual LUSTRE specification. Although the coverage criteria, specially adjusted to the synchronous paradigm (BC, ECC, MCC), provide an adaptable framework for evaluating the test data thoroughness of LUSTRE programs, they cannot be performed on the complete operator set of the language; they can be applied only to specifications that are defined under a unique global clock. However, nested clocks may be defined through the temporal operators `when` and `current` presented in Section 3.3.3 to restrict the operation of certain flows when this is necessary, without affecting at the same time the rest of the program variables. Real-time systems are rather sophisticated and usually operate on multiple clocks. For this reason, we propose the extension of the criteria definition to support also the multi-clock operators, `when` and `current`.

Aside from that, integration testing must be considered in addition to unit testing. That means that in order to compute the system coverage, the internal nodes that represent compound operators should not be unfolded in order to avoid a possible explosion in the number of calculated paths and activation conditions. If already measured, the coverage of such nodes should be reused or otherwise approximately estimated. With the purpose of solving this problem, we recommend the definition of a set of new criteria, including a notion of abstraction to make it possible to measure the program coverage also in the case of large-sized programs without expanding the internal nodes and, thus, reducing the cost of the coverage analysis.

In regard to test data generation, a new testing methodology based on LUTESS was recently proposed [60] for synchronous reactive applications. Taking advantage of the new features introduced to the tool [61, 62], a progressively increasing approach of test data generation according to the environment description arises, which could contribute to the full automation and the effectiveness of the testing process. In the following, we evaluate this approach with respect to the difficulty of its applicability, its effectiveness and usefulness as well as its scalability. To do so, we use a case study of a land gear controller of a military aircraft and we propose the guidelines towards testing. Eventually, the combination of this approach with the structural coverage criteria, according to the above extensions, could result in a concrete testing methodology adapted in modern industrial needs, that would not only automatically produce test cases but also provide the means to adequately test a system.

Part II

Extended structural test coverage criteria

Chapter 4

Multiple clocks

Despite the research conducted through both theoretical and empirical investigation in testing of LUSTRE/SCADE applications, there are still some insufficiently treated key issues. The proposed approaches and tools must be extended in order to meet the needs of real-world critical industrial systems.

In fact, a basic feature of current major applications is complexity; the whole system is composed of many distinct components that constantly interact with each other and some functions may use more than one clock. Such sophisticated systems cannot be effectively and adequately tested using the existing LUSTRE structural coverage criteria [33]. These criteria do not provide the means of assessing the coverage of LUSTRE nodes that make use of more than one clocks. This chapter deals with the extension of the criteria towards this aspect [47]. This extension not only allows to take into account the complete operator set of the language but also to efficiently apply the coverage metrics to real-world programs.

Malgré les recherches, tant théoriques que pratiques, qui ont été menées sur le test de logiciels critiques écrits en LUSTRE/SCADE, il reste des questions importantes insuffisamment traitées. Les approches et les outils proposés doivent être étendus afin de satisfaire les besoins des systèmes industriels critiques.

En effet, une caractéristique des plusieurs applications est la complexité; le système en sa totalité est composé de plusieurs composants distincts qui interagissent sans cesse entre eux. Certaines fonctions peuvent opérer sous plusieurs horloges. De tels systèmes ne peuvent pas être testés en utilisant les critères de couverture structurelle qui ont été spécialement définis pour les programmes

LUSTRE [33]. Ces critères ne fournissent pas les moyens d'évaluer la couverture des noeuds LUSTRE qui utilisent des horloges multiples. Ce chapitre traite l'extension des critères vers cet aspect [47]. Une telle extension permet non seulement de tenir compte de l'ensemble complet des opérateurs du langage mais d'appliquer également efficacement les métriques de la couverture sur des programmes de grande taille.

4.1 The use of multiple clocks

The cyclic behavior of a LUSTRE program execution implies the notion of a basic clock that represents the frequency that output flows are evaluated in function of the values of input flows. Since a flow is defined not only by its sequence of values but also by a boolean flow indicating *when* the flow is evaluated, a LUSTRE node may use more than one clocks in reference to the basic one. The appropriate LUSTRE operators that handle multiple clocks are `when` and `current`, as they are presented in Section 3.3.3.

Multiple clocks are mainly used in order to avoid the execution of some program functions when this is not necessary. In other words, the use of multiple clocks implies the filtering of some program expressions. It consists in changing their execution cycle, activating it only at certain cycles of the basic clock. Consequently, the associated paths are activated only if the respective clock is true. As a result, the tester must adjust this rarefied path activation rate according to the global *timing*.

The extension of LUSTRE criteria to the use of multiple clocks does not affect the criteria definition itself; instead, it requires the definition of the activation conditions for the two temporal multi-clock operators, `when` and `current`.

4.2 Activation conditions for `when` and `current`

Informally, the activation conditions associated with the `when` and `current` operators are based on their intrinsic definition. Since the output values are defined according to a condition (i.e. the *true* value of the clock), these operators can be represented by means of the conditional operator `if-then-else`. For the expression E and the boolean expression B with the same clock,

- `X=E when B` could be seen as `X=if B then E else NON_DEFINED` and similarly,

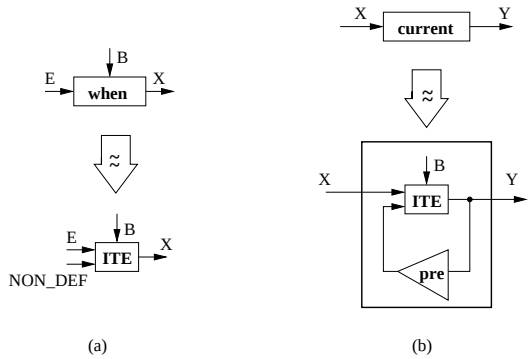


Figure 4.1: Modeling the `when` and `current` operators using the `if-the-else`.

- `Y=current(X)` could be seen as `Y=if B then X else pre(Y)`.

Hence, the formal definitions of the activation conditions result as follows:

Definition 4.1. *Let e and s be the input and output edges respectively of a* `when` *operator and let b be its clock. The activation conditions for the paths $p_1 = \langle e, s\rangle$ and $p_2 = \langle b, s\rangle$ are:*
$AC(p_1) = b$
$AC(p_2) = true$

Definition 4.2. *Let e and s be the input and output edges respectively of a* `current` *operator and let b be the clock on which it operates. The activation condition for the path $p = \langle e, s\rangle$ is:*
$AC(p) = b$

As a result, to compute the paths and the associated activation conditions of a LUSTRE node involving several clocks, one has just to replace `when` and `current` operators by the corresponding conditional operator (see Figure 4.1). At this point, two basic issues need to be further clarified. The first one concerns the case of `when`. In fact, there is no way of defining the value of the expression X when the clock B is not *true* (branch *NON_DEF* in Figure 4.1(a)). By default, at these instants, X does not occur and such paths (beginning with a non defined value) are infeasible[1]. In case of `current`, the operator implicitly

[1] An infeasible path is a path which is never executed by no test cases, hence it can never be covered.

refers to the clock parameter B, without using a separate input variable (see Figure 4.1(b)). This hints at the fact that `current` always operates on an already sampled expression, so the clock that determines its output activation should be the one on which the input is sampled.

Example 4.3. Let us assume the path $p = \langle m, x, M_1, M_2, M_3, M_4, c\rangle$ in the example of Section 3.3.3, displayed in bold in Figure 3.4. Following the same procedure for the activation condition computation and starting from the last path edge, the activation conditions for the intermediate unit paths are:

$AC\,(p) = false$ `->` $AC\,(p_1)$, where $p_1 = \langle m, x, M_1, M_2, M_3, M_4\rangle$
$AC\,(p_1) = true\ and\ AC\,(p_2)$, where $p_2 = \langle m, x, M_1, M_2, M_3\rangle$
$AC\,(p_2) = false$ `->` $pre\,(AC\,(p_3))$, where $p_3 = \langle m, x, M_1, M_2\rangle$
$AC\,(p_3) = c\ and\ AC\,(p_4)$, where $p_4 = \langle m, x, M_1\rangle$
$AC\,(p_4) = c\ and\ AC\,(p_5)$, where $p_5 = \langle m, x\rangle$
$AC\,(p_5) = c$

After backward substitutions, the activation condition of the selected path is:

$$AC\,(p) = false \ \texttt{->}\ pre\,(c)\,.$$

This condition corresponds to the expected result and is compliant with the above definitions, according to which the clock must be true to activate the paths with `when` and `current` operators.

In order to evaluate the impact of these temporal operators on the coverage assessment, we consider the operator network of Figure 3.4 and the paths:
$p_1 = \langle m, x, M_1, y\rangle$
$p_2 = \langle m, x, M_1, M_2, M_3, M_4, c\rangle$
$p_3 = \langle m, x, M_1, M_2, M_3, M_5, y\rangle$

Intuitively, if the clock c holds true, any change of the path input is propagated through the output, hence the above paths are activated. Formally, the associated activation conditions to be satisfied by a test set are:

$AC\,(p_1) = c$

$AC\,(p_2) = false$ `->` $pre\,(c)$

$AC\,(p_3) = not\,(c)\ and\, false$ `->` $pre\,(c)$.

Eventually, the input test sequences satisfy the basic criterion. Indeed, as soon as the input m causes the clock c to take the suitable values, the activation conditions are satisfied, since the latter depend only on the clock. In particular, in case the value of m at the first cycle is an integer different from zero (for sake of simplicity, let us consider $m = 2$), the basic criterion (BC) is satisfied in two steps since the corresponding values for c are *c=true*, *c=false*. On the contrary, if at the first execution cycle m equals to zero, the BC is satisfied after three steps with the corresponding values for c: *c=true*, *c=true*, *c=false*. These two samples of input test sequences and the corresponding outputs are shown in Table 4.1.

	c_1	c_2	c_3	c_4	...
m	i_1 ($\neq 0$)	i_2	i_3	i_4	...
c	*true*	*false*	*false*	*true*	...
y	i_1	$i_1 - 1$	0	i_4	...

	c_1	c_2	c_3	c_4
m	i_1 ($= 0$)	i_2	i_3	...
c	*true*	*true*	*false*	...
y	0	i_2	$i_2 - 1$	...

Table 4.1: Test cases samples for the input m.

Table 4.1 implies that the application of the coverage criteria depends on the way test data are generated. Sometimes, the first test data sequence determines the way that testing is carried out and, consequently, the coverage results. However, the complexity of the test generation process with regard to the criteria is hard to be estimated. In fact, in case of manual test data construction where test cases are based on system requirements, coverage metrics help the tester determine additional test cases covering paths not yet exercised. On the other hand, in case of automatic test data generation, coverage metrics are used to produce test cases satisfying a criterion.

4.3 Example: a counter

In order to demonstrate the usage and the application of the extended criteria, as these were presented above, let us consider a simple example. This example concerns a LUSTRE node [24] that receives at the input a boolean signal *set* and returns at the output a boolean signal *level*. The latter must be true during *delay* cycles after each reception of *set*. Now, suppose that we want the *level* to be high during *delay* seconds, instead of *delay* cycles. Taking advantage of

```
node TIME_STABLE(set, second: bool; delay: int) returns (level: bool);
var ck: bool;
let
   level = current(STABLE((set, delay) when ck));
   ck = true -> set or second;
tel;
node STABLE(set: bool; delay: int) returns (level: bool);
var count: int;
let
   level = (count>0);
   count = if set then delay
           else if false->pre(level) then pre(count)-1
                else 0;
tel;
```

Figure 4.2: The node `TIME_STABLE`: a simple example with the `when` and `current` operators.

the use of the `when` and `current` operators, we could call the above node on a suitable clock by filtering its inputs. The second must be provided as a boolean input *second*, which would be *true* whenever a second elapses. The node must be activated only when either a *set* signal or a *second* signal occurs and in addition at the initial cycle, for initialization purposes. The LUSTRE code is quite simple and it is shown in Figure 4.2, followed by the associated operator network[2].

Similarly to the previous example, the paths (complete and cycle-free) to be covered are:

$p_1 = \langle set, T_2, T_3, T_9, level \rangle$
$p_2 = \langle delay, T_8, T_3, T_9, level \rangle$
$p_3 = \langle set, T_1, ck, T_2, T_3, T_9, level \rangle$
$p_4 = \langle second, T_1, ck, T_2, T_3, T_9, level \rangle$
$p_5 = \langle set, T_1, ck, T_8, T_3, T_9, level \rangle$
$p_6 = \langle second, T_1, ck, T_8, T_3, T_9, level \rangle$

To cover all these paths, one has to select a test set satisfying the following activation conditions, calculated as it is described above:

$AC\,(p_1) = ck$, where $ck = true$ `->` $set\,or\,second$

[2]The nested node `STABLE` is used unfolded, since for the moment, the dependencies between a called node inputs and outputs cannot be determined.

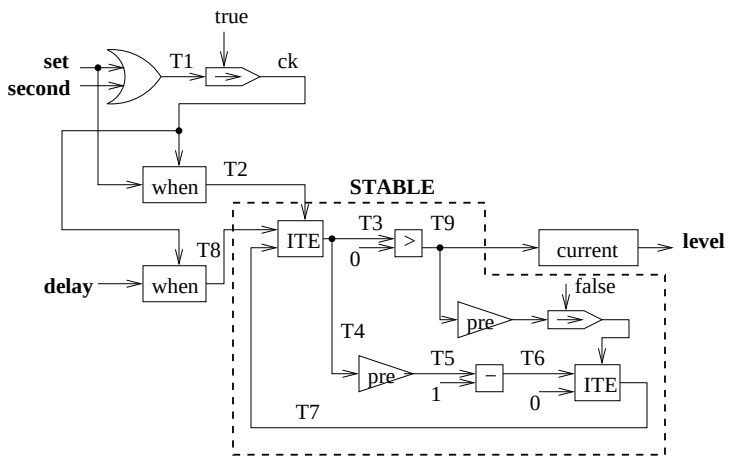


Figure 4.3: The operator network for the node `TIME_STABLE`.

$AC\,(p_2) = ck\,and\,set$
$AC\,(p_3) = ck\,and\,false$ -> $set\,or\,not\,(second)$
$AC\,(p_4) = ck\,and\,false$ -> $second\,or\,not\,(set)$
$AC\,(p_5) = ck\,and\,set\,and\,false$ -> $set\,or\,not\,(second)$
$AC\,(p_6) = ck\,and\,set\,and\,false$ -> $second\,or\,not\,(set)$

Since the code ensures the correct initialization of the clock, hence its activation at the first cycle, the above paths are always activated at the first execution cycle. For the rest of the execution, the basic criterion is satisfied with the following test sequence for the inputs $(set, second)$: $(1,0)$,$(0,1)$, $(1,1)$. This test set, which contains almost every possible combination of the inputs, satisfies also the elementary conditions criterion (ECC), since the activation of the paths depends on both boolean inputs.

Admittedly, the difficulty to meet the criteria is strongly related to the complexity of the system under test as well as to the test case generation effort. Moreover, activation conditions covered with short input sequences are easy to be satisfied, as opposed to long test sets that correspond to complex instance executions of the system under test. However, the enhanced definitions of the structural criteria complete the coverage assessment issue for LUSTRE programs, as all the operators of the language are supported. In addition, the complexity of the criteria is not further affected, because, in substance, we use nothing but `if-then-else` operators.

4.4 SCADE MTC

We have implemented the above criteria in the prototype tool LUSTRUCTU. In fact, we have integrated LUSTRUCTU within SCADE MTC in the form of a new plug-in for user-instrumented libraries. In particular, in the instrumentation phase the user is prompted to fill in the necessary parameters for the LUSTRUCTU execution. The activation conditions corresponding to the defined criteria (BC, ECC, MCC) are automatically transformed into suitable MTC expressions, the respective user-instrumented libraries are created and the coverage node automatically generated by LUSTRUCTU corresponds to the final instrumented model. This model is ready to be launched into the acquisition phase.

Chapter 5

Integration testing

In this chapter, we present an integration testing technique for the coverage measurement of large-scale LUSTRE programs that involve several internal nodes. Actually, we propose an approximation for the coverage of the called nodes by extending the definition of the activation conditions for these nodes. Coverage criteria are redefined according not only to the length of paths but also the level of integration. This extension reduces the total number of paths at the system level and hence, the overall complexity of the coverage computation.

Dans ce chapitre, nous présentons une technique de test d'intégration pour la mesure de la couverture de programmes LUSTRE de grande taille qui se composent des plusieurs noeuds appelés. En effet, nous proposons une abstraction de la couverture des noeuds appélés en étendant la définition des conditions d'activation pour ces noeuds. Nous définissons de nouveau les critères de couverture par rapport à la longueur de chemins et au niveau d'intégration. Cette approche de couverture réduit le nombre de chemins au niveau du noeud principal et ainsi, diminue la complexité de l'analyse de couverture.

5.1 Node integration

The existing coverage criteria are defined on a unit-testing basis and cannot be applied to LUSTRE nodes that locally use user-defined operators (compound operators). The cost of computing the program coverage is affordable as long as the system size remains small. However, big or complex nodes must be locally expanded and code coverage must be globally computed; as a result, the

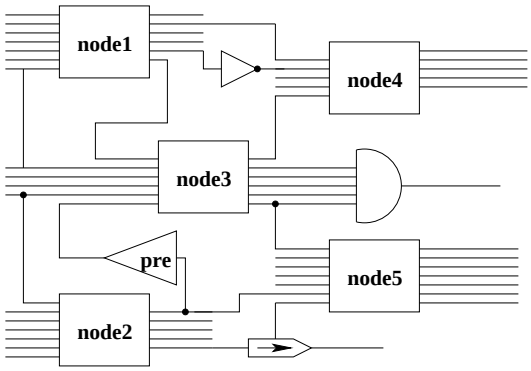


Figure 5.1: Example of a complex LUSTRE program.

number and the length of the paths to be covered are highly increased, hence these coverage metrics become impracticable.

Indeed, as far as relatively simple LUSTRE programs are concerned, the required time for coverage computation is rather short, especially in the cases of the basic and elementary condition coverage [32]. Paths are relatively short and the corresponding activation conditions are respectively simple. As long as the path length remains low, the number of the activation conditions to be satisfied is affordable. However, coverage analysis of complex LUSTRE nodes, like the one shown in Figure 5.1, may require a huge number of paths to be analyzed and the coverage cost might become prohibitive and, consequently, the criteria inapplicable.

This is particularly true for the MCC criterion, where the number of the activation conditions to be satisfied increases dramatically when the length and the number of paths are high. In fact, in order to measure the coverage of a node that contains several other nodes (i.e. internal calls to other nodes), the called nodes are unfolded, the paths and the corresponding activation conditions are locally computed and then they are combined with the global node coverage. This may result in a huge number of paths and activation conditions. In fact, covering a path of length k requires $2\,(k-1)$ activation conditions to be satisfied. Consequently, satisfying a criterion for the set of paths P_n, r_i being the number of paths of length equal to i, necessitates the satisfaction of $2\,(r_2 + 2r_3 + \cdots + (n-1)\,r_n)$ activation conditions.

5.2 Path length w.r.t. temporal loops

The length of a path $p = \langle e_1, e_2, \ldots, e_{n-1}, e_n \rangle$ in an operator network is defined as the number of edges of which p is comprised; hence, *length(p)=n*, assuming that p is composed of the consecutive edges e_1 to e_n. This parameter is an essential feature of each path, yet it does not refer to its nature, that is, if it is cycle-free or not. In reality, the number of cycles that a path contains is determined by the number of temporal loops in the path (i.e. the number of `pre` operators) and it characterizes its complexity.

So far, coverage analysis is performed in relation to a given value of path length $n \geq 2$; the number of cycles in the obtained paths is an implicit parameter according to n. As a result, only the complete paths[1] of length n are analyzed, regardless of the number of cycles. On the contrary, complete cyclic paths are of greater interest in practice, since their coverage strongly depends on the number of execution cycles and, consequently, on the test input sequence length. Usually, professionals are interested in measuring the coverage for a set of paths of a given number of cycles ($c \geq 0$) rather than a given path length. Noted that $c = 0$ denotes the set of complete cycle-free paths.

5.3 Path activation conditions in integration testing

The proposed approach consists in defining an approximated estimation of the coverage of the called nodes that will be used for the coverage analysis of the main node. In this way, the code coverage is effectively measured and the complexity is reduced.

The coverage approximation is made thanks to an abstraction of the called node, based on a new type of operator: the `NODE` operator. This new operator denotes the operator network corresponding to the called nodes and it is used to replace them while analyzing the coverage. The inputs (outputs) of the `NODE` operator are the inputs (outputs) of the replaced node. The set of paths beginning from an input e_i and ending to an output s_i in the operator network of the replaced node is represented by a single unit path $p = \langle e_i, s_i \rangle$ in the `NODE` operator as it is shown in Figure 5.2. The activation condition of p is a

[1] A path $p = \langle e_1, \ldots, e_n \rangle$ is complete if and only if e_1 is an input edge and e_n is an output edge. In other words, a complete path connects a program input with a program output.

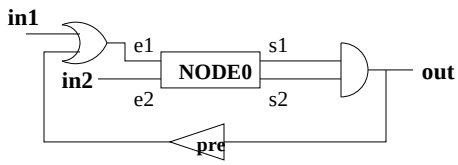


Figure 5.2: A LUSTRE node using a compound operator (global level: `NODE0` abstracted).

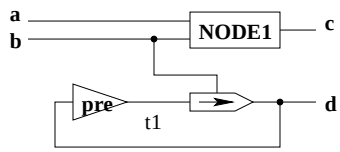


Figure 5.3: Expansion of `NODE0` in Figure 5.2.

combination of the activation conditions of the set of paths from the edge e_i to the edge s_i. As it is the case for every basic operator (boolean, arithmetic, temporal), an activation condition is associated with every unit path of a `NODE` operator.

In addition, it is possible that a called node contains a call to another node, as it is the case in Figure 5.3, forming a tree structure of called nodes. Each level of this structure corresponds to the *depth* of node integration. The paths to consider for testing and the associated activation conditions depend on this level. For example, in Figure 5.2, let $AC^0(p)$ be the activation condition of $p = \langle e_1, s_1 \rangle$ at level zero. Since `NODE0` calls `NODE1`, the following expression describes the correlation of activation conditions between the different levels of integration:

$$AC^0(\langle e_1, s_1 \rangle) = AC^1(\langle a, c \rangle) \tag{5.1}$$

Consequently, at level m, a path activation condition depends on the activation condition of level $m+1$, as shown in the following definition.

Definition 5.1. *Let E and S be the sets of inputs and outputs respectively of a* `NODE` *operator at level m so that there is an input $e_i \in E$ and an output $s_i \in S$. Let $n>0$ be any positive integer and $p_1, p_2, \ldots, p_k$ be the paths from the input e_i to the output s_i the length of which is lower or equal to n. The* ***abstract activation condition*** *of the unit path $p = \langle e_i, s_i \rangle$ for the depth m and the path length n is defined as follows:*

$$AC_n^m(p) = AC_n^{m+1}(p_1) \vee AC_n^{m+1}(p_2) \vee \cdots \vee AC_n^{m+1}(p_k) \tag{5.2}$$

Similarly to the activation condition definition for the basic operators of the language, the abstract activation condition indicates the propagation of the input value towards the output. Due to the disjunction of the activation conditions of the involved paths, at least one of the dependencies of s_i on e_i is taken into account (i.e. if at least one path of length lower or equal to n in the `NODE` operator is activated, then the unit path p is activated as well, $AC_n^m(p) = true$, meaning that the value of the output s_i is affected by the value of the input e_i of the `NODE` operator).

5.4 Extended structural coverage criteria

Similarly to the criteria definition for unit testing, we define three classes of path-based criteria to support operator networks containing compound operators (in addition to basic ones). The definition of each class takes into account three parameters:

1. The *depth of the integration*. As it has been noticed before, the depth determines the level where `NODE` operators will stop being unfolded (i.e. after this level, the abstract activation condition will be used).

2. The *length of the paths* from an input to an output. This parameter, already considered in the criteria definition presented in section 3.6, has a different meaning in presence of `NODE` operators in the program under test. Indeed, the length of the paths to cover varies whether the `NODE` operators are unfolded or not.

3. The *constraints* imposed on the paths that determine the way that a path should be covered and the name of the criterion (BC, ECC, or MCC).

5.4.1 Integration depth

The depth of node integration is a significant parameter specific to integration testing. Figure 5.4 shows the call graph of an application with several embedded nodes presented in [32]. Coverage analysis of a node with $depth = i$ indicates that every called node up to this level i is expanded, while called nodes of the following levels are integrated and replaced by a `NODE` operator[2]. For instance, in Figure 5.4, covering node #1 with:

[2]The main node, which actually contains all the other nodes corresponds to level 0.

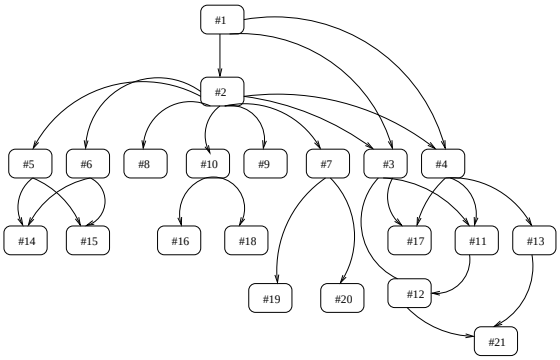


Figure 5.4: Structure of a large application..

- *depth* = 0 means that nodes #2, #3 and #4 (i.e. the set of nodes that are called by node #1) will be replaced by a `NODE` operator (i.e. they will not be expanded).
- *depth* = 1 means that only nodes #2, #3 and #4 will be expanded and all the other nodes will be replaced by a `NODE` operator (similarly for levels higher than 1).

The choice of the value for the integration depth may depends on the complexity of the application.

5.4.2 Path length

The length of the considered paths is the second parameter taken into account in the criteria definition. It is an important factor of the criteria complexity since it determines, in particular, the number of `pre` operators that will be tested (temporal loops). The introduction of `NODE` operators raises several issues:

1. The notion of the path length may be different for the same program according to the integration depth. For instance, in the node of Figure 5.2, we can identify one path $p = \langle in_2, s_2, out \rangle$ the length of which is 3, if we consider that `NODE0` is not unfolded (the integration depth is 0). But if the integration depth is 1, `NODE0` will be unfolded and, then, the previously considered path corresponds (Figure 5.3) either to the unit path $p_1 = \langle b, d \rangle$ (the length of which is 2) or to $p_2 = \langle b, d, t_1, d \rangle$ (the length of which is 4 or more, according to the number of loops that should be considered).

2. The computation of the abstract activation conditions associated with a `NODE` operator N (used when the latter is not expanded) depends on the length of the paths in N that have been taken into account (for instance, the abstract activation condition of `NODE0` in Figure 5.2 may be computed as the disjunction of the activation condition of $p_1 = \langle b, d \rangle$, $p_2 = \langle b, d, t_1, d \rangle$, $p_3 = \langle b, d, t_1, d, t_1, d \rangle$ and so on). There is no obvious way to determine the path length n to consider when a `NODE` operator abstract condition is computed. Indeed, this length depends on the integration depth as well as on the `NODE` operator structure and complexity (since, for instance, the activation of certain paths requires the execution of several temporal loops). This is why, ideally, the path length for the `NODE` operators abstract activation condition computation should be user-defined with regard to each node.

5.4.3 Criteria definition

From the above discussed issues, it appears that integration oriented coverage criteria must specify the integration depth, the path length to consider as well as the path length that will be used to compute the abstract conditions of the not unfolded `NODE` operators. Similarly to the criteria defined for unit testing in 3.6, we define three families of criteria, *iBC*, *iECC* and *iMCC* (*i* stands for "integration oriented"). Let m be the integration depth, P_n be the set of all the paths of length lower or equal to n for this depth, l be the maximum path length to be considered for the abstract activation condition computation, and $\mathcal{T}$ be a set of input sequences. $in\,(p)$ denotes the input of path p while e denotes one of its internal edges.

The basic coverage criterion (*iBC*) requires activating at least once all the paths of a given length and for a given depth of integration.

Definition 5.2. *The operator network is covered according to the basic coverage criterion* $iBC^m_{n,l}$ *if and only if:*
$\forall p \in P_n,\ \exists t \in \mathcal{T}\colon\ AC^m_l\,(p) = true.$

The elementary conditions criterion (*iECC*) requires activating a path, exercising both the possible values for its boolean input, *true* and *false*.

Definition 5.3. *The operator network is covered according to the elementary conditions criterion* $iECC^m_{n,l}$ *if and only if:*
$\forall p \in P_n,\ \exists t \in \mathcal{T}\colon\ in\,(p) \wedge AC^m_l\,(p) = true \text{ and } not\,(in\,(p)) \wedge AC^m_l\,(p) = true.$

The multiple conditions coverage criterion ($iMCC$) requires the paths to be activated for every possible value of the internal boolean edges.

Definition 5.4. *The operator network is covered according to the multiple conditions criterion* $iMCC^m_{n,l}$ *if and only if:*
$\forall p \in P_n$, $\forall e \in p$, $\exists t \in \mathcal{T}$: $e \wedge AC^m_l(p) = true$ *and not* $(e) \wedge AC^m_l(p) = true$.

5.5 Subsumption relations

Keeping the integration depth m fixed, the longer the paths that a criterion examines, the stronger the criterion is. In other words, for each one of the three criteria, it is:

$$iC^m_{n-1,l} \subseteq iC^m_{n,l} \tag{5.3}$$

Since the set of all paths of length lower or equal to n (P_n) can be obtained by the union of all the paths of length lower or equal to *n-1* (P_{n-1}) with those of length lower or equal to n (P_n), the set of the activation conditions of the class P_{n-1} is a subset of the set of the activation conditions of the class P_n. Therefore, if an input sequence satisfies the activation conditions of a criterion for a set of paths P_n, it necessarily satisfies the activation conditions of a criterion for the previous class (i.e. the set of paths P_{n-1}).

In addition, when the path length n remains unchanged at the level m, it could be expected that exploring longer paths inside the integrated nodes (for abstract conditions computation) would enhance the criterion. However, this is not true and decreasing the length of the considered paths inside a `NODE` operators may actually not affect accordingly the abstract activation conditions. Increasing the parameter l for the locally considered paths definitely results in a more thorough coverage analysis of the integrated nodes, though the abstract activation condition is issued from the disjunction of set of activation conditions and only one of them affects the final value. Thus, it is difficult to reach a definitive conclusion about the power of the criteria with regard to the length of local paths. On some occasions, analyzing longer local paths may improve coverage (as compared to shorter local paths), but it is not guaranteed that the opposite may not happen. Such a conclusion depends on the activation

condition that is "transmitted" by the disjunction and, to a certain extent, on the test input data (and consequently, on the way the latter are generated).

Furthermore, we can define the above criteria considering that different values for the path length are considered for each internal node. Thus, $iBC^m_{n,(l_0,l_1,l_2)}$ would denote the integration oriented basic coverage criterion, for the integration depth m, the path length n and the path lengths l_0, l_1 and l_2 for the three `NODE` operators. In such a case, the subsumption relation is a partial order expressed as follows:

$$\begin{array}{ccc} & iBC^m_{n,(l_0+1,l_1+1,l_2+1)} & \\ \swarrow & \downarrow & \searrow \\ iBC^m_{n,(l_0+1,l_1,l_2)} & iBC^m_{n,(l_0,l_1+1,l_2)} & iBC^m_{n,(l_0,l_1,l_2+1)} \\ \searrow & \downarrow & \swarrow \\ & iBC^m_{n,(l_0,l_1,l_2)} & \end{array}$$

With regard to the integration depth, there is no obvious subsumption relation between the integration oriented criteria. This is due to the additional parameters used in their definition: the integration depth and the path length used for computing the abstract conditions. Consider, for instance, $iBC^m_{n,l}$ and $iBC^{m-1}_{n,l}$. The satisfaction of the former requires to unfold some `NODE` operators that are abstracted in the latter. But $iBC^m_{n,l}$ does not ensure the coverage of the sub-paths of these operators covered by $iBC^{m-1}_{n,l}$. Indeed, a path of length l in an unfolded `NODE` operator will be covered by $iBC^m_{n,l}$ only if it is part of a sub-path of length lower or equal to n.

Consider the example of Figure 5.2. $iBC^1_{4,l}$ requires the coverage of the following paths inside `NODE0`: $p_1 = \langle b, d, t_1, d\rangle$, $p_2 = \langle a, c\rangle$, $p_3 = \langle b, c\rangle$ (i.e. all the paths of length lower or equal to 4). The satisfaction of $iBC^0_{4,l}$ is not assured by the satisfaction of $iBC^1_{4,l}$, because if at level 0 we consider $l = 3$, then the satisfaction of $iBC^1_{3,l}$ is required. Because of the changed n, the comparison between $iBC^1_{3,l}$ and $iBC^0_{4,l}$ is not possible.

Example 5.5. Let us consider the LUSTRE node [22] shown in Figure 5.5, a device that monitors response times. The output *alarm* is raised when no *reset* signal has occurred for a given time since the last *set*, this time being given as a number *delay* of basic clock cycles. This node uses the node `WD1`[3] (Figure 5.5(b)), a simpler version of the former, by providing it with an appropriate *deadline* parameter: on reception of *set*, a register is initialized which is then

[3]The node `WD1` receives the input signals *set*, *reset* and *deadline* and returns *alarm* at the output. *alarm* is raised whenever a *deadline* occurs and the last received signal was *set*.

decremented. *deadline* occurs when the register value reaches zero; it is the output of node `EDGE` (Figure 5.5(c)) which returns true at each rising edge of its input.

`WD1` and `EDGE` are compound operators, hence the corresponding operator networks are replaced by two `NODE` operators with respective inputs and outputs. In this case, there is only one level of integration; the main node contains two direct calls to different nodes. In the analysis that follows we consider that $depth = 0$ (i.e. m=0 in the criteria definition) (for $depth = 1$, nodes `WD1` and `EDGE` would be fully expanded, hence there would be no integration). Let us assume the paths of length lower or equal to 6 (i.e. n=6) beginning from every input of `WD2` to the output:

$p_1 = \langle set, alarm \rangle$
$p_2 = \langle set, D3, D1, D2, deadline, alarm \rangle$
$p_3 = \langle reset, alarm \rangle$
$p_4 = \langle delay, D3, D1, D2, deadline, alarm \rangle$

First of all, calculating the condition that activates p_1 requires the node `WD1` to be locally expanded (Figure 5.5(b)). In order to best demonstrate the way that the criteria can be deployed, we use different values of path length for each internal node, considering that at most one temporal loop (i.e. one cycle) is taken into account in the computation of the length of a complete path. Hence, inside `WD1`, we consider the complete paths of length 9 (i.e. l=9) from *set* to *alarm*:

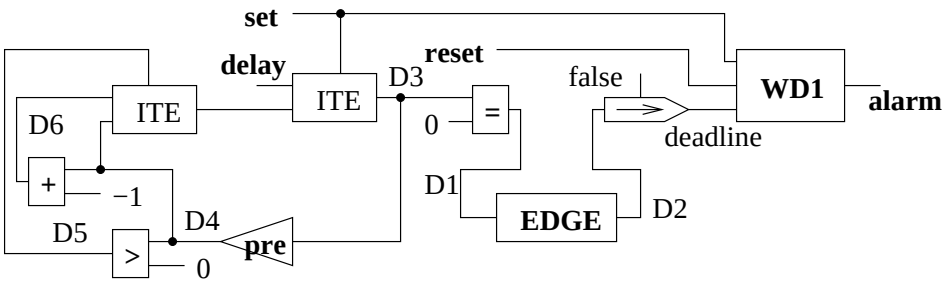
$p_{11} = \langle set, L1, alarm \rangle$
$p_{12} = \langle set, L1, L4, L3, L2, L1, alarm \rangle$
$p_{13} = \langle set, L2, L1, L4, L3, L2, L1, alarm \rangle$.

The corresponding activation conditions are:

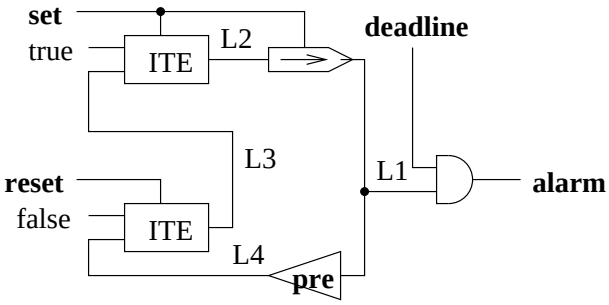
$AC_9^1\,(p_{11}) = (true \text{ -> } false)\ and\ (not\,(L1)\ or\, deadline)$

$AC_9^1\,(p_{12}) = (false \text{ ->} pre\,(true \text{ -> } false))\ and\ (not\,(reset))$
$and\ (not\,(set))\ and\ (false \text{ ->} true)\ and\ (not\,(L1)\ or\, deadline)$
$\Rightarrow AC_9^1\,(p_{12}) = false\text{->}pre\,(true)\ and\, not\,(reset)$
$and\, not\,(set)\ and\ (not\,(L1)\ or\, deadline)$

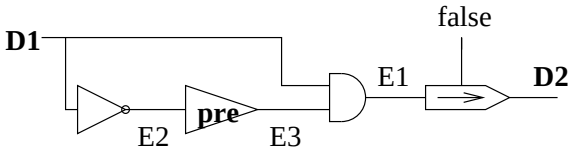
$AC_9^1\,(p_{13}) = (false \text{ ->} pre\,(false \text{ -> } true))\ and\ (not\,(reset))$
$and\ (not\,(set))\ and\ (false \text{ ->} true)\ and\ (not\,(L1)\ or\, deadline)$



(a) Operator network for the node `WD2` (global level).



(b) `WD1` expanded (local level).



(c) `EDGE` expanded (local level).

Figure 5.5: Operator network for the node `WD2` (level 0) and the called nodes `WD1` and `EDGE` (level 1).

$\Rightarrow AC_9^1(p_{13}) = AC_9^1(p_{12})$.

The disjunction of the above conditions results in the activation condition for the path p_1, $AC_7^0(p_1)$:
$AC_7^0(p_1) = AC_9^1(p_{11})\ or\ AC_9^1(p_{12})\ or\ AC_9^1(p_{13})$.

Similarly for the path p_3 and l=9, the obtained paths and the combined activation condition are:
$p_{31} = \langle reset, L3, L2, L1, alarm\rangle$
$p_{32} = \langle reset, L3, L2, L1, L4, L3, L2, L1, alarm\rangle$

$AC_9^1(p_{31}) = false\text{->}not\,(set)\ and\ (not\,(L1)\ or\,deadline)$

$AC_9^1(p_{32}) = (false\text{->}pre\,(not\,(set)\ and\ (false\text{ -> }true)))\ and$
$not\,(reset)\ and\,not\,(set)\ and\ (false\text{->}true)\ and\ (not\,(L1)\ or\,deadline)$

$\Rightarrow AC_7^0(p_3) = AC_9^1(p_{31})\ or\ AC_9^1(p_{32})$.

For the paths p_2 and p_4, determining the activation conditions requires the analysis of unit sub-paths $p_{21} = p_{41} = \langle D1, D2\rangle$ and $p_{22} = p_{42} = \langle deadline, alarm\rangle$, which also involves the local expansion of node `EDGE` (Figure 5.5(c)). Inside the node `EDGE` and for l=5, it is:
$p_{211} = p_{411} = \langle D1, E1, D2\rangle$
$p_{212} = p_{412} = \langle D1, E2, E3, E1, D2\rangle$

$\Rightarrow AC_7^0(p_{21}) = AC_7^0(p_{41}) = AC_5^1(p_{211})\ or\ AC_5^1(p_{212})$.

The activation conditions that follow are:
$AC_5^1(p_{211}) = AC_5^1(p_{411}) = false\text{->}not\,(D1)\ or\ E3$
$AC_5^1(p_{212}) = AC_5^1(p_{412}) = false\text{->}pre\,(true)\ and\ (not\,(E3)\ or\ D1)$

$AC^0(p_{22}) = AC^0(p_{42}) = not\,(deadline)\ or\ L1$

Number of paths

In this example and considering for the sake of simplicity the rather simplified case in which only cycle-free paths at the external level are taken into account, the above integration theory results in 4 such paths in the main node `WD2` (the

paths p_1, p_2, p_3 and p_4). If nodes `WD1` and `EDGE` were used in their expanded version, we would have to deal with at least 7 paths[4], whose length would increase according to the number of temporal loops. For instance, in case we consider paths of maximum length 10, coverage analysis (without node integration) in node `WD2` results in 9 paths while for length 20 (including 3 temporal loops) the number of paths is increased to 19. Hence, in this small example, we save three paths in the simplest case; this fact could be a good outcome for the proposed abstraction and could imply that for more realistic and complicated systems a more significant reduction might be possible. In Chapter 6, we examine a case study of a flight alarm management system and the gain in the number of paths is rather important. Obviously, the dependence of path length on the number of paths according to each node level is inevitable and should be further investigated to obtain more conclusive results.

[4]Two paths from *set* to *alarm* passing through `WD1` and two passing through `EDGE`, one path from *reset* to *alarm* and two paths from *delay* to *alarm* passing through `EDGE`.

Chapter 6

Experimental evaluation

In this chapter, we describe the application of the proposed coverage metrics to a case study derived from an avionics application. The application size and complexity are suitable so that the obtained results provide a preliminary validation of the defined criteria.

Dans ce chapitre, nous décrivons une étude de cas extraite d'une application du domaine de l'avionique sur laquelle nous avons effectué les mesures de couverture proposées. La taille et la complexité de l'application sont telles que les résultats obtenus fournissent une première validation de la pertinence des critères définis.

6.1 Case study: an alarm management software

This section treats the evaluation of our approach about node integration. To this end, we have used several relatively small, academic examples, like the one used in Section 5.1. Here, we present the results of an industrial case study concerning an alarm management system developed in SCADE.

6.1.1 Objectives

The principal objective of this experiment is to illustrate the usage and the applicability of the integration-oriented coverage criteria to big and complex systems that are really used in practice. To this end, we firstly looked into the prospective gain in the number of paths with reference to the coverage

criteria that do not require full node expansion. More specifically, we examined the number of paths that the node integration yields, using as a reference the respective number when no node integration is used. Assuming that all called nodes are expanded, we used LUSTRUCTU to automatically compute the paths and the associated activation conditions. Then, we compared the number of the calculated paths with the respective number of paths when node integration is used. In general, the number of the calculated paths should decrease as longer paths are considered. In addition, for the extended coverage metrics that take into account the integration of locally called nodes, we observed the number of the calculated paths and activation conditions in relation to the required system resources. An important feature of the proposed criteria is their capability in precisely evaluating the progress of the obtained coverage. In general, this means that a slight increase in the criteria satisfaction ratio should correspond to a proportionately small increase in the testing effort.

This speculation points to the second objective of this case study which is to study the required testing effort to meet the criteria with regard to the criteria power, that is the selected criterion (iBC, iECC or iMCC) and the path complexity. The testing effort is measured in terms of the required length of the test input sequences towards the criteria satisfaction. In order to obtain fair and unbiased results, test input data were randomly generated.

Furthermore, we aim at assessing the capability of the proposed criteria in detecting faults in the program under test. Mutation testing was used to simulate various faults in the program. In particular, a set of mutation operators was defined and several mutants were automatically generated. Then, the mutants and the coverage nodes were executed over the same test input data and the mutation score (ratio of killed mutants) was compared with the coverage ratio.

6.2 System description

This case study is concerned with a software component specified in LUSTRE/SCADE that models the operation of an alarm in a flight control system used in [33]. The main function of this component is to send a warning signal according to the flight parameters calculated by the system, the parameters from some sensors that provide the aircraft status as well as the parameters provided by the pilot. Informally, the alarm is set off when the difference between a value provided by the pilot, the theoretic value calculated by the center of gravity and

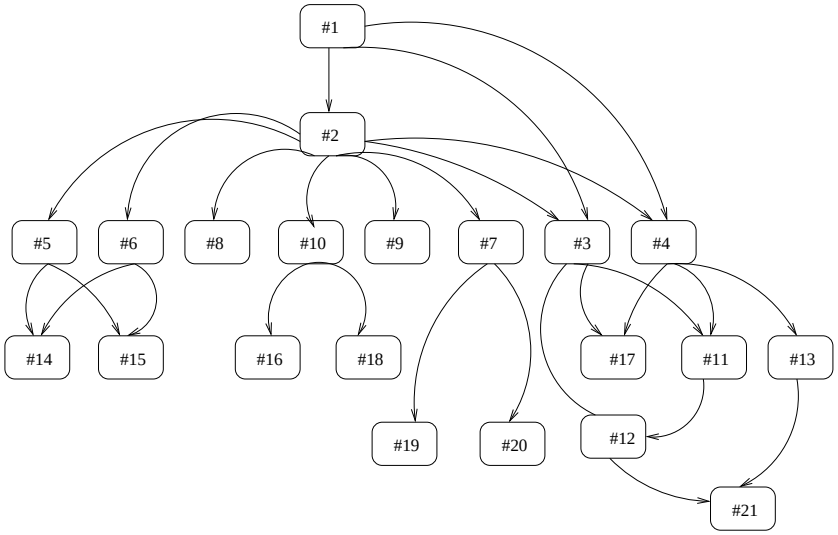


Figure 6.1: Call graph for the alarm management software component.

the value of the real position exceeds 1.5^o.

The system implementation forms a hierarchical structure. In fact, the component is composed of 21 LUSTRE nodes generated by SCADE. The main node comprises the calls to the auxiliary nodes, as Figure 6.1 depicts.

Table 6.1 provides information concerning the system size and complexity. For each module of the application, the first three columns concern the number of lines of LUSTRE code and the number of the node inputs and outputs respectively. For instance, the main node (#1) totally contains 830 lines of LUSTRE code with 29 input variables (23 of which are boolean and 6 are integer) and 13 output variables (10 of which are boolean and 3 are integer). This module implements a property specified in the two alarms that are simulated by the modules #2 and #3. The last two columns refer to the associated operator network of each LUSTRE node and give the number of edges and operators in each one of them; this data corresponds to a version of the application in which all nodes are expanded and only basic LUSTRE operators are used. That is, the operator network of the main node comprises 190 basic operators linked to each other by 275 edges. A complete view of the system is shown Figure 6.2, in which the internal modules are represented by black boxes for the sake of simplicity.

Nodes #3 to #21 are of lower complexity; they are composed of boolean variables and computations without temporal loops. Therefore, the total num-

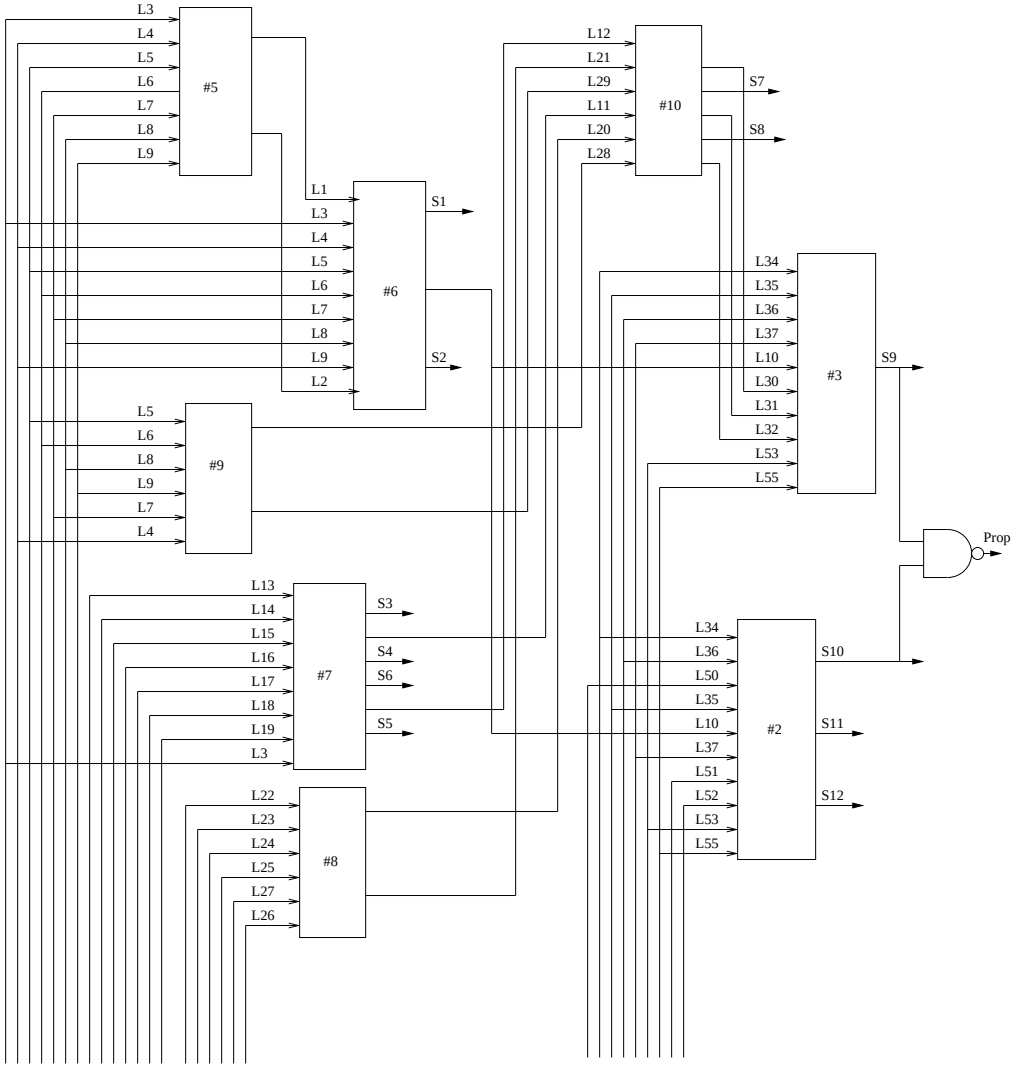


Figure 6.2: Operator network of the alarm management system.

	Code			**Operator Network**	
node	*LOC*	*inputs*	*outputs*	*edges*	*operators*
#1	830	29	13	275	190
#2	148	10	3	52	32
#3	157	10	1	59	37
#4	148	10	3	52	32
#5	98	7	2	33	22
#6	98	9	2	33	22
#7	67	8	6	60	32
#8	40	6	2	12	6
#9	40	6	2	12	6
#10	132	6	5	36	24
#11	50	3	1	16	9
#12	31	2	1	11	5
#13	30	2	1	11	5
#14	16	2	1	3	1
#15	16	2	1	3	1
#16	16	2	1	3	1
#17	21	1	1	8	5
#18	19	1	1	5	3
#19	8	1	1	23	12
#20	8	1	1	17	6
#21	4	3	1	5	2

Table 6.1: Size and complexity of the alarm management system.

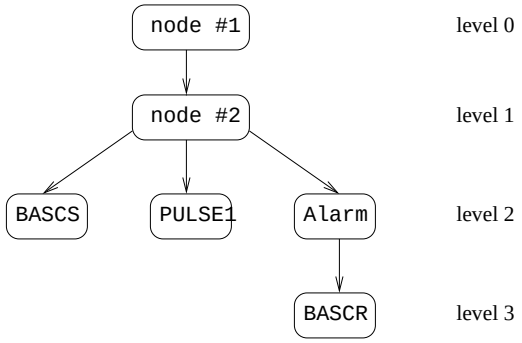


Figure 6.3: Call graph for node #2.

ber of paths in these nodes is fixed for a specific (maximum) path length. It is hence rather simple to estimate their coverage. On the contrary, node #2 contains several nested temporal loops which in fact renders the whole system so complex. Indeed, this node locally calls 3 nodes, one of which calls one more, hence forming two more levels of integration. Thus, examining thoroughly node #2 is more interesting from the point of view of node integration. For this reason, during this experiment, we concentrated on node #2. We present in the following the results of the integration analysis conducted over this node that models the function of a system alarm. Tests were performed on a Linux Fedora 9, Intel Pentium 2GHz and 1GB of memory.

In the coverage analysis that follows, we go through node #2 and the nodes that the latter locally calls, as these are depicted in the call graph of Figure 6.3.

6.3 Number of paths

Let N be the number of paths of length lower or equal to $n \geq 2$ in an operator network. Recall that we only consider complete paths connecting the program inputs with the program outputs. Then, we can safely assume that the number of paths N depends on two factors:

- The program size and subsequently the operator network size (number of operators and edges). Indeed, the number of paths increases proportionally to the operator network size.
- The program complexity in point of the presence of cyclic paths. In fact, cyclic paths presume longer paths, which consequently presumes more

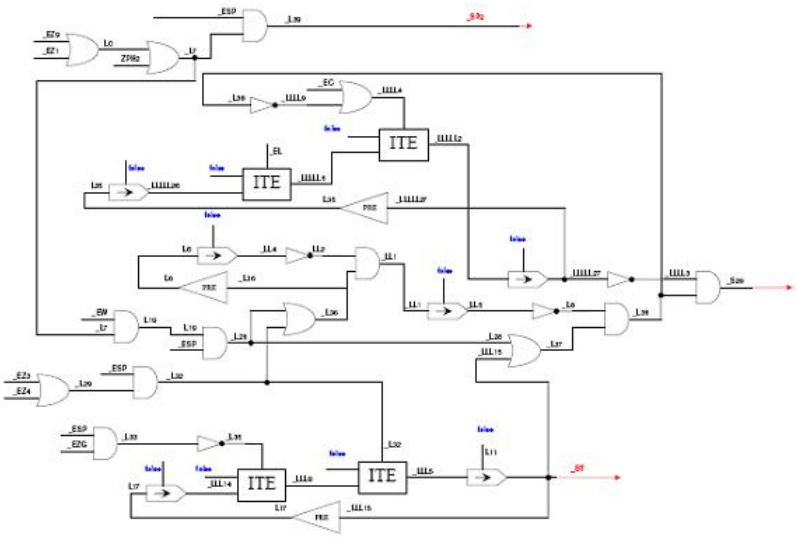


Figure 6.4: Operator network of node #2 (the called nodes are expanded).

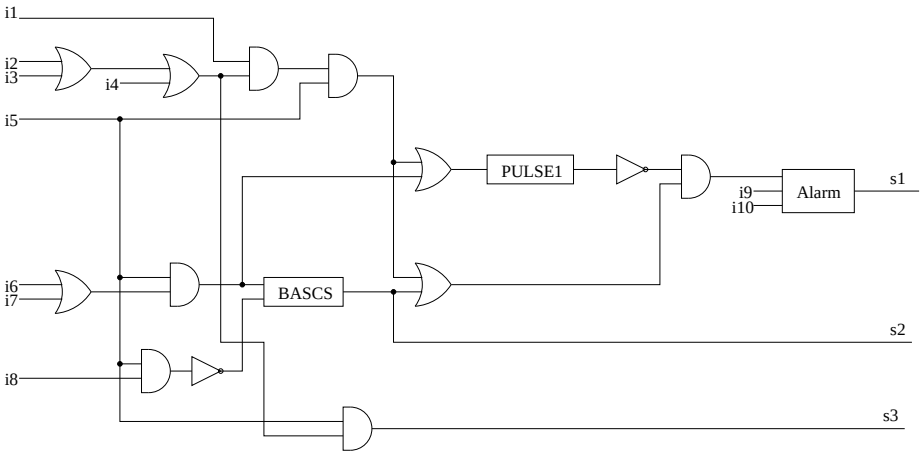


Figure 6.5: Operator network of node #2 (the called nodes are abstracted).

paths.

It is difficult to imply a precise estimation of the number of paths considering only the program size or only the program complexity. However, the latter provides a more objective and concrete idea of the number of paths, since the coverage of cyclic paths strongly depends on the number of execution cycles and, consequently, on the test input sequences length. Moreover, in practice, professionals are usually interested in measuring the coverage for a set of paths of a given number of cycles ($c \geq 0$)[1] rather than a given path length. Therefore, in the analysis that follows, we will consider various sets of complete paths in an operator network according to the number of cycles c contained in them.

When the integration-oriented approach is used (i.e. the called nodes are simulated by black boxes), complete paths inside the called nodes are represented by unit paths in the calling node. Therefore, the number of paths at the global level is reduced, as compared to the number of paths found in case no node integration is used. Table 6.2 illustrates how the number of paths found in node #2 increases in relation to the number of cycles taken into account. The second column refers to the version of node #2 where `PULSE1`, `BASCS`, `Alarm` are expanded (cf. Figure 6.4). On the contrary, in case of node integration, these nodes are replaced by `NODE` operators, hence they are abstracted. Since node #2 does not contain any temporal loop, the number of cycles is constant regardless of the complexity of paths considered inside the called nodes. In comparison to the full expansion of the locally called nodes, the decrease in the number of paths is significant, especially in case of long cyclic paths.

On the other hand, the number of activation conditions that must be satisfied depends on the selected criterion (iBC, iECC, iMCC). The stronger the criterion, the more constraints are imposed on the paths. If N is the number of paths of length lower or equal to n in an operator network, m_k is the number of paths of length lower or equal to $k \leq n$, then Table 6.3 summarizes the number of activation conditions M according to the criteria power.

Table 6.4 juxtaposes the number of activation conditions for node #2 in case the integration-oriented approach is used and the corresponding number in case no node integration is used. For the sake of simplicity, data correspond to complete paths with at most 1 and 3 cycles. Between the two approaches, there is remarkable decrease in the number of activation conditions, particularly for the multiple conditions criterion. Noted that at this level of integration (level

[1] Noted that $c = 0$ denotes the set of complete cycle-free paths.

	# paths	
# cycles	no node integration	with node integration
0	25	25
1	50	25
2	88	25
3	131	25
4	179	25
5	232	25
6	290	25
7	353	25
8	421	25
9	494	25
10	572	25

Table 6.2: Number of paths in node #2 w.r.t. the number of cycles.

Criterion	#ACs
iBC	$M_{iBC} = N$
iECC	$M_{iECC} = 2 * N$
iMCC	$M_{iMCC} = 2 * \sum_{k=2}^{k=n} ((k-1) * m_k)$

Table 6.3: Number of activation conditions (M) w.r.t. the criteria power.

1 in Figure 6.3), there are only cycle-free paths (i.e. no temporal loops), hence the number of activation conditions shows no change.

As far as the required time to calculate the activation conditions is concerned, this is relatively negligible; a few seconds (maximum 2 minutes) were needed to calculate complete paths with maximum of 10 cycles and the associated activation conditions. Even for the multiple conditions criterion, this calculation remains minor, considering that the number of paths to be analyzed is affordable.

The above observations suggest that the extended criteria are useful for measuring the coverage of large-scale programs. The reduction in the number of paths and subsequently to the number of activation conditions indicates that covering this type of programs is feasible. To gain a better understanding of

	# ACs					
# cycles	iBC	iECC	iMCC	BC	ECC	MCC
1	29	58	342	50	100	1330
3	29	58	342	131	262	4974

Table 6.4: Number of activation conditions for node #2.

Length	1	2	3	4	5	6	7	8	9	10
iBC	10,68	42,41	55,17	61,37	66,20	71,03	73,79	78,27	81,37	92,41
iECC	2,93	6,89	15,00	20,86	27,58	37,06	45,34	51,72	56,03	61,20
iMCC	1, 75	4,91	6,22	11,66	16,31	20,38	25,23	32,01	34,32	39,97

Table 6.5: Coverage ratio for test sequences of length 1 to 10.

the proposed approach, in the following section, we take a closer look at the relation between the required testing effort and the criteria satisfaction ratio.

6.4 Testing effort w.r.t. criteria satisfaction

The required effort to generate test input sequences satisfying a specific criterion not only depends on the test selection technique but also on human testers. Test data are commonly generated in practice (for instance in DO-178B) based on system functional requirements, regardless of the system structural characteristics. Thus, it is hard to evaluate a tester's attempt to build a test data set that meets a coverage criterion, since test input selection is strictly determined by subjective factors.

With the intention of assessing the required testing effort in an independent and unbiased way, we used random test generation irrespective of any functional or structural requirements. Even though random testing is a rather ineffective and expensive testing technique, it is quite convenient with respect to the coverage criteria.

In order to ensure the equity of the generated test data, we used different seed values, linearly varied, to initialize the generator. A test campaign comprises several test sets of variable sequence length. Based on the theoretic assumption that the more complicated a criterion, the longer the required test sequences are in order to satisfy it, we considered test sequences of length equal to 1 up to 1000.

The general procedure of measuring the coverage was organized as follows:

1. Random generation of test campaigns.

2. Coverage node execution over each one of the input sequences of every test campaign and computation of the corresponding coverage ratio.

3. Computation of the average of the obtained results.

Length	10	20	30	40	50	60	70	80	90	100
iBC	92,41	97,93	99,31	**100**						
iECC	61,20	72,41	78,44	81,55	83,27	83,96	84,65	84,82	84,82	85,34
iMCC	39,97	64,64	74,12	82,22	82,89	84,50	86,40	86,49	86,75	87,39

Table 6.6: Coverage ratio for test sequences of length 10 to 100.

Length	100	200	300	400	500	600	700	800	900	1000
iECC	85,34	87,41	87,58	87,75	87,75	87,75	87,93	87,93	87,93	87,93
iMCC	87,39	88,01	88,15	88,45	88,74	88,74	88,74	88,74	88,88	88,88

Table 6.7: Coverage ratio for test sequences of length 100 to 1000.

Data in Tables 6.5, 6.6 and 6.7 and the corresponding graphic representation in Figure 6.6 show the relation between the criteria coverage ratio and the required test sequence length for the node #2. Noted that these results consider 3-cycle complete paths in this node. Obviously, coverage is in proportion to the test input sequence length for all the three criteria. This means that longer sequences achieve higher coverage ratios and conversely.

Moreover, the basic criterion, which is the simpler among all, is satisfied generally fast; test input sequences of length 40 are sufficient to reach 100% coverage. On the contrary, the elementary and multiple conditions criteria never attain total coverage. The coverage ratio for these two criteria converges at a certain threshold around of which it progresses very slowly. Additionally, for short test sequences, the coverage ratio increases rapidly, independently of the criterion complexity. As the sequences become longer, the progress in the coverage ratio gradually falls off until it reaches the above threshold, which differs according to the criterion complexity.

Another important observation is that the required testing effort to a criterion satisfaction is rather precise. For example, in order to achieve 80% of the criteria satisfaction, sequences of length 9 must be generated for the iBC criterion, while for the iECC and iMCC criteria sequences of length 40 or more are required.

The fact that the two stronger criteria (iECC and iMCC) are never completely met is partially due to the randomly generated test data; especially built test cases by experient testers could possibly complete the remaining coverage. Another reason is related to the program structure with regard to the constraints involved in the criteria. In other words, the incomplete coverage occurs because of some infeasible paths. Recall that an infeasible path is a path which

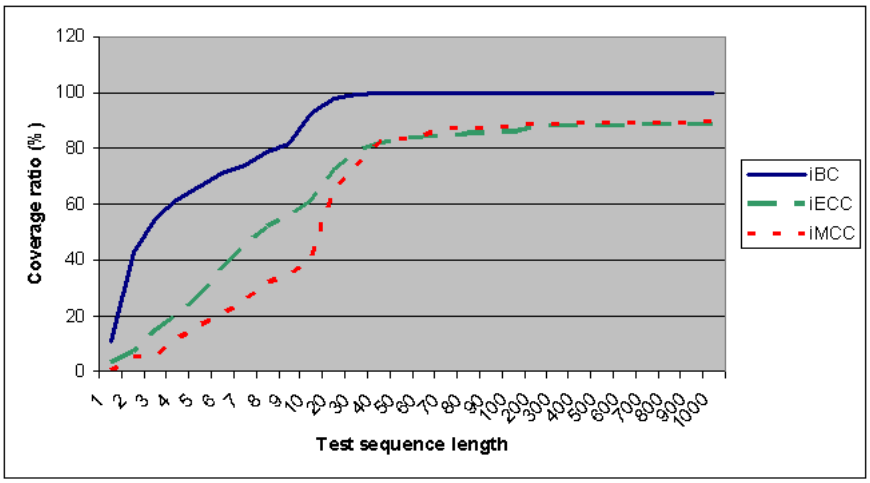


Figure 6.6: Coverage ratio for test sequence length 1 to 1000.

is never executed by any test cases, hence it can never be covered. The total satisfaction of the basic criterion implies the absence of infeasible paths in terms of basic coverage. By contrast, the fact that the elementary and the multiple conditions coverage are not fully satisfied, even for test sequences of 1000 steps, might be attributed to infeasible activation conditions (i.e. conditions that can never be fulfilled) or to rare combinations of the input values (i.e. if some input variables are not assigned to a specific value at the first execution step, then the corresponding paths are never activated during test execution).

6.5 Fault detection capability

Mutation testing is a widely used technique as a means of estimating the capability of a test set in finding out faults in the program under test. Taking into account the nature of LUSTRE/SCADE applications, we used operators changes to model program faults. More precisely, we defined a set of mutant operators (presented in Table 6.8) and every LUSTRE operator in the original program was replaced by one mutant operator so that the obtained program (the mutant) is syntactically correct. With reference to Table 6.8, a simple mutant generator was built that produced automatically the set of program mutants; since we are considering node integration and called nodes are simulated by black boxes, the replacements of operators were performed exclusively at the level of node #2 (cf.

operator	mutant
`not`	`pre`, [*delete*]
`and`	`or`, `fby`
`or`	`and`, `fby`
`pre`	`not`, [*delete*]
relational	relational
arithmetic	arithmetic

Table 6.8: Set of mutant operators.

Length	1	2	3	4	5	6	7	8	9	10
iBC	10,68	42,41	55,17	61,37	66,20	71,03	73,79	78,27	81,37	92,41
iECC	2,93	6,89	15,00	20,86	27,58	37,06	45,34	51,72	56,03	61,20
iMCC	1, 75	4,91	6,22	11,66	16,31	20,38	25,23	32,01	34,32	39,97
mutation score	16,15	32,30	38,46	45,38	52,30	62,30	66,92	70,76	72,30	73,84

Table 6.9: Coverage ratio VS Mutation score (test sequence length 1-10).

Figure 6.5), that is no modification was introduced inside nodes `PULSE1`, `BASCS` and `Alarm`. In this way, we obtained 25 mutants of the original program, for which mutation score[2] was calculated and compared with the criteria coverage ratio.

The results are summarized in Tables 6.9, 6.10 and 6.11; the curves of the achieved coverage ratio in comparison to the mutation score are depicted in Figure 6.7. It is evident that there is a correlation between the criteria satisfaction ratio and the number of killed mutants, which holds for all three criteria. The fact that the basic coverage criterion is completely satisfied does not indicate that the criterion is prone to detecting faults in the program; instead, it is rather weak. On the contrary, the correlation is stronger for the elementary and the multiple conditions criteria, which proves their capability in discovering common errors in a LUSTRE program.

[2]Mutation score is defined by the formula: $mutation\,score = \frac{number\,of\,killed\,mutants}{number\,of\,mutants}$.

Length	10	20	30	40	50	60	70	80	90	100
iBC	92,41	97,93	99,31	**100**						
iECC	61,20	72,41	78,44	81,55	83,27	83,96	84,65	84,82	84,82	85,34
iMCC	39,97	64,64	74,12	82,22	82,89	84,50	86,40	86,49	86,75	87,39
mutation score	73,84	89,23	92,30	93,84	93,84	93,84	93,84	93,84	96,92	97,69

Table 6.10: Coverage ratio VS Mutation score (test sequence length 10-100).

Length	100	200	300	400	500	600	700	800	900	1000
iECC	85,34	87,41	87,58	87,75	87,75	87,75	87,93	87,93	87,93	87,93
iMCC	87,39	88,01	88,15	88,45	88,74	88,74	88,74	88,74	88,88	88,88
mutation score	97,69	98,46	**100**							

Table 6.11: Coverage ratio VS Mutation score (test sequence length 100-1000).

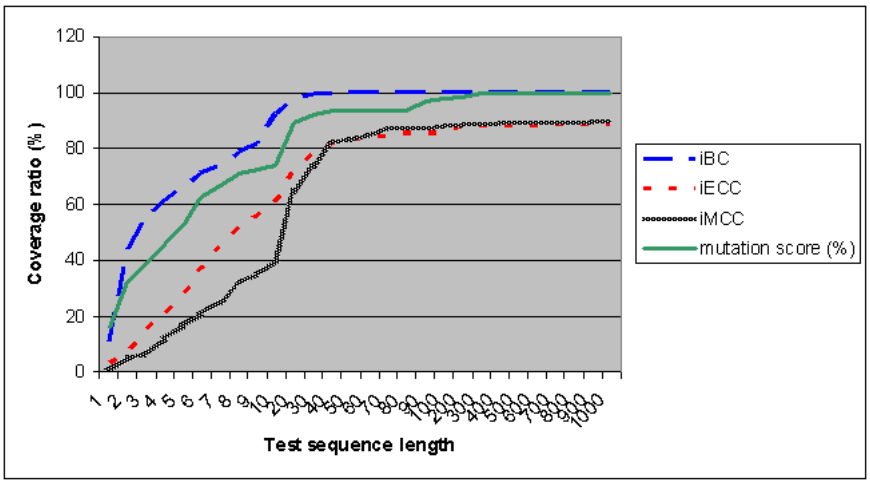


Figure 6.7: Coverage ratio VS Mutation score (test sequence length 1-1000).

6.6 Concluding remarks

The extension of the coverage criteria mainly aims at providing a family of coverage criteria that are in practice applicable to large-scale LUSTRE/SCADE applications. This case study showed that iBC, iECC and iMCC are appropriate to this purpose as far as the computation of the activation conditions for various values of the test sequence length is concerned. Besides, the correlation between the achieved coverage ratio and the mutation score demonstrates the capability of the proposed criteria in revealing errors in a program.

Part III

Test generation

Chapter 7

Automatic test data generation with Lutess

As described in Section 3.5, LUTESS is a black-box testing tool designed for testing reactive software by automatically generating test data sequences. The basic idea behind the testing approach using LUTESS lies in properly modeling the system environment in order to guide the test data generator to produce meaningful and valid test input sequences. This chapter proposes the use of LUTESS in the framework of a testing methodology that we have defined [48], describing the required steps and the guidelines in the construction of a test model. First, we show how this approach can be applied to a rather realistic case study, the steam boiler control system. Then, we study the applicability and the scalability of this task of modeling on a real-world industrial application from the field of avionics, the land gear control system of a military aircraft.

Comme nous avons discuté dans la Section 3.5, LUTESS est un outil de test "boîte noire" qui a été développé pour le test des logiciels réactifs permettant la génération automatique de séquences de test. L'idée fondamentale derrière l'approche de test avec LUTESS consiste à modéliser correctement l'environnement du système afin de guider le générateur vers la génération des jeux de test significatifs et valides. Ce chapitre propose l'utilisation de cet outil dans le cadre d'une méthodologie que nous avons définie [48] en décrivant les étapes nécessaires à la construction d'un modèle de test. Tout d'abord, nous présentons comment cette approche peut être employée sur une étude de cas

de taille et de complexité moyenne, le contrôleur de niveau d'eau dans une chaudière. Ensuite, nous étudions l'applicabilité et l'extensibilité de cette tâche de modélisation aux systèmes de plus grande échelle, en utilisant une application industrielle du domaine de l'avionique.

7.1 Testing methodology

Testing a reactive system using LUTESS requires modeling the external environment, that is, expressing the conditions that the system inputs should invarianltly satisfy i.e the hypotheses under which the software is designed. The model of the external environment, the so called *test model*, is built based on the system specification document. The informal requirements describing the system environment are extracted and transformed into LUSTRE formal expressions. In other words, the test model consists of all the constraints that describe the correct operation of the system under test. The goal is to obtain valid and suitable test suites in line with the system specification and the test objective. The environment description is made in a new LUSTRE file, the *testnode*, as described in Section 3.5.1. Different testing techniques (conditional probabilities, safety-property guided testing) can be used, through some specially introduced operators, to guide the test data generation process, to avoid unrealistic or unreasonable test cases, or to reach suspicious situations.

Although the test models are specific to the program under test, we claim that the modeling and testing process can follow a progressively incremental approach. Such a testing approach of test data generation according to the environment description not only provides the means to adequately test a system but also guides the system towards different modes of operation as well as the detection of faults. Moreover, the more advanced a test model is, the more thoroughly the system is tested; as a result, the testing process is automated as much as possible without requiring further intervention by the tester.

In Figure 7.1, we recall the general form of a testnode for the sake of clarity, previously presented in Section 3.5.1. In reference to Figure 7.1, the basic steps of the LUTESS testing methodology are the following:

1. **Domain definition**: Definition of the domain for integer inputs, so that the generator avoids producing any possible integer value with no interest for testing (e.g. an integer input corresponding to a temperature value in Celsius degrees could not be lower than 270).

```
testnode Env(<SUT outputs>) returns (<SUT inputs>);
var <local variables>;
let
   environment(Ec_1); ... environment(Ec_n);
   prob(C_1, E_1, P_1); ... prob(C_m, E_m, P_m);
   safeprop(Sp_1, Sp_2, ..., Sp_k);
   hypothesis(H_1, H_2, ..., H_l);
   <definition of local variables>;
tel;
```

Figure 7.1: General form of a testnode syntax.

2. **Environment dynamics**: Specification of different temporal relations between the current inputs and past inputs/outputs. These relations often include, but are not limited to, the physical constrains of the environment (e.g. the temperature variation cannot be higher than 2°C between two successive instants; the temperature would raise if a heater is on).

   The above specifications are introduced in the testnode by means of the `environment` operator. Simple random test sequences can be generated, without a particular test objective, but considering all inputs allowed by the environment.

3. **Scenarios**: In order to test specific properties of the system, the tester can define more precise scenarios, by specifying additional invariant properties or conditional probabilities. The LUTESS `prob` operator provides a mean for specifying conditional probabilities which can be used either to force the test data generator to conform to realistic scenarios, either to simulate failures (e.g. when the heater is on, there is a probability of 90% for the temperature to raise).
   Recall that the values of the defined conditional probabilities cannot be statically checked (cf. Section 3.5.1, [61]).

4. **Property-based testing**: This step uses formally specified safety properties through the `safeprop` operator in order to guide the generation toward the violation of such a property [62]. The resulting generation will remove inputs that, given the property expression, obviously cannot lead to its violation. In order to be effective, this guidance requires the safety properties to be expressed as a relation between inputs that imply some outputs (e.g. when the temperature is high, then the heater must be on). Test hypotheses on the software behavior can also be introduced using the

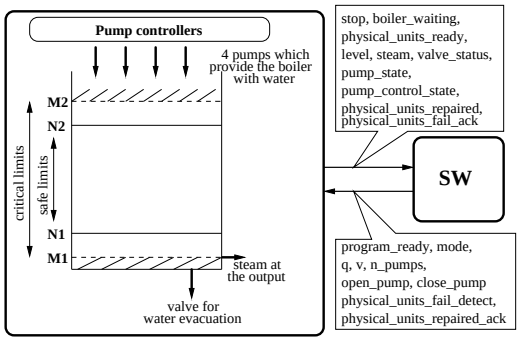


Figure 7.2: The steam boiler control system.

`hypothesis` operator and possibly make this guidance more effective (e.g. the program will turn the heater on only if the on/off button has been pushed).

7.2 The steam-boiler controller

The steam boiler controller system has been used more than ten years ago as a common case study for several formal specification methods [3]. An implementation of the system in LUSTRE has been proposed in [12]. We added the missing nodes in this implementation and we used a completed version of the LUSTRE code in this case study [48]. The steam boiler controller can be properly modeled by a real synchronous system as it represents a good example of a control/command system. Therefore, in point of both problem size and complexity, it is a rather suitable example to use with LUTESS in order to demonstrate the basic steps of its testing methodology, which is the main objective of this case study.

7.2.1 System specification

According to the informal specifications provided in [3], the physical system of the boiler is composed of four pumps which supply the boiler with water while it turns it into steam at its output. In order to avoid any malfunction or erroneous situation, the controller must maintain the level of water in the steam boiler, within some safe limits. More precisely, as it is shown in Figure 7.2, the physical units of the system are:

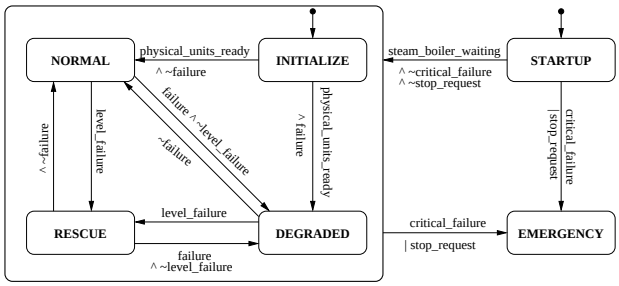


Figure 7.3: Operational modes of the steam-boiler system.

- **The boiler itself** comprises a valve, which at the initialization phase, evacuates the remaining water. It has a total capacity of C liters and produces a maximum steam quantity of W liters/sec. The minimum and maximum water limits are M_1 and M_2 respectively; outside these limits, the system will be endangered after five seconds, due to either lack of water supply or water overflow.

- **Four pumps** provide the boiler with water. Each pump is characterized by its capacity (p liters/sec) and its state, "on" or "off". Although a pump can be stopped instantly, when it is being started, it needs a whole cycle before it is being opened.

- **Four controllers** (one for each pump) inform if there is flow of water from the pumps to the boiler or not.

- **A water unit** measures the quantity of the water (q) in the boiler, measured in liters.

- **A steam unit** measures the quantity of the steam (v) at the output of the boiler, measured in liters/sec.

The physical system communicates with the program that controls its function, i.e. the controller, via messages. At each execution cycle, the controller receives and analyzes the messages from the physical system, then it sends back new commands.

According to the messages sent by the environment and the detected failures, the controller can operate in different modes. Figure 7.3 illustrates these modes.

1. *startup mode*: This is the very beginning of the program, where there is no critical failure neither a stop request detected and the program is waiting

for the appropriate message from the physical system that it is ready to begin functioning. If a critical failure is detected or there is a stop request, the program goes into the *emergency mode*.

2. *initialize mode*: As soon as the physical system is ready to start functioning, the program enters the initialization mode. In this mode, the program sends continuously to the environment a message denoting that it is ready to function; that happens until it receives back from the environment the corresponding positive response. Then, if there is no failure detected, the program enters the *normal mode*; otherwise, i.e. if there is a failure detected in a physical unit, the program enters the *degraded mode*.

3. *normal mode*: This is the mode where the program tries to maintain the quantity of water within the normal limits, N_1 and N_2, with all the physical units operating correctly, of course. Once a failure in the water unit is detected, the program enters the *rescue mode*, whereas in case of any other kind of failure, the program enters the *degraded mode*.

4. *degraded mode*: In this mode, the program tries to maintain the quantity of water in a satisfactory level, despite of a possible failure in a physical unit other than the water unit. When this failure is repaired, the program returns to the *normal mode*. When a failure in the water unit is detected, the program goes to the *rescue mode*.

5. *rescue mode*: In this mode, the program tries to maintain the quantity of water at a satisfactory level, despite of the failure in the water unit. In this case, the quantity of the water in the boiler is estimated, taking into account the quantity of steam at the output and the intake of water that the pumps supply. When the failure is repaired, the program goes back to *normal mode*, or to the *degraded mode* when a failure in another physical unit is detected.

6. *emergency mode*: This is the mode where the program must enter any time there is a critical failure or a stop request. Once the program reaches the *emergency mode*, it stops its execution and the physical environment is responsible of taking appropriate functions.

7.2.2 Modeling and testing the boiler

The primary function of the boiler controller is to keep the water level between the given limits, based on inputs received from different boiler devices. Thus, in order to test the controller in its normal functioning, we first consider that all devices behave correctly and provide the correct inputs to the controller. In a later stage, we consider different faults that are tolerated by the controller in order to test its reaction in these cases.

7.2.2.1 Domain definitions

The domain of an input variable is the set of meaningful values that an input is designed to receive. Integer inputs used in controllers are often used to represent a state, consisting of a limited subset of integers, like in the case of *valve_status* or *pump_state* variables. In other cases, such as *level*, they represent an interval of integers.

We use the following expressions to define the domain of each integer input in the controller[1] (inside parentheses are given the values of constants used):

- The water level value should be between 0 and the maximum capacity, C (1000), of the boiler in water:

  ```
  0 <= level and level <= C
  ```

- The steam value should be between 0 and the maximum quantity of steam, W (25):

  ```
  0 <= steam and steam <= W
  ```

- The valve can be closed (0) or open (1):

  ```
  closed <= valve_status and valve_status <= open
  ```

- The pump state can be closed or open:

  ```
  AND(N_pump, closed <= pump_state and pump_state <= open)
  ```

where `N_pump` is the number of pumps (4); `AND` is a boolean operator applying the above boolean expression to the whole array *pump_state* and resulting in the conjunction of the corresponding values.

[1] Note that these invariants are meaningful only under the assumption that all the devices are functioning.

So far, the above model can be directly used to generate random test sequences. Nevertheless, these sequences are not very conclusive. The controller cannot deal with the random behavior of all the devices, and as soon as it is started it goes to the *emergency mode*: either a stop request has been sent (*stop* being true for 3 consecutive steps), either a failure has occurred while initializing. Note that, according to the specification, any message received when not expected is considered as a failure. Thus, in order to observe meaningful executions, we should specify, more thoroughly, the environment dynamics.

7.2.2.2 Environment dynamics

Environment dynamics can be expressed as temporal relations between the current and past values of the software inputs and outputs. We can derive directly from the specification, the properties that identify the correct behavior of the devices, some of which are presented below:

- The *steam_ boiler_ waiting* message is sent only in *startup mode*:

  ```
  steam_boiler_waiting = (false -> (pre mode = startup))
  ```

- The *physical_ units_ ready* message is sent as a notification of the received message *program_ ready*:

  ```
  physical_units_ready= (false -> (pre program_ready))
  ```

- The water level is equal to its previous value, to which is added the quantity of water entering through the open pumps and removed the quantity exiting through the steam or the valve. Thus, the expected water level is:

  ```
  expected_level = level -> (pre(expected_level) + Dt*sum_flow(N_pump,
                   pump_control_state) - Dt*steam - Dt*valve_status*V)
  ```

where the `sum_flow` node calculates the sum of the flow passing through all the pumps, based on the controller state.

If the expected level is negative, this means that no more water remains in the steam boiler (*level=0*). If the expected level is greater then the total capacity, it means that not all the expected water has been flowing through the pumps and in this case the boiler is full (*level=C*). When the expected level is between 0 and *C*, it represents the actual quantity of water (*level=expected_ level*):

```
true -> if expected_level < 0 then level=0
        else if expected_level > C then level=C
             else level=expected_level
```

- When the boiler starts, the steam flow is supposed to be zero:

```
implies(true -> pre(mode=startup), steam=0)
```

- The state of the valve (*valve_ status*) changes only when the *valve* message is sent by the controller:

```
(valve_status=closed) -> (pre valve=(valve_status <> pre valve_status))
```

- The pump is opened (resp. closed) when it receives the *open_ pump* message (resp. *close_ pump*). This behavior is implemented in the *correct_ pump_ state* node (that we do not detail here for the sake of simplicity) and applied to all the pumps:

```
AND(N_pump, correct_pump_state(open_pump, close_pump, pump_state))
```

- The pump controller indicates that a pump is opened after a delay of one cycle (due to the time needed to balance the pressure as specified in [3]) and immediately when stopped:

```
AND(N_pump, pump_control_state = (pump_state = open^N_pump
                                  and pre(pump_state = open^N_pump)))
```

- When a fault message is received from the controller, the device acknowledges the controller that it has received the message. This behavior is applied to all the pumps and their controllers, the level and the steam. We give here the example of the level unit:

```
level_failure_acknowledgment = (false -> pre level_failure_detection)
```

- When a fault detection message is received, the devices send the repaired message until they receive a message of acknowledgment from the controller. This behavior is applied to all the pumps and their controllers, the level and the steam. We show below how to apply this to all the pumps:

```
AND(N_pump, pump_repaired = (false^N_pump -> pre(pump_repaired)
                             and not(pre pump_repaired_acknowledgment)
                             or pre pump_failure_acknowledgment))
```

Table 7.1 shows a generated test case resulting from the above test model. In order to avoid premature stop of the system, we added the `not(stop)` invariant.

	t_0	t_1	t_2	t_3	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{998}	t_{999}	t_{1000}
stop	0	0	0	0	0	0	0	0	0	0	0	0	0
steam_boiler_waiting	0	1	0	0	0	0	0	0	0	0	0	0	0
physical_units_ready	0	0	0	0	0	0	1	0	0	0	0	0	0
level	964	964	859	804	519	434	389	374	329	459	534	544	509
steam	0	0	11	1	13	17	9	3	24	4	22	13	22
valve_status	closed	closed	open	open	open	closed	closed	closed	closed	closed	closed	closed	closed
pump_state[0..3]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[1,0,0,0]	[1,1,0,0]	[1,1,1,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]
pump_control_state[0..3]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[1,0,0,0]	[1,1,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]
program_ready	0	1	0	0	0	0	0	0	0	0	0	0	0
mode	start	init	init	init	init	init	normal	normal	normal	normal	normal	normal	normal
valve	0	1	0	0	1	0	0	0	0	0	0	0	0
q	964	964	859	804	519	434	389	374	329	459	534	544	509
v	0	0	11	1	13	17	9	3	24	4	22	13	22
open_pump[0..3]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[1,0,0,0]	[0,1,0,0]	[0,0,1,0]	[1,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]
close_pump[0..3]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[1,0,0,0]	[1,0,0,0]	[1,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]

Table 7.1: Excerpt of a test case using all the possible invariant properties.

The generated test sequences simulate the normal function of the steam boiler. At the first execution cycle, the controller requests the opening of the valve, because of the high water level, until the latter is reduced to the safe limits (t_7). Afterward, the controller continues on *normal mode*, since no failure is signaled by the physical units.

7.2.2.3 Test scenarios

Previously generated test cases are obtained by adding systematically all the invariants that define correct functioning of the boiler devices. This can be seen as a normal execution of the software. But, often when testing, one can try to put the software into abnormal execution scenarios. In LUTESS, specific execution scenarios can be obtained either with invariant properties or by specifying different conditional probabilities.

In Section 7.2.2.2, using an invariant property, we have set *stop* to be always false, which can be seen as a simple scenario: "the boiler is never stopped". If we want to test whether the software behaves correctly in case of a shutdown, we may want to allow "sometimes" the *stop* message to be true, by specifying a small probability: `prob(true, stop, 0.05)`. Recall that the expression `prob(C,E,P)` means that if the condition `C` holds then the probability of the expression `E` to be true is equal to `P`. Hence, with the above probability, the obtained sequences have rare occurrences of the *stop* message and the system can be observed for some time, before being stopped late in the testing process.

Failures of the system can also be simulated this way. To do so, invariant

properties expressed in Sections 7.2.2.1 and 7.2.2.2 can be replaced by `prob` expressions. For instance:

- A possible failure of the system can occur in the *initialize mode*, if we assume that there is a small probability to get the message *physical_ units_ ready*, when expected:

```
prob(true, physical_units_ready = (false -> (pre program_ready)), 0.2)
```

- Another failure may consist of a change of the valve state, even if no command has been sent to the valve:

```
prob(true, (valve_status=closed) ->
           (pre valve = (valve_status <> pre valve_status)), 0.8)
```

Of course, this list is not exhaustive. Every property in an `environment` operator could be replaced by such a `prob` expression. The value of the assigned probability must be empirically determined by the tester.

We show here, a more advanced simulation, of a non signaled failure of the level measurement unit. To do so, we first remove any domain constraints for the level input. Then, we consider the previously shown invariant specifying the current level value:

```
level_inv = true -> if expected_level < 0 then level=0
                    else if expected_level > C then level=C
                         else level = expected_level
```

We keep the same invariant conditions when not in *normal mode*:

```
implies(true -> pre(mode)<>normal, (0<=level and level<=C) -> level_inv)
```

And, while in *normal mode*, we introduce:

```
prob(false -> pre(mode)=normal, level_inv, 0.9)
```

Table 7.2 shows a generated test case for this level device fault simulation and the corresponding controller reaction. At instant t_{13}, an arbitrary negative level value has been generated and the controller has detected a fault in the water level measuring device (`level_failure_detection`=1). We can notice that the mode has changed to *rescue* and the level value has been estimated. At the next two steps, the device has been repaired and the controller goes back to *normal mode*.

	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}
physical_units_ready	0	1	0	0	0	0	0	0	0	0	0
level	579	519	509	464	429	334	-1069082252	389	514	574	614
level_repaired	0	0	0	0	0	0	0	0	1	0	0
level_failure_acknowledgment	0	0	0	0	0	0	0	1	0	0	0
mode	init	normal	normal	normal	normal	normal	rescue	rescue	normal	normal	normal
q	579	519	509	464	429	334	279	389	514	574	614
level_failure_detection	0	0	0	0	0	0	1	0	0	0	0
level_repaired_acknowledgment	0	0	0	0	0	0	0	0	1	0	0

Table 7.2: Excerpt of a test case with broken level device scenario.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
stop	1	0	1	0	1	1	0	1	1	1	0	0
physical_units_ready	0	0	0	0	0	0	1	0	0	0	0	0
mode	start	init	init	init	init	init	normal	normal	normal	emergency	emergency	emergency

Table 7.3: Excerpt of a test case guided by a safety property.

7.2.2.4 Property-based testing

Safety properties are specified in a testnode using the `safeprop` operator. When this operator is used, testing is performed to check if these properties are satisfied by the program under test. Property-based testing guides the test generation by avoiding, when possible, input values that cannot lead to a property violation. Consider, for instance, the property: "issuing the *stop* message for 3 consequent steps leads the controller into the *emergency mode*". To formally express this property we use two local boolean variables `pre_stop` and `pre_pre_stop`, referring to the value of `stop` for the past 2 steps:

```
pre_stop = false -> pre stop
pre_pre_stop = false -> pre pre_stop
safeprop(implies(false -> pre(mode)=normal and stop and pre_stop
                 and pre_pre_stop, mode=emergency))
```

Violating this property requires setting the left part of the implication to *true*. Table 7.3 shows the effect of the above specification on the generated sequence. Immediately after the controller has passed into *normal mode*, the value of `stop` is set to *true* for the following 3 steps.

7.2.3 Remarks on the problem size

The steam boiler problem requires exchanging an important number of messages between the system controller and the physical system. The main program

handles 38 inputs and 34 outputs, boolean or integer, and it is composed of 30 internal functions. The main node is made, when unfolded, of 686 lines of LUSTRE code. Each testnode consists of about 20 invariant properties modeling the boiler environment to which are added various conditional probabilities or safety properties. The average size of a testnode, together with the auxiliary nodes, approximates 200 lines of LUSTRE code. It takes approximately less than 30 seconds to generate a sequence of hundred steps, for any of the test models we used (tests performed on a Linux Fedora 9, Intel Pentium 2GHz and 1GB of memory; LUTESS uses the ECLiPSe[2] environment for constraint solving.

7.3 The land gear controller

In this section, we apply the LUTESS testing approach to a specification of a real critical embedded software: a landing gear control system of a military aircraft. This system had been used in [8] to assess some formal verification tools. We describe in detail the testing modeling procedure with the intention of assessing the difficulties in the activities of test modeling and test generation in case of a real world application. Moreover, the objective of this case study is to check the applicability and the scalability of our approach, as far as the latter can be assimilated to the length and the complexity of the specification, and the resources needed to the test generation. The particularity in this case study is that the system handles only boolean variables, so controlling the input variables and the environment dynamics is rather limited. Therefore, we examine more closely the use of conditional probabilities and the possible violation of the safety properties. In addition, we show that having in mind a specific test objective and guiding the test generation process by a specific technique, it is possible to detect faults in the program.

7.3.1 System specification

The landing gear control system of a military aircraft [8] is in charge of maneuvering landing gears and associated doors. The system is controlled by the command software in normal mode or analogically in emergency mode. It is composed of 3 landing gears: front, left and right. Each gear has an up lock box and a door with two latching boxes. Gears and doors are controlled by an hydraulic system. Figure 7.4 shows a general description of this system which

[2] http://www.eclipse-clp.org

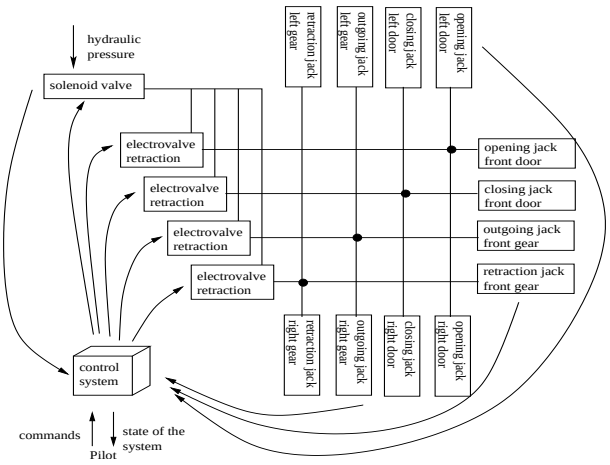


Figure 7.4: The landing gear system.

consists of :

- a set of actuators:
  - a valve to isolate the emergency system;
  - electrovalves to open or close the doors and let down or retract gears;
- a set of sensors giving the state of each component of the system.

To order the retraction and outgoing of gears, the pilot has a set of 2-position buttons and a set of lights giving the current positions of doors and gears. When the command line is working, the controller obeys the orders of the pilot, within the control software, executing a sequence of actions, directly animating the mechanics. The outgoing of gears is decomposed as follows:

1. stimulation of the solenoid isolating the command unit;
2. stimulation of the door opening solenoid;
3. once the doors are opened, stimulation of the gear outgoing solenoid;
4. once the gears are locked down, stop the stimulation of the gear outgoing solenoid and start the stimulation of the door closure solenoid;
5. once the doors are closed, stop the stimulation of the door closure solenoid;
6. finally, stop the isolating electrovalve.

The retraction sequence is symmetric. These two sequences lead from the "cruise state" (i.e. gears up and doors closed) to the "landing state" (i.e. gears down and doors closed) and conversely.

Because of hydraulic constraints, a given timing must exist between stimulation and stop of the valves. Moreover, the sequences should be interruptible if a counter order is given by the pilot, for example a retraction order during the let down sequence.

7.3.2 The control software

The control software handles the landing gear physical system in normal mode (no failure), receiving data from the sensors or pilot orders, and producing commands for the physical system. It is decomposed in 3 main functions:

- a monitoring function for gears and doors signaling inconsistencies;
- a monitoring function checking that the system reacts correctly and controlling the functional and temporal coherence of the commands;
- a command function implementing the sequence of outgoing and retraction of gears. This function gives direct orders to the physical system and is decomposed in two main parts:
  - a function computing stimulation command for each component;
  - functions managing the emission of the command (timing constraints).

All input and output variables of the control system are boolean. Through this experiment we will consider a single gear, as the three gears are identical.

The input variables are:

- `Lever_Up`: true if the lever is in up position for retracting the gears, false if it is in down position for outgoing the gears.
- `Gears_Off`: true if the shock absorbers are relaxed (the aircraft is flying).
- 7 variables indicating the initial state of the gears and the doors, for instance:
  - `Init_Closed_Door_Up_Gear`: true if the door is closed and the associated gear is up;

  - `Init_Maneuvering_Door_Up_Gear`: true if the door is moving and the associated gear is up;
  - `Init_Open_Door_Up_Gear`: true if the door is opened and the associated gear is up.

- 12 variables providing the state of the gear, the door, the oil pressure and the solenoid valve in case of blocking (failure), for instance:
  - `Gear_Up_Fail`: true if the gear is blocked in the up position;
  - `Gear_Down_Fail`: true if the gear is blocked in the down position.

The output variables are:

- 5 variables giving commands to start and stop stimulation of the electrovalves and solenoid valve:
  - `Solenoid_Valve_Stimulation`: true if the solenoid valve is stimulated for allowing the fluid to pass;
  - `Gear_Outgoing_Stimulation`: true if the electrovalve of outgoing the gears is stimulated for allowing the fluid to pass;
  - `Gear_Retraction_Stimulation`: true if the electrovalve of retracting the gears is stimulated for allowing the fluid to pass;
  - `Door_Opening_Stimulation`: true if the electrovalve of opening the doors is stimulated for allowing the fluid to pass;
  - `Door_Closing_Stimulation`: true if the electrovalve of closing the doors is stimulated for allowing the fluid to pass.

- 6 variables indicating to the pilot the state of the gear, the door and the oil pressure after the solenoid valve. The controller calculates these states by considering the pilot actions (the lever movement), the necessary time for the outgoing and the retraction of gears, the opening and the closing of the doors as well as the solenoid valve state in case there is no failure:
  - `Gear_Up`: true if the gear is in up position;
  - `Gear_Down`: true if the gear is in down position;
  - `Door_Closed`: true if the door is closed;
  - `Door_Open`: true if the door is opened;

  - `Under_Pressure_Manometer`: true if the pressure is present in the hydraulic system after the solenoid valve;
  - `Analogic_Link_On`: true when the command coming from the lever is allowed to be transmitted.

- other variables aiming at warning the pilot in case of malfunction or non response of mechanical components.

7.3.3 Modeling the land gear controller

7.3.3.1 Domain definitions

Firstly, we assume that we test the controller in its normal functioning and that all components operate correctly feeding the controller with the expected values. In this way, we can define the following properties:

- There is no failure in the components. For instance, the corresponding formal expression of this property in case of a failure in the gears[3] is:

  ```
  not Gear_Up_Fail and not Gear_Down_Fail
  ```

  Of course, this specification may be modified or removed if failures must be taken into consideration.

- At the first step (t_0), there is only one active initial state variable for the doors and the gears and it is active only at this step:

  ```
  (one_active(Init_Closed_Door_Up_Gear, ...) -> true)
  and
  true -> neg_conj(Init_Closed_Door_Up_Gear, ...)
  ```

  where `one_active` and `neg_conj` are two user-defined boolean operators, especially built so that the code is easily maintainable and readable. The first one handles the 7 variables that indicate the initial state of the gears and the doors (presented in Section 7.3.2) and allows exactly one of them to be true at the initial step. Similarly, `neg_conj` operates on the same variables prevent them from being true.

The above, very simple, test model, can be directly used to generate test input sequences. Test inputs will be randomly generated at every instant and

[3]The expressions for the other components failures are similar and they are not presented here for the sake of simplicity.

their values will be chosen in the above defined domains. Nevertheless, these sequences may not be very conclusive, as it is usually the case with random testing. At the next step of the test methodology, to cope with this issue, we specify more thoroughly the environment dynamics, in order to observe more meaningful executions.

7.3.3.2 Environment dynamics

As it is mentioned above, in this case study, there are very few properties that determine the temporal relations between the current and past values of the system inputs and outputs, because of the nature of the variables. Indeed, most of them are directly expressing human actions and only few variables (mainly related to failures) are subject to physical constraints. However, in this test model this feature can be used to express the absence of failures, for instance by means of the following property:

- When the aircraft reaches the ground, the gears are locked down and the doors are closed:

  ```
  implies(not Gears_Off, Init_Closed_Door_Down_Gear)->
  implies(falling_edge(Gears_Off), pre Gear_Down and pre Door_Closed)
  ```

  The user-defined node `falling_edge` returns a true value when its input was previously *true* (at instant $t_{\text{n-1}}$) but currently (at instant t_{n}) it is *false*.

Table 7.4 shows a generated test case resulting from the above test model using invariant properties that define the variable domains and the environment dynamics. The domain constraints are indeed verified, but the test sequences are not absolutely convincing. The command lever moves quite often and prevents many states from being generated (mainly the complete deployment and retraction of the gears). These states need for the lever to be stable for some time. We can also notice that the aircraft lands (`Gears_Off=false`) whenever the gears and the doors are in the correct position (`pre Gear_Down=true` and `pre Door_Closed=true`).

7.3.3.3 Test scenarios

The previous test cases are generated by specifying invariant properties of the program environment. However, this trace of test cases does not represent a realistic system behavior. For instance, the lever must be kept in the same

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
Lever_Up	1	0	1	0	0	0	0	1	0	1	1	1
Gears_Off	0	0	0	0	1	0	1	1	1	1	1	0
Init_Closed_Door_Down_Gear	1	0	0	0	0	0	0	0	0	0	0	0
Gear_Up_Fail	0	0	0	0	0	0	0	0	0	0	0	0
Gear_Down_Fail	0	0	0	0	0	0	0	0	0	0	0	0
Gear_Up	0	0	0	0	0	0	0	0	0	0	0	0
Gear_Down	1	1	1	1	1	1	1	1	1	1	1	1
Door_Closed	1	1	1	1	1	1	1	1	1	1	1	1

Table 7.4: Excerpt of a test case using only invariant properties.

position for a given time period in order to change the state to "cruise state" or "landing state" (c.f. Section 7.3.1). So, we need to guide more thoroughly the `Lever_Up` variable. To do so, we could use the following invariant property scenario (using the `environment` operator):

- When the pilot puts the lever in down position for deploying the gears, he maintains it in this position until reaching the "landing state" (gears down and doors closed), and conversely:

```
implies(pre not Lever_Up and (pre not Gear_Down or pre not Door_Closed),
        not Lever_Up)
and
implies(pre Lever_Up and (pre not Gear_Up or pre not Door_Closed),
        Lever_Up)
```

The same scenario may be expressed differently by specifying a certain conditional probability and taking into consideration the time required for a full gear deployment or retraction. In fact, all functions included in the control system are implemented by periodic and sequential processes executed every 40ms. More specifically, according to the system implementation provided in [8], the time needed to fully deploy the gear from the "cruise state" or to fully retract it from the "landing state" corresponds to 36 execution cycles at most[4]. As a result, the property could be expressed by means of the `prob` operator as follows:

- When the lever is in down/up position, it is highly probable that it will remain in this position. Let us assume that the probability is fixed at

[4]Actually, the initial experiment was intended to verify this property for a 14-seconds time period. However, since explosion of the number of states occurs partly due to timings issued in the system and in the property, timing was divided by 10 (both in the system and the property). Thus, the required time was considered to be equal to 1.4 seconds, which corresponds to approximately 36 execution cycles.

	t_{50}	t_{51}	t_{52}	t_{53}	t_{54}	t_{60}	t_{61}	t_{62}	t_{63}	t_{64}	t_{65}	t_{72}	t_{73}	t_{74}	t_{75}	t_{76}	t_{77}	t_{78}
Lever_Up	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
Gears_Off	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Gear_Up	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Gear_Down	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Door_Closed	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1
Door_Open	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0

Table 7.5: Excerpt of a test case using the scenarios specified in Section 7.3.3.3.

97%. So, since 36 steps are needed to fully deploy/retract the gear, the probability to get a sequence of 36 successive steps with the same lever position is approximately $1/3(0.97^{36} \approx 0.33)$.

```
prob(true -> pre Lever_Up, (Gears_Off and Lever_Up) -> Lever_Up, 0.97)
prob(false -> not(pre Lever_Up),
     ((not Lever_Up) and (not Gears_Off)) -> not Lever_Up, 0.97)
```

Clearly, every property in an `environment` operator could be replaced by a similar `prob` expression. The value of the assigned probability must be empirically determined by the tester. Such scenarios describe non-deterministic behaviors.

In another scenario, we could consider the plane in its landing position:

- When the gears have been locked down, the pilot should most probably land the plane:

```
prob(false -> pre Gear_Down and pre Door_Closed and pre Gears_Off,
     true -> not Gears_Off, 0.9)
```

- When the aircraft is on the ground, it may stay there for some time:

```
prob(false -> pre not Gears_Off, true -> not Gears_Off, 0.6)
```

Table 7.5 shows a generated test case for the scenarios specified previously. We notice that the lever rarely changes its position, allowing to observe a full retraction of the gear. In addition, the aircraft remains on the ground for a few cycles (t_{51} to t_{53}). Most importantly, the changes in the gears position and the door state become more clear. For example, at instant t_{60}, the gear is down and until the moment it goes up (t_{65}), it is neither down nor up. Similarly, the door moves (it is neither open nor closed) before it is closed (t_{72} to t_{74}). Eventually, after a few cycles, the aircraft goes into the "cruise state" (gear up and door closed) due to the long duration that the lever was held in the up position.

	t_{103}	t_{104}	t_{105}	t_{106}	t_{195}	t_{196}	t_{197}	t_{198}	t_{199}	t_{200}	t_{380}	t_{381}	t_{382}	t_{383}	t_{384}	t_{385}
Gear_Up_Fail	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
Gear_Up	1	1	1	0	1	1	1	0	0	0	1	1	1	1	1	0
Gear_Outgoing_Stimulation	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1

Table 7.6: Excerpt of a test case using scenario concerning the gear up failure specified in Section7.3.3.3.

Test scenarios can also be used to simulate system failures. In the present case study, failures are mainly caused by blocked physical components. Such erroneous situations can only take place when a stimulation of a component is issued. For example, let us consider a system failure described by the following scenario: "when the gear is up and a stimulation for its outgoing starts, the sensor can detect the failure in this position of the gear". In order to simulate this scenario using LUTESS, we should replace the environment constraint that prevents the failure of the gear in its up position (`not Gear_Up_Fail`) seen in Section 7.3.3.1 by the following property:

- If there is a failure in the gear when this is in the up position, then the gear was up and there was an outgoing stimulation at the previous cycle :

  ```
  not Gear_Up_Fail->implies(Gear_Up_Fail, pre Gear_Up and pre Gear_Outgoing_Simulation)
  ```

  The absence of failure at the first execution cycle is important in order to reassure that the system starts up normally.

We can also specify the probability for the occurrence of this failure when the right part of the previous implication is verified. To do so, a possible scenario could be the following:

- If the gear is up and there is a stimulation for its outgoing, then there is a small probability that the gear will be blocked in the up position:

  ```
  prob(false -> pre Gear_Up and pre Gear_Outgoing_Stimulation, Gear_Up_Fail, 0.2)
  ```

Table 7.6 shows a generated test case for this scenario simulating a failure of the gear and the corresponding controller reaction. We observe that the failure occurs only when the gear was up and the gear outgoing stimulation was on at the previous instant. Since the value of the specified probability is rather low, the failure rarely occurs, even if the precondition is verified (e.g. at instants t_{104}, t_{105} or t_{196}, t_{197}).

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Gears_Off	0	0	0	0	0	0	0	0	0	0	1
Analogic_Link_On	1	1	1	1	1	1	1	1	1	1	1
Gear_Retraction_Simulation	0	0	0	0	0	0	0	0	0	0	0

Table 7.7: Excerpt of a test case guided by a safety property.

7.3.3.4 Property-based testing

At this step, we first show a simple case of property violation, considering the following property: "there is no retraction of the gears when the aircraft is on the ground, even if the pilot acts on the lever to achieve that". This property is formally expressed as follows:

```
safeprop(implies(not Gears_Off,
         not Gear_Retraction_Stimulation or not Analogic_Link_On))
```
[5]

Violating this property requires setting the left part of the implication to true. Table 7.7 shows the effect of the above specification on the generated test cases. `Gears_Off` is set to *false*, which as specified, is a necessary condition to violate the property; therefore, this trace helps to observe if the property is verified by the program.

Let us consider a more sophisticated situation, in which the property to be tested is: "if the door is closed or maneuvering, then the gears do not move (remain in the up or in down position)". The formal expression of this property is:

```
safeprop(implies(Door_Closed or (not Door_Closed and not Door_Open),
                 Gear_Up or Gear_Down))
```

For this property, setting the left part of the implication to true is more complicated, since it contains exclusively some of the program outputs, which cannot be directly handled. The tester can only lead the system in such a state, so that the system gives the desirable outcome at the output. To this end, we use the keyword `hypothesis`, which makes it possible to introduce into the test generation process some knowledge concerning the behavior of the program under test. Such hypotheses can provide information, even incomplete, on the way that outputs are computed and hence facilitate the computation of inputs during safety property guided testing.

[5]The physical action of moving the gear is allowed by the combination of the stimulation and the open state of analogic link.

model with hypotheses	t_{103}	t_{104}	t_{105}	t_{106}	t_{107}	t_{108}	t_{411}	t_{412}	t_{413}	t_{414}	t_{415}	t_{416}	t_{417}
Door_Closed	1	1	1	0	0	0	0	0	0	1	1	1	1
Door_Open	0	0	0	0	0	0	0	0	0	0	0	0	0
model without hypotheses													
Door_Closed	0	0	0	0	0	0	0	0	0	0	0	0	0
Door_Open	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 7.8: Excerpt of a test case guided by a safety property and certain hypotheses on the system.

Some of the hypotheses we could use in order to close the door or maintaining it in its position are:

- The software ensures that if the lever is maintained in the same position for the required period (36 cycles) so that the outgoing/retraction operation is finished, the door will be closed:

```
hypothesis(implies(maintained(36, not Lever_Up) or maintained(36, Lever_Up),
          Door_Closed))
```

- If the lever is maintained in the down position for the same duration, for the same reason, the gear will be down and vice-versa:

```
hypothesis(implies(maintained(36, not Lever_Up), Gear_Down))
hypothesis(implies(maintained(36, Lever_Up), Gear_Up))
```

Doubtlessly, program analysis, usually thorough enough, results in this kind of hypotheses. Alternatively, hypotheses might concern some program properties that are considered as satisfied, because they have been successfully tested before. In any case, this step requires the tester to be well trained and experient as well as to be familiar with the system under test.

Table 7.8 illustrates the generated test cases from the above specification, where the test generation is guided by a safety property combined with hypotheses on the system. In a sample of 1000 test sequences, the generator managed to rise the number of cases in which the left part of the implication in the safety property is true. Indeed, the door remains closed (`Door_Closed=false`) or the door is maneuvering (`Door_Closed=false` and `Door_Open=false`).

7.3.4 Evaluation aspects

So far, the test oracle (Ω, in Figure 3.7 of Section 3.5.1) that describes the system requirements has not been taken into account during testing analysis.

	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
Gears_Off	0	0	0	1	1	1
Analogic_Link_On	1	1	1	1	1	1
Gear_Retraction_Simulation	1	1	1	0	0	0
Gear_Up	0	0	0	0	0	0
Gear_Down	1	1	1	0	0	0
Door_Closed	1	1	0	0	0	0
Door_Open	0	0	0	0	0	0
Oracle	0	0	0	0	0	0

Table 7.9: Excerpt of a test case with oracle violation.

Towards our attempt to evaluate the LUTESS testing methodology in terms of its fault detection ability, we introduced certain errors in the program code and, by observing the oracle verdict, we were able to verify if the intentional errors were detected. More precisely, we considered some operator changes in the original program, that is, replacements of some operators by others so that the obtained program is syntactically correct. First, we modified the module that processes the commands concerning the gears and the doors; in the equation that determines the gears retraction command (`Gear_Retraction_Stimulation`) in function of the gears position, the doors state and the shock absorbers state, the `and` operators were replaced by `or` operators. Indeed, in the original program, in order to stimulate gears retraction, the gears must not be up, the doors must be open and the shock absorbers must be relaxed (`Gears_Off`); this last condition indicates that the plane is flying. On the contrary, due to the introduced error, at least one of the above conditions is sufficient for the gear retraction stimulation. As for the environment description, we used a test model which combines the scenarios described in Section 7.3.3.3, except the one of failure, and the first safety property presented in Section 7.3.3.4. Lastly, in the oracle, we included both the safety properties explained in Section 7.3.3.4.

It is worth mentioning that at 94 out of 100 steps, the oracle was violated, hence guiding the test data generation by safety properties results in test sets adequate to validate these properties. An excerpt of the results obtained by the above simulation and the oracle verdict is demonstrated in Table 7.9. At steps t_7 to t_9, the first property is violated, whereas at steps t_{10} to t_{12} the second property is violated.

In addition, we attempted to detect a system failure, taking into account the failure simulation concerning the gear position, that was described in Sec-

	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	t_{23}
Gear_Up_Fail	0	1	0	0	0	0	0	0	0	1	1	0	0
Gear_Up	1	1	1	1	1	1	1	1	1	1	0	0	0
Gear_Outgoing_Stimulation	1	1	1	1	1	1	1	1	1	1	1	1	1
Oracle	1	1	1	1	1	1	1	1	1	1	0	1	1

Table 7.10: Excerpt of a test case with oracle violation simulating a failure in the up gear.

tion 7.3.3.3. In this case, we used the environment description with the failure simulation and in the oracle, we added the following property:

- If a gear is blocked in the up position (`Gear_Up_Fail`), then the gear must be up (`Gear_Up`), since this variable value is sent back to the pilot:

```
implies(Gear_Up_Fail, Gear_Up)
```

In the program under test, we introduced a fault in the module that handles the system components. In particular, in the equation that counts the required time (execution cycles) for moving the up gear, we removed the `not` operators, since the beginning and the interruption of this counter depends on the gear position, i.e. if it is blocked or not in the up position. The oracle violation resulting from this configuration is shown in Table 7.10. At step t_{21}, the property concerning the failure is not verified.

Furthermore, an important remark is that in case we do not use property-based testing (i.e. guide the test generation only by invariant properties and test scenarios), the violation of the property is not systematic and depends, in particular, on the seed used to initialize the random generator of LUTESS. On the contrary, when test generation has been guided by the safety property, the latter has been always violated, regardless of the seed.

7.3.5 Remarks on the problem size

The main program of the land gear control system is composed of 32 internal functions and handles 21 input and 29 output boolean variables. Each testnode consists of about 20 properties modeling the system environment to which are added several conditional probabilities, safety properties and hypotheses on the system. Test were performed on a P4 2.8 GHz single core with 1 GB of memory under Linux Fedora 9. It takes approximately less than 30 seconds to generate a sequence of hundred steps, for any of the test models we used. The time seems to linearly increase with regard to the number of properties used in the test model

(as they are relatively close in complexity). Also, for the same environment model, time increases linearly to the number of steps.

7.4 Concluding remarks

In this section, we have illustrated the guidelines for the test model construction using LUTESS by modeling in different ways the external environment of two systems of different size and complexity; on the one hand, the steam boiler control system and on the other, the land gear control system of a military aircraft. With regard to a test objective or a specific system behavior that needs to be tested, the tester can model the system environment with reference to its specification in order to guide accordingly the test data generation.

The objective of the steam boiler case study was to provide the guidelines and recommendations for the test model construction as well as to assess the test modeling difficulty and the test generation complexity of LUTESS. The overall study showed that relevant test models for the steam boiler controller were not difficult to build; modeling the steam boiler environment required a few days of work. This first experiment suggests that the methodology and the tool could be suitable for real-world industrial applications.

Furthermore, the objective of the land gear controller case study was to evaluate the applicability and the scalability of the LUTESS testing approach in a real system from the avionics. We concentrated mostly on the last steps of the testing procedure, aiming at demonstrating how the tester can achieve valid and more thorough test sequences with regard to the system functional requirements. In addition, we deliberately introduced specific errors in the program under test in order to evaluate the fault-detection capability of the tool. The results indicate that guiding the test data generation by safety properties is the most powerful testing technique in detecting errors. Of course, this is also the most demanding technique in terms of tester's experience and familiarity with both the tool and the system under test.

In general, it should be noted that although the test modeling activity might be performed in a short period of time, the effort required to build the test models for a complete test operation is rather difficult to assess accurately. Actually, this depends on the desired thoroughness of the test sequences which may lead the tester to write several scenarios corresponding to different situations and resulting in different test models. However, building a new testnode in order to

generate a new set of test cases usually requires a slight modification of a previous testnode (adding/deleting some properties). This makes easy to generate a big number of test sequences with a small effort. Thus, when compared to manual test data construction, that the test professionals often use in practice, such an automatic generation of test cases could certainly facilitate the testing process.

Chapter 8

Conclusions and future work

8.1 Conclusion

This dissertation deals with the current aspects and open issues concerning the development of formal methods and tools for the validation and verification of synchronous LUSTRE/SCADE applications. According to the actual industrial as well as academic needs in critical avionics software testing, which is the most demanding and rigorous phase of the development cycle, we attempt to improve the existing testing practices.

The research in this work lies in two main axes.

- With regard to the structural coverage evaluation, we proposed the extension and the enhancement of the structural coverage metrics defined in the past [33]. The proposed coverage approach provides criteria defined directly on the LUSTRE/SCADE model as a means to evaluate test data quality and determine when testing can stop.

  On the one hand, these criteria were extended towards the use of multiple clocks in order to enlarge the scope of their applicability by completing the LUSTRE operator set. More precisely, in the coverage model of the operator network, we have defined the notion of coverage for the temporal operators `when` and `current`, which handle multiple clocks in a LUSTRE program [47]. These criteria have been integrated in SCADE environment by properly assimilating the instrumentation phase in MTC module.

  On the other hand, we studied the extension of the criteria to complex LUSTRE specifications that use numerous nested nodes. We proposed the

integration of the nested nodes in order to avoid their complete expansion and eventually reduce the complexity of coverage measurement. An *abstract* notion of coverage has been introduced which results in a simplified version of the obtained coverage node. Therefore, computing the coverage of a large-scale program that contains several internal modules (LUSTRE nodes) consists in considering these internal nodes as black boxes and applying the *abstracted* criteria directly to them. As a result, the size and complexity of the obtained coverage nodes is reduced, particularly for the multiple conditions criterion, in which case the number of the obtained paths and activation conditions was prohibitive to coverage measurement even for medium-sized applications.

- With regard to test data generation, we studied the modeling of the system environment as a means to generate test cases complied with the system informal requirements. We have contributed to the definition of testing methodology [48] based on LUTESS, a testing tool for automatic test data generation for synchronous reactive software. We provided the guidelines and helpful recommendations for the environment model construction according to the system functional specifications. Taking advantage of the different testing strategies provided by the tool, we went over the complexity of the corresponding test models by assessing the difficulty to build them in relation to the thoroughness and the size of the system specifications.

In order to support the above contributions with empirical results, we conducted experiments on two case studies derived from the avionics domain. For the coverage evaluation part, we used a module of an alarm management system in the flight control software of an aircraft to apply our proposed coverage approach. The objectives of this case study were to demonstrate the criteria applicability to realistic examples, to measure the required testing effort in order to meet the criteria as well as to assess their fault detection ability. We used mutation testing to simulate fault models of the program. Furthermore, for the test generation part, we tested over LUTESS the land gear controller system of a military aircraft. The purpose of this case study was to illustrate the usefulness of LUTESS testing approach for real-world applications and to evaluate the scalability and the efficiency of the test models according to the pertinence and validity of the obtained test cases. We were especially concerned with the property-based testing technique as a means to guide the test data generation process towards

the violation of specific properties of the system. In order to provide evidence of the power of this technique, we intentionally introduced certain errors in the program under test and we verified its fault detection capability.

Even if more case studies are required to assess the applicability and scalability of the proposed approaches, the conducted experiments suggest that they could be effective and applicable to the validation and verification of real industrial applications.

8.2 Perspectives

The proposed approaches in this dissertation contribute to the improvement of testing techniques for synchronous embedded systems but there are still several topics that require further investigation.

First of all, the defined coverage criteria are limited to LUSTRE specifications that handle exclusively boolean variables. Their definition implies that the path activation is examined in relation to the possible values that path inputs can take on, that is *true* and *false*. This means that, in case of integer inputs, the criteria would be inapplicable. Therefore, the criteria extension to more variable types appears to be a significant task, since real-world applications deal with variables of different types. A possible extension might consider examining more thoroughly the system specifications in order to narrow down the input variables domain to a subset containing only those values that are used in reality.

Moreover, the results obtained by the experimental evaluation showed that in many cases, one test suite satisfies several activation conditions. It would be interesting to investigate more thoroughly the coverage ratio that the criteria yield with regard to the number of the required test cases and hence explore ways of finding a minimal test cases set that satisfies all the activation conditions for a given criterion. A suitable technique for reducing the size of test case sets is proposed in [43]; this technique relies on the application of heuristics and uses mutation testing to try to find the minimal test set that meets a testing criterion.

Another extension of the defined coverage criteria concerns the new version of SCADE V6. This version supports parallel use of data-flow diagrams and control-flow graphs. As Figure 8.1 illustrates, a system can be represented by a control-flow graph; inside the control states, data-flow diagrams can be defined. In order that the coverage criteria can be applicable to this new model, it is

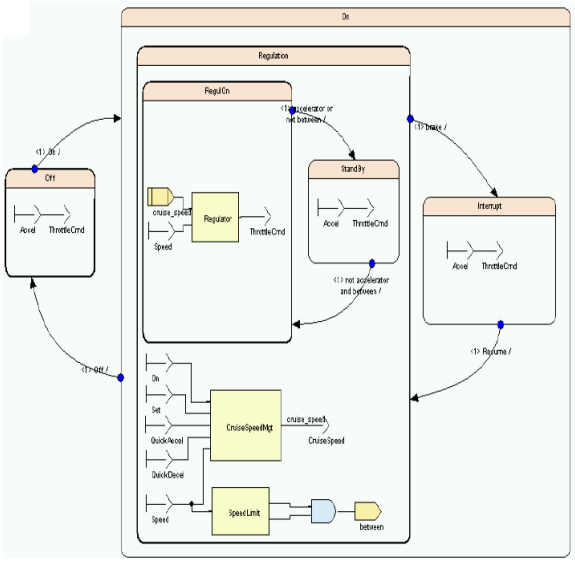


Figure 8.1: New feature in SCADE V6.

necessary to redefine the notion of the path activation condition in the context of the control-flow graph. Besides, the coverage of states and transitions should be one of the objectives of the coverage analysis.

Conclusion et perspectives

Bilan de la thèse

Le travail que nous venons de présenter s'inscrit dans le cadre de développement de techniques et d'outils formels pour la validation et la vérification des programmes synchrones écrits en LUSTRE/SCADE. En respectant la réalité industrielle et nous appuyant sur l'état de l'art concernant le test de logiciels embarqués critiques, qui est la phase la plus exigeante et rigoureuse du cycle de développement, nous avons tenté d'améliorer les techniques de test.

Ce travail de recherche s'appuie sur deux parties principales.

- En ce qui concerne l'évaluation de la couverture structurelle, nous avons proposé l'extension et l'amélioration des critères de couverture déjà définis dans [33]. L'approche de couverture proposée fournit des critères définis directement sur le modèle LUSTRE/SCADE qui peuvent évaluer la qualité de données de test ainsi que déterminer l'arrêt du test.

  D'une part, ces critères ont été étendus vers l'utilisation des plusieurs horloges afin d'élargir leur applicabilité en incluant tous les opérateurs temporels du langage LUSTRE. Plus précisément, nous avons défini la notion de la couverture pour les opérateurs `when` et `current`, qui gèrent l'utilisation d'horloges multiples en LUSTRE [47]. Ces critères ont été implantés dans l'environnement SCADE et plus précisément dans le module SCADE MTC.

  D'autre part, nous avons étudié l'extension des critères à des spécifications LUSTRE plus complexes qui utilisent des nombreux noeuds imbriqués. Nous avons proposé une approche de mesure de la couverture compatible avec une stratégie d'intégration des noeuds appelés par d'autres noeuds, évitant leur dépliage complet, et réduisant la complexité. Une notion de

couverture *abstraite* a été introduite qui a pour résultat une version plus simple du noeud de couverture obtenu. En conséquence, le calcul de la couverture d'un programme de grande taille qui contient plusieurs composants (d'autres noeuds LUSTRE) consiste à considérer ces composants comme des boîtes noires et à y appliquer directement ces critères *abstraits*. Ainsi, la taille et la complexité des noeuds de couverture obtenus sont réduits, particulièrement dans le cas du critère de conditions multiples (MCC) où le nombre de chemins et de conditions d'activation obtenus pourrait être prohibitif, même pour les applications de taille moyenne.

- En ce qui concerne la génération de données de test, nous avons étudié la modélisation de l'environnement du système comme un moyen de générer de séquences de test conformes aux besoins informels du système. Nous avons contribué à la définition d'une méthodologie [48] basée sur LUTESS, un outil de test destiné aux logiciels réactifs synchrones. Nous avons fait des recommandations concernant la construction du modèle d'environnement par rapport aux spécifications fonctionnelles du système, profitant des nombreuses stratégies de test offertes par l'outil. Nous avons, enfin, étudié la complexité des modèles de test; pour cela, on a évalué la difficulté associée à leur construction en fonction de la clarté et la taille des spécifications.

Afin d'enrichir et compléter les contributions décrites ci-dessus avec des observations empiriques, nous avons mené des expériences sur deux études de cas du domaine de l'avionique. Pour la partie évaluation de couverture, nous avons utilisé un composant de contrôle d'alarme dans un système de commande de vol d'un avion, sur lequel nous avons appliqué l'approche de couverture proposée. Les objectifs de cette étude ont consisté, entre autres, à montrer l'applicabilité des critères aux exemples extraits du monde industriel réel, à mesurer l'effort de test nécessaire à la satisfaction de chacun des critères et enfin à évaluer leur aptitude à révéler les fautes dans un programme LUSTRE au moyen du test par mutation.. Pour la partie génération de tests, nous avons utilisé LUTESS pour tester un système de contrôleur de train d'atterrissage d'un appareil militaire. Les objectifs de cette étude étaient de démontrer l'utilité de tester un système avec LUTESS dans un contexte plus réaliste et d'évaluer l'extensibilité et l'efficacité des modèles de test par rapport à la pertinence et la conformité des séquences de test générées. Dans cette étude, on a été spécialement intéressé au test guidé par les propriétés de sûreté afin de guider le processus de génération à

la violation de certaines propriétés du système. Nous avons intentionnellement introduit des fautes dans le programme sous test et nous avons vérifié l'aptitude de la technique à les détecter.

Même si d'autres études de cas sont nécessaires pour évaluer l'applicabilité et le passage à l'échelle des approches que nous avons proposées, les expérimentations que nous avons menées suggèrent qu'elles pourraient être utilisées pour la validation et la vérification d'applications industrielles réelles.

Évolutions et perspectives

Les approches proposées dans cette thèse contribuent à l'amélioration des techniques de test de systèmes embarqués synchrones mais de nombreuses questions nécessitent d'études plus approfondies.

Tout d'abord, les critères de couverture définis sont limités aux spécifications LUSTRE qui contiennent exclusivement des variables booléennes. Leur définition implique que l'activation de chemins est considérée par rapport à toutes les valeurs possibles que les entrées de chemins peuvent porter, à savoir *vrai* et *faux*. Cela signifie que, dans le cas d'entrées numériques, les critères ne sont pas pertinents. L'extension des critères à d'autres types de variables apparaît donc comme une tâche importante. Une possibilité consisterait à examiner plus en détail les spécifications du système afin de réduire autant que possible le domaine de variables d'entrée à un sous-ensemble de valeurs réellement utilisées.

De plus, les résultats obtenus par les expérimentations ont démontré que, dans plusieurs cas, une seule séquence de test satisfait plusieurs conditions d'activation. Il serait intéressant d'envisager une manière de trouver un ensemble minimal de jeux de test qui satisferait toutes les conditions d'activation liées à un critère donné. Une technique appropriée pour réduire la taille d'ensembles de séquences de test a été proposée dans [43]; cette technique consiste à appliquer de méthodes heuristiques et elle utilise le test de mutation afin de trouver un ensemble minimal de tests satisfaisant un critère de test.

Une autre extension potentielle des critères définis concerne la nouvelle version de l'environnement SCADE V6. Cette version permet l'utilisation simultanée de diagrammes de flux de données et de machines d'états finis. D'après la figure 8.1, un système peut être représentée par un graphe de flot de contrôle; de diagrammes de flux de données sont définis dans les états de ce graphe. La prise en compte de ce modèle par les critères de couverture nécessite de

redéfinir la notion d'activation de chemins dans le contexte du graphe de flot de contrôle. Par ailleurs, la couverture des états et des transitions devrait être un des objectifs de couverture.

Bibliography

[1] Do-178b, software considerations in airborne systems and equipment certification. Technical report, RTCA, Inc., www.rtca.org, 1992.

[2] Scade language reference manual. technical report SC-LRM - SC/70257u3-5.0.1, Esterel Technologies SA, Parc Avenue, 9 rue Michel Labrousse, 31100 Toulouse, France, 2005.

[3] Jean-Raymond Abrial. Steam-boiler control specification problem. In *Formal Methods for Industrial Applications*, pages 500–509, 1995.

[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[5] A. F. E. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, pages 391–438. Springer Verlag, 2005.

[6] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[7] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[8] F. Boniol, V. Wiels, and E. Ledinot. Experiences in using model checking to verify real time properties of a landing gear control system. In *ERTS 2006: 3rd European Congress Embedded Real Time Software*, Toulouse, France, January 25-27 2006.

[9] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[10] Timothy Alan Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick Gerald Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'80)*, pages 220–233, Las Vegas, Nevada, 28-30 January 1980.

[11] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[12] T. Cattel and G. Duval. The steam boiler problem in lustre. *Formal Methods for Industrial Applications*, pages 149–164, 1996.

[13] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. 9(5):193–200, 1994.

[14] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Software Eng.*, 15(11):1318–1332, 1989.

[15] Richard A. DeMillo. Test adequacy and program mutation. In *ICSE*, pages 355–356, 1989.

[16] Lydie du Bousquet, Farid Ouabdesselam, Jean-Luc Richier, and Nicolas Zuanon. Lutess: A specification-driven testing environment for synchronous software. In *ICSE*, pages 267–276, 1999.

[17] Lydie du Bousquet and Nicolas Zuanon. An overview of lutess: A specification-based tool for testing synchronous software. In *ASE*, pages 208–215, 1999.

[18] Joe W. Duran and Simeon Ntafos. A report on random testing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.

[19] Guy Durrieu and Virginie Wiels. Premiere proposition de methodologie de test. ANR Technical report 07TLOG019, June 2009.

[20] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Software Eng.*, 14(10):1483–1498, 1988.

[21] Alain Girault and Xavier Nicollin. Clock-driven automatic distribution of lustre programs. In *3rd International Conference on Embedded Software, EMSOFT'03*, volume 2855 of *LNCS*, pages 206–222, Philadelphia, USA, October 2003. Springer-Verlag.

[22] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[23] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992.

[24] Nicolas Halbwachs. A tutorial of lustre, 1993.

[25] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence (program testing). *IEEE Trans. Softw. Eng.*, 16(12):1402–1411, 1990.

[26] Richard G. Hamlet. Theoretical comparison of testing methods. In *Symposium on Testing, Analysis, and Verification*, pages 28–37, 1989.

[27] D. Harel and A. Pnueli. On the development of reactive systems. pages 477–498, 1985.

[28] Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A practical tutorial on modified condition/decision coverage. Technical report, 2001.

[29] S. Rao Kosaraju. Analysis of structured programs. *Journal of Computer and System Sciences*, 9(3):232–255, 1974.

[30] A. Lakehal and I. Parissis. Lustructu: A tool for the automatic coverage assessment of lustre programs. *In proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005)*, pages 301–310, Chicago, USA, November 2005.

[31] A. Lakehal and I. Parissis. Structural test coverage criteria for lustre programs. *In the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), a joint event of ESEC/FSE'05*, pages 35–43, Lisbon, Portugal, September 2005.

[32] A. Lakehal and I. Parissis. Automated measure of structural coverage for lustre programs: A case study. *In proceedings of the 2nd IEEE International Workshop on Automated Software Testing (AST'2007), a joint event of the 29th ICSE*, Minneapolis, USA, May 2007.

[33] Abdesselam Lakehal. Criteres de couverture structurelle pour les programmes lustre. Phd thesis, Université Joseph Fourier, Grenoble, France, September 2006.

[34] Jean-Claude Laprie. *Guide de la Sûreté de Fonctionnement*. Cépaduès, 1995.

[35] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, 1983.

[36] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[37] Bruno Marre and Agnès Arnould. Test sequences generation from lustre descriptions: Gatel. In *IEEE International Conference on Automated Software Engineering*, pages 229–237, Grenoble, France, October 2000.

[38] Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electr. Notes Theor. Comput. Sci.*, 111:93–111, 2005.

[39] Christine Mazuet. Stratégies de test pour des programmes synchrones - application au langage lustre. Phd thesis, Institut National Polytechnique de Toulouse, Toulouse, France, December 1994.

[40] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[41] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[42] Simeon C. Ntafos. An evaluation of required element testing strategies. In *International Conference on Software Engineering*, pages 250–256, Orlando, Florida, USA, March 1984.

[43] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. Procedures for reducing the size of coverage-based test sets. In *In Proceedings of the Twelfth*

International Conference on Testing Computer Software, pages 111–123, 1995.

[44] A Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. *Engineering of Complex Computer Systems, IEEE International Conference on*, 0:119, 1999.

[45] F. Ouabdesselam and I. Parissis. Testing synchronous critical software. In *5th International Symposium on Software Reliability Engineering*, Monterey, USA, november 1994.

[46] F. Ouabdesselam and I. Parissis. Constructing operational profiles for synchronous critical software. *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 286–293, Oct 1995.

[47] V. Papailiopoulou, L. Madani, L. du Bousquet, and I. Parissis. Extending structural test coverage criteria for lustre programs with multi-clock operators. *In the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, L'Aquila, Italy, September 2008.

[48] Virginia Papailiopoulou, Besnik Seljimi, and Ioannis Parissis. Revisiting the Steam-Boiler Case Study with LUTESS : Modeling for Automatic Test Generation. In *Proceedings of the 12th European Workshop on Dependable Computing, EWDC 2009 12th European Workshop on Dependable Computing, EWDC 2009*, Toulouse France, 05 2009. Helène WAESELYNCK.

[49] I. Parissis. A tool for testing synchronous critical software. In *3rd International Conference on Achieving Quality in Software*, Florence, Italie, january 1996.

[50] I. Parissis and J. Vassy. Strategies for automated specification-based testing of synchronous software. In *16th International Conference on Automated Software Engineering*, San Diego, USA, November 2001. IEEE.

[51] Ioannis Parissis. Test de logiciels synchrones spécifiés en lustre. Phd thesis, Université Joseph Fourier, Grenoble, France, September 1996.

[52] David Parnas, John van Schouwen, and Shu Po Kwan. Evaluation of Safety-Critical Software. *Communications of the ACM*, 33(6):636–648, june 1990.

[53] Ajitha Rajan. Coverage metrics for requirements-based testing. Phd thesis, University of Minnesota, Minneapolis, July 2008.

[54] Ajitha Rajan, Michael W. Whalen, and Mats P.E. Heimdahl. Model validation using automatically generated requirements-based tests. *High-Assurance Systems Engineering, IEEE International Symposium on*, 0:95–104, 2007.

[55] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, 11(4):367–375, 1985.

[56] Pascal Raymond. Compilation efficace d'un langage declaratif synchrone: le generateur de code lustre-v3. Phd thesis, Institut National Polytechnique de Grenoble, Grenoble, France, November 1991.

[57] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *IEEE Real-Time Systems Symposium*, pages 200–209, 1998.

[58] D. Richardson and L. Clarke. Partition analysis : A method combining testing and verification. *IEEE Transactions on Software Engineering*, 11(12):1477–1490, December 1985.

[59] Jacobson Ivar Booch Grady Rumbaugh, James. *The Unified Modeling Language Reference Manuel*. Addison-Wesley Longman, Inc., Reading, Massachussetts, 1998.

[60] Besnik Seljimi. *Test de logiciels synchrones avec la PLC*. PhD thesis, Université Joseph-Fourier - Grenoble I, July 2009.

[61] Besnik Seljimi and Ioannis Parissis. Using clp to automatically generate test sequences for synchronous programs with numeric inputs and outputs. In *ISSRE*, pages 105–116, 2006.

[62] Besnik Seljimi and Ioannis Parissis. Automatic generation of test data generators for synchronous programs: Lutess v2. In *DOSTA '07: Workshop on Domain specific approaches to software test automation*, pages 8–12, Dubrovnik, Croatia, 2007. ACM.

[63] Jérome Vassy. Génération automatique de cas de test guidée par des propriétés de sureté. Phd thesis, Université Joseph Fourier, Grenoble, France, October 2004.

[64] Sergiy A. Vilkomir and Jonathan P. Bowen. Formalization of software testing criteria using the z notation. In *Proceedings of COMPSAC 2001: 25th*

IEEE Annual International Computer Software and Applications Conference, pages 351–356. Society Press, 2001.

[65] Sergiy A. Vilkomir and Jonathan P. Bowen. Reinforced condition/decision coverage (rc/dc): A new criterion for software testing. In *ZB*, pages 291–308, 2002.

[66] Dolores R. Wallace and Roger U. Fujii. Verification and validation: Techniques to assure reliability. *IEEE Software*, 6(3):8–9, 1989.

[67] L. White and E. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[68] M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Trans. Softw. Eng.*, 6(3):278–286, 1980.

[69] Nicolas Zuanon. *Test de spécifications de services de télécommunication*. PhD thesis, Université Joseph Fourier, Grenoble, France, June 2000.