HAL
open science

On Model-based Testing of GALS Systems

Lina Marsso

To cite this version:

Lina Marsso. On Model-based Testing of GALS Systems. Formal Languages and Automata Theory [cs.FL]. Université Grenoble Alpes, 2019. English. NNT : 2019GREAM078 . tel-02948083

HAL Id: tel-02948083

https://theses.hal.science/tel-02948083v1

Submitted on 24 Sep 2020

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.

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Lina MARSSO

Thèse dirigée par **Radu MATEESCU**, responsable équipe CONVECS, Inria Grenoble – Rhône-Alpes,
codirigée par **Ioannis PARISSIS**, professeur, Grenoble INP Laboratoire de Conception et d'Intégration de Systèmes (LCIS),
et co-encadrée per **Wendelin SERWE,** charge de recherche CONVECS, Inria Grenoble - Rhône-Alpes
préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Étude de génération de tests à partir d'un modèle pour les systèmes GALS

On Model-based Testing of GALS Systems

Thèse soutenue le **10 décembre 2019**,
devant le jury composé de :

Monsieur Holger Hermanns
Professeur, Saarland University, Rapporteur
Madame Virginie Wiels
Directeur de Recherche, ONERA, Rapporteur
Monsieur Roland Groz
Professeur, Grenoble INP, Examinateur
Monsieur Eric Jenn
Ingénieur de Recherche, IRT SAINT-EXUPERY, Examinateur
Monsieur Dumitru Potop-Butucaru
Chargé de Recherche (HdR), Inria, Examinateur
Monsieur Radu Mateescu
DR2, INRIA, Directeur de thèse
Monsieur Ioannis Parissis
Professeur, Laboratoire de Conception et d'Intégration de Systèmes (LCIS), Co-directeur de thèse
Monsieur Wendelin Serwe
Chargé de Recherche, Inria, Encadrant de thèse

Acknowledgements

I would like to thank my PhD advisors Radu Mateescu, Wendelin Serwe and Ioannis Parissis, for their invaluable assistance and supervision throughout my PhD. I am very grateful for their thorough involvement, their scientific and educational advice and their patience without whom I would not be able to achieve my objectives. I have learned a great deal from them, and benefited from their rich and complementary perspectives, both in scientific terms and in terms of the research process. I have also deeply appreciated their day-to-day kindness and care.

My sincere thanks also goes to Holger Hermanns and Virginie Wiels, for having accepted to review my thesis, and for their precious remarks on my manuscript, and as well to Roland Groz, Eric Jenn, and Dumitru Potop-Butucaru, for accepting to be part of the committee.

I am also grateful to all the CONVECS team members, whose team spirit, enthusiasm and trust have been a great source of motivation during my PhD. I would like to thank Frederic, Gwen, Fatma, Armen, Alexito, Philippe and Lucas, for both technical and theoretical advices and for being such pleasant company. Thanks also to my fellows: Myriam, our team assistant, a friend, for her kindness and the support with all the administrative and everyday life concerns; My office mates Gianluca, Ajay, and Pierre for helping me, for making the atmosphere enjoyable, for appearing at critical times, and for their soft presence; Hubert, who, during my PhD, has been an amazing collaborator, a mentor, and who generously dedicated some of his time to our scientific and cultural discussions, it has been a great honor and pleasure to work closely with such brilliant mind and generous individual.

My thoughts go also to the dedicated researchers, PhD students, interns at INRIA for their continuous support throughout my PhD and for our fruitful scientific conversations. And, I am very fortunate and grateful also for our collaborations and interactions in the RIDINGS project, with Josip Bozic, Hermann Felbinger, Birgit Hofer and Franz Wotawa.

A special thank goes to the doctoral school assistant Zilora Zouaoui, for her precious help, and for her ability to find solutions in tedious administrative situations.

Finally, last but not least, I place on record, my sense of gratitude to my family, my teammate, for being the best, for supporting me every day, in the good and in the bad times.

Abstract

This dissertation focuses on the model-based testing of GALS (Globally Asynchronous and Locally Synchronous) systems, which are inherently complex because of the combination of synchronous and asynchronous aspects. To cope with this complexity, we explore three directions: (1) techniques for synchronous components; (2) techniques for communication protocols between components; and (3) techniques for complete GALS systems, combining the results of the two previous directions.

In the first direction, we explore formal techniques for the functional testing of synchronous components. As a case-study, we reconsider the Message Authenticator Algorithm (MAA), a pioneering cryptographic function designed in the mid-80s, and formalize it as a synchronous dataflow. The modeling and validation of the MAA enabled us to discover various mistakes in prior (informal and formal) specifications of the MAA, the test vectors and code of the ISO 1987 and ISO 1990 standards, and in compilers and verification tools used by us.

In the second direction, we explore the formalization and the functional testing of a communication protocol. As a case-study, we reconsider the formalization of the Transport Layer Security (TLS) handshake, a protocol responsible for the authentication and exchange of keys necessary to establish or resume a secure communication. Our model of the TLS version 1.3 has been validated by an approach using our new on-the-fly conformance test case generation tool, named TESTOR, developed on top of the CADP toolbox. TESTOR explores the model and generates automatically a set of controllable test cases or a complete test graph (CTG) to be executed on a physical implementation of the system.

In the third direction, we propose a testing methodology for GALS systems combining the two previous directions. We leverage the conformance test generation for asynchronous systems to automatically derive realistic scenarios (inputs constraints and oracles), which are necessary ingredients for the unit testing of individual synchronous components, and are difficult and error-prone to design manually. Thus our methodology integrates (1) synchronous and asynchronous concurrent models; (2) functional unit testing and behavioral conformance testing; and (3) various formal methods and their tool equipments. We illustrate our methodology on a simple, but relevant example inspired by autonomous cars.

Résumé

Cette thèse porte sur la génération de tests à partir d'un modèle pour les systèmes GALS (Globalement Asynchrones et Localement Synchrones). La combinaison des aspects synchrones et asynchrones en font des systèmes complexes, imposant de recourir à de nouvelles méthodes d'analyse. Pour faire face à cette complexité, nous explorons trois directions : (1) techniques pour les composants synchrones ; (2) techniques pour les protocoles de communication entre les composants ; et (3) techniques pour des systèmes GALS complets, combinant les résultats des deux directions précédentes.

Dans la première direction, nous explorons des techniques formelles pour le test fonctionnel de composants synchrones. En tant qu'étude de cas, nous reprenons l'algorithme d'authentification de message (MAA), une fonction cryptographique conçue au milieu des années 80. Nous formalisons cet algorithme en tant que flux de données synchrone. La modélisation et la validation du MAA nous ont permis de découvrir diverses erreurs dans les spécifications (informelles et formelles) préalables du MAA, les vecteurs de test et code des normes ISO 1987 et ISO 1990 ; dans les compilateurs et outils de vérification que nous avons utilisés.

Dans la seconde direction, nous explorons la formalisation et le test fonctionnel d'un protocole de communication. Dans notre étude de cas, nous évaluons le protocole d'établissement d'une liaison sécurisé au niveau de la couche de transport (TLS), responsable de l'authentification et de l'échange de clés nécessaires pour établir ou reprendre une communication sécurisée. Notre modèle de la version 1.3 TLS a été validé par une approche utilisant notre nouvel outil de génération de cas de test de conformité à la volée, nommé TESTOR, développé à partir de la boîte à outils CADP. Cet outil explore le modèle et génère automatiquement un ensemble de cas de tests ou un graphe de test complet (CTG), à exécuter sur une implémentation physique d'un système.

Dans la troisième direction, nous proposons une méthodologie de test permettant d'analyser les systèmes GALS dans leur ensemble. Nous tirons parti de la génération de tests de conformité des systèmes asynchrones pour dériver automatiquement des scénarios réalistes (contraintes d'entrées et oracles), qui sont ardus à concevoir manuellement et sujet d'erreurs. Ainsi, notre méthodologie intègre (1) modèles concurrents synchrones et asynchrones; (2) les tests unitaires fonctionnels et les tests de conformité comportementale; et (3) diverses méthodes formelles et leurs outils. Nous illustrons notre méthodologie sur un exemple simple, mais représentatif inspiré des voitures autonomes.

Contents

1 Introduction **1**

2 Formal Modelling and Validation of GALS Systems **7**
2.1 Preliminaries . . . 7
2.1.1 Labelled Transition Systems (LTS) . . . 7
2.1.2 Equivalence checking . . . 9
2.1.3 Model checking . . . 10
2.1.4 Testing and coverage criteria . . . 11
2.2 Synchronous Models and their Validation . . . 12
2.2.1 Overview of the Lustre language . . . 12
2.2.2 Synchronous system validation . . . 14
2.2.3 Overview of the Lurette testing tool . . . 15
2.3 Asynchronous Models and their Validation . . . 19
2.3.1 Overview of the LNT language . . . 19
2.3.2 Verification of asynchronous systems . . . 26
2.3.3 Asynchronous model-based testing (the ioco theory) . . . 29
2.4 GALS Models and Validation . . . 35
2.4.1 An overview of the GRL language . . . 37
2.4.2 Verification of GALS systems . . . 44

3 Validation Techniques for Synchronous Components **47**
3.1 The Message Authenticator Algorithm (MAA) . . . 47
3.2 Modelling the MAA in Lustre . . . 51
3.3 Testing the MAA Model . . . 54
3.4 Formal models of the Message Authenticator Algorithm (MAA) . . . 59

4 Validation of Communication Protocols between Components **63**
4.1 Transport Layer Security Handshake Protocol . . . 63
4.1.1 Handshake TLS 1.3 interactions . . . 64
4.1.2 Main TLS 1.3 handshake messages . . . 67
4.2 Formal Model of the TLS Handshake in LNT . . . 68
4.2.1 Handshake interactions . . . 68

4.2.2 Handshake messages . . . 69
4.3 TESTOR: On-the-Fly Conformance Test Case Generation . . . 71
4.3.1 Architecture . . . 71
4.3.2 On-the-fly test selection algorithm . . . 73
4.3.3 Examples of different ways to express a test purpose . . . 74
4.3.4 Experimental comparison of TESTOR and TGV . . . 77
4.3.5 Comparison of TESTOR and other MBTs . . . 80
4.4 Testing the TLS Handshake Model with TESTOR . . . 81

5 Validation of GALS Systems 85
5.1 GALS Example: An Autonomous Car . . . 85
5.2 GRL Model of the GALS System . . . 86
5.3 Model Checking of the GALS Behavior . . . 92
5.4 Manual Testing of a Synchronous Component . . . 94
5.5 Test Projection and Exploration . . . 97
5.5.1 Test graph projection . . . 97
5.5.2 Translating and renaming . . . 99
5.5.3 Test graph exploration . . . 99

6 Conclusion 105

A Formal Model of the MAA in Lustre 109
A.1 Definitions for type Bit . . . 109
A.2 Definitions for Type Byte . . . 110
A.3 Definitions for Type OctetSum . . . 123
A.4 Definitions for Type Half . . . 124
A.5 Definitions for Type HalfSum . . . 125
A.6 Definitions for Type Block . . . 126
A.7 Definitions for Type BlockSum . . . 132
A.8 Definitions for Type Pair . . . 133
A.9 Definitions for Type Key . . . 134
A.10 Definitions (1) of MAA-specific Cryptographic Functions . . . 134
A.11 Definitions (2) of MAA-specific Cryptographic Functions . . . 136
A.12 Definitions (3) of MAA-specific Cryptographic Functions . . . 137
A.13 Test Vectors (1) for Checking MAA Computations . . . 140
A.14 Test Vectors (2) for Checking MAA Computations . . . 142
A.15 Test vectors (3) for Checking MAA Computations . . . 148
A.16 Test Vectors (4) for Checking MAA Computations . . . 152
A.17 Functional Testing of MAA with Lurette . . . 155

B Formal Model of a Simple Autonomous Car in GRL 157
B.1 Definitions for the Synchronous Blocks . . . 157
B.2 Definitions for the Asynchronous Mediums . . . 160

B.3 Definitions for the Asynchronous Environement 161
B.4 Definitions of the global GALS system 165
B.5 Definitions of LNT library 166
B.5.1 Definitions of LNT Graph Functions and Types 166
B.5.2 Definitions of LNT Datatypes 170
B.5.3 Definitions of the Scenario 179

C XTL Scripts Extracting Synchronous Test Scenarios 183
C.1 The Generation of Input Constraints (`extract_input`) 183
C.2 The Generation of Oracles (`extract_oracle`) 185

List of Figures

1.1 Overview of the testing methodology for GALS systems 5

2.1 The LTS corresponding to a simple coffee/tea vending machine 8
2.2 The IOLTS (and the IOTS in green) corresponding to the LTS in Figure 2.1 9
2.3 Overview of the Lurette testing tool 15
2.4 Simple scenario automaton with the inputs values of the heater (`actual_room_temperature` and `season`) 16
2.5 Part of the oracle automaton with output `new_room_temperature` and input `season`, corresponding to an initial value of 20 for `actual_room_temperature` 17
2.6 Examples of the implementation relation **ioco**. 31
2.7 Model M of the running example [JJ05] 33
2.8 Test purpose TP of the running example [JJ05] 33
2.9 The visible behavior SP_{vis}, complete test graph CTG (gray), and a test case TC (dark gray) of the running example [JJ05] 34
2.10 Simple smart room temperature management 36
2.11 Architecture of the GRL model of the room temperature management . . . 38
2.12 Overview of the tools and languages selected for this thesis 45

3.1 MAA data flow 48
3.2 Mode of operation of the MAA 50
3.3 The node MAC 53
3.4 The function prelude 54
3.5 Overview of the Lurette testing tool 55
3.6 The MAC node rewritten 56
3.7 Environment 56
3.8 Tests (T3) inspired by Table 5 of [DC88]. 57
3.9 Oracle 58

4.1 TLS handshake client 65
4.2 TLS handshake server 66
4.3 Architecture of TESTOR 71
4.4 The visible behavior SP_{vis}, complete test graph CTG (gray), and a test case TC (dark gray) of the running example [JJ05]. 73

4.5 TC I: TLS handshake with classical TLS 1.3 order 81
4.6 TC II: TLS handshake aborted with an unexpected Alert 81
4.7 TC III: TLS handshake with renegotiation 82

5.1 Simple autonomous cars 86
5.2 Geographical map example with the autonomous car and a pedestrian . . . 87
5.3 Architecture of the GRL model of an autonomous car 88
5.4 GRL geographical map representation 89
5.5 Geographical map, with natural numbers identifying the streets fragments. 94
5.6 Simple scenario automaton 95
5.7 Overview of the derivation of synchronous test scenarios by projection and exploration of an asynchronous complete test graph CTG 97
5.8 Test Purpose $T1$ 98
5.9 Overview of the testing methodology for GALS systems 103

List of Tables

2.1 Synchronous test generation tools 15
2.2 Comparison of **ioco** model-based testing tools 32
2.3 Approaches to model GALS systems 37

4.1 Run-time performance for selected examples 77

5.1 Sizes and run-time performance for the complete test graphs for the test purposes 98
5.2 Sizes and run-time performance for the tests generated for the test purposes 99

Chapter 1

Introduction

"On March 18, 2018, in Tempe, Arizona at 10pm, for the first time, an autonomous vehicle, more particularly the Uber SUV, hit and killed a pedestrian. Per that report, the Uber car's radar and LiDAR sensors were able to detect an object in the path of the vehicle (the pedestrian) about six seconds before the crash, but it misclassified her as an unknown object, then a vehicle, then finally a bicycle. At 1.3 seconds before impact, the system tried to initiate an emergency braking maneuver, but Uber had deactivated the vehicle's factory-equipped automated emergency braking system to help ensure less erratic testing." [Eco18]

This crash, which killed a pedestrian, is the consequence of an error in the system controlling the car. The LiDAR (Light Detection and Ranging) sensor can detect obstacles perfectly well in the dark, with the same precision as in daylight. However, the sensor is only responsible for the *perception* of the environment and does not make the decision to brake the car, to slow down or to choose another route: these decisions are up to the *decision* controller of the vehicle, which has to interpret and use the sensor data to initiate appropriate actions.

The Uber car is an example of a critical system, i.e., a system whose failure or malfunction may result in at least one of the following outcomes: death or serious injury to human, loss or severe damage to equipment, and environmental harm. Nowadays, critical systems can be found in almost any domain, such as infrastructure, medicine, nuclear engineering, transport (railway, aviation, automobile, spaceflight). Because a malfunction of these systems might endanger human life or cause an environmental damage, ensuring correct functioning of critical systems is more important than ever.

Testing is still one of the most common practices to ensure that a system does but only what it was designed to do. Testing enables to identify errors in a system and also any missing requirements in the design of a system. Testing can be done manually or using automated tools. One can distinguish three testing approaches. One can either test the internal structure, design, and coding of the system, which is called *white box* or *glass* testing. For instance, the system performing the perception of a LiDAR in an autonomous

car could be tested by analyzing its internal code. Otherwise, one can test the behavior, without looking at the internal code structure, implementation details and knowledge of internal system internals, which is called *black box* testing. For instance, the same system performing the perception of a LiDAR can be tested by just focusing on the inputs and observing the outputs without knowing the internal implementation. White-box testing is difficult to achieve, because in order to test the global system one should access all its internal details, which costs time and memory if it is even feasible at all. On the other hand, black-box testing is sometimes too restrictive, especially when testing security systems, where the principal information is hidden in the internal system. Between black-box and white-box testing, an intermediate testing approach exists, called *grey-box* testing, where the dynamic behavior of the system is described e.g., using high-level models in terms of states and transitions systems (also called automata). Grey-box testing is based entirely on the requirements and specifications. In this dissertation, we focus on grey-box testing, and more particularly using formal methods for testing [BGM91, Gau95], as it is required for the validation of critical systems [MLD+13].

Formal methods are mathematically well-founded techniques designed to assist the development of correct complex systems [GG13]. A central idea of formal methods is to consider systems as mathematical objects that can be described and analyzed rigorously. For the description, formal methods are often associated with computer languages having a formal semantics that could either describe the properties expected from a system or the observable behavior of the system. For the analysis, formal methods are often equipped with tools, ensuring that the system will function as expected, under certain assumptions. A formal model can be used to verify and test a system. An important difference between testing with formal methods and traditional testing techniques is the capacity of formal methods to automatically analyze and test a large amount of possible and (by construction) relevant behaviors of the system, and not only a few ones. Relevant behaviors characterize interesting situations, including the ones that the system should do, that the system should not do, and the ones stressing the system. This is essential for critical systems, where the correct functioning of the system has to be mathematically demonstrated, and not only estimated.

Model-Based Testing (MBT) [BJK+05] is a grey-box testing technique taking advantage of a formal *model* of a *system* to automate the generation of relevant *test cases*. Here, the model represents the expected behavior of the system, in other words, it should contain all possible behaviors of the system. The model can also include a description of the system's environment, so as to precisely describe what the system expects from its environment, i.e., what the environment can do, and what it cannot do. Relevant test cases are then generated from this model, to ensure that the tester examines the system behavior from as many angles as possible. A *test case* is a set of step-by-step instructions to test if the system behaves as required. A test case usually contains the test steps, the expected result, and the verdict corresponding to the result of the test execution. The MBT technique is particularly suitable for concurrent systems, such as communication protocols [WTK00], distributed systems [GBHG14], and multiprocessor architectures [GVZ00]. The model

describes the system's requirements, thus the test cases generated are directly linked to these requirements, which allows readability, understandability, and maintainability of the test execution. Model-based testing also helps to ensure a repeatable scientific basis for testing and has the potential to measure the coverage of the system's behavior [Utt05]. Moreover, it is a way to reduce effort and cost and improve the quality of testing [BJK+05]. In this thesis we focus primarily on model-based testing of a specific kind of critical systems, namely those that contain a reactive controller.

Reactive controllers [Ber89, Hal98] are characterized by the fact that they continuously interact with their environment, by observing it (via input sensors) and modifying it (via output commands) to obtain the desired behavior. A simple example is the control of the room temperature by commanding a heater. The synchronous approach [Hal93] to reactive systems programming supposes that the controller operates triggered by a clock, such that at each clock pulse the system computes the outputs from the current inputs and its internal state before the next clock pulse. Thus, time can be considered as discrete and computation as atomic. There is no need to consider fluctuation and clock synchronization difficulty, such as unpredictable message delay, different speed of components, and physical distance. The conjunction of component actions at the same clock pulse constitutes an action of the whole system. This synchronous approach is mainly supported by synchronous languages, such as Esterel [BG92], Lustre [HCRP91], and Signal [BGJ91]. The simplicity of this abstraction is the reason for the success of the synchronous approach, which has been widely used for over two decades for the design and analysis of safety critical systems in various application domains (e.g., avionics, railway transportation, nuclear plants, etc.). For instance, Lustre, industrialized under the name SCADE [Ber07], serves for the control of nuclear power plants, flight control software and in railway systems. SCADE models can be validated with the synchronous MBT tool GATeL [MA00].

Critical systems are more and more often distributed, involving several components running independently without a global clock and interacting with each other. For such systems, the strong synchronous assumptions about time and order of events is not realistic anymore, because of the concurrency between independent components of these systems and message-passing communication. The asynchronous approaches [GR84, Gol88, GL88] are more appropriate for describing these systems, the concurrency between components being reduced to a nondeterministic choice between the possible sequences of the component actions, and these approaches are able to capture the nondeterminism induced by unknown message transmission delays. This model of concurrency is mainly supported by process algebras, such as CCS [Mil89], CSP [Hoa85], ACP [BK85], and LNT [GLS17]. Particularly suited for nondeterministic concurrent systems, conformance testing aims at extracting from a formal model of a system a set of test cases to assess whether an actual implementation of the system under test conforms to the model. In this setting, the most prominent conformance relation is input-output conformance (**ioco**) [dT01, Tre08].

Nowadays, modern asynchronous systems, such as autonomous cars or the Internet of Things, are not only increasingly large and distributed, but also consisting of multi-

ple synchronous components that execute independently and interact with each other. These systems are adequately described as GALS (*Globally Asynchronous, Locally Synchronous*) [Cha84], i.e., composed of concurrent synchronous reactive systems interacting with each other asynchronously, by means of message passing with non-zero communication delays. For instance, in an autonomous car, the perception devices (radar, GPS, cameras, etc.) and the engine controls (decision-maker) are separate components, located in various places of the car, operating independently and connected through communication links. Each of these components might be considered and implemented as a synchronous reactive system, but the complete autonomous car is rather a GALS system.

The GALS approach allows the smooth integration and reuse of existing and time-proven synchronous components when designing new systems that no longer fit into the framework of synchronous programming. Rigorous design approaches, based on formal methods and assisted by efficient validation and verification tools exist for the separate modeling and analysis of synchronous or asynchronous parts of a GALS system, but their combination can further improve the effective support in the design of a complete GALS system. In this thesis we consider three aspects of MBT for GALS systems: synchronous, asynchronous, and their combination.

Contributions

In this thesis we bring two main contributions.

First, we propose a new on-the-fly MBT tool called TESTOR, which explores the model and generates automatically a set of controllable test cases or a complete test graph to be executed on a physical implementation of the system. The generated test cases are automata that attempt to drive a system under test towards these desired states. TESTOR extends the algorithms of the conformance test generation tool TGV [JJ05] to extract test cases completely on the fly (i.e., during test case execution against the system under test), making TESTOR suitable for online testing. Concretely, given a formal specification of a system and a test purpose, TESTOR automatically generates test cases, which assess using black box testing techniques the conformance to the specification of a system under test. In this context, a test purpose describes the goal states to be reached by the test and enables one to indicate parts of the specification that should be ignored during the testing process. Compared to the conformance test generation tool TGV, TESTOR has a more modular architecture, based on generic graph transformation components, is capable of extracting a controllable test case completely on the fly, and enables a more flexible expression of test purposes, taking advantage of the multiway rendezvous of LNT.

Then, we propose a testing methodology for GALS systems that integrates (1) synchronous and asynchronous concurrent models; (2) functional unit testing and behavioral conformance testing; and (3) various formal methods and their tool equipments. Figure 1.1 gives

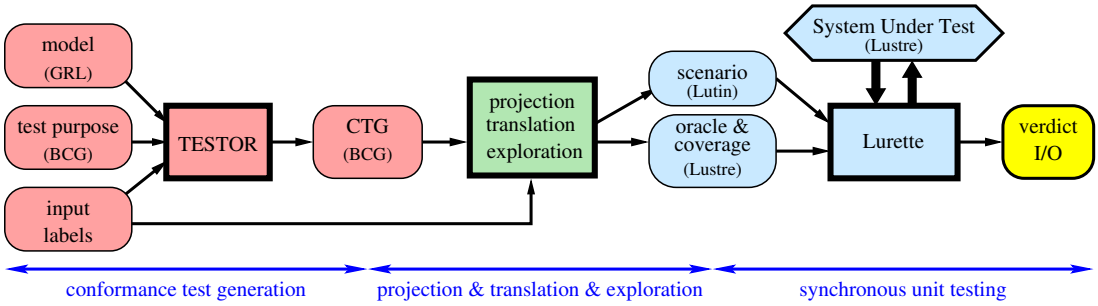


Figure 1.1: Overview of the testing methodology for GALS systems

an overview of our GALS testing approach, combining the existing synchronous testing tool Lurette presented in Chapter 2 and used in Chapter 3 and our new on-the-fly conformance test case generation tool TESTOR described in Chapter 4. The idea is to exploit the information gathered by the analysis of a globally asynchronous system to automate and finely tune the testing of its individual synchronous components. We illustrate our approach on a simple, but relevant example, namely an autonomous car, which has to reach a destination, following roads on a map, in the presence of moving obstacles. The car is modeled as a GALS system, comprising synchronous components for perception, decision, and action. As far as we know, this is the first approach of model-based testing for GALS systems.

Structure of this thesis

This dissertation focuses on the MBT of GALS systems, taking into account synchronous and asynchronous aspects. Chapter 2 briefly presents the scientific background necessary for the subsequent chapters and also places our work with respect to the state of the art.

Chapter 3 explores formal techniques for the functional testing of synchronous components, using the testing tool Lurette [JRB06] for synchronous programs. As a case-study, we reconsider the Message Authenticator Algorithm (MAA), a pioneering cryptographic function designed and standardized by ISO in the mid-80s. Formally modeling and validating the MAA as a synchronous dataflow enabled us to discover various mistakes in prior (informal and formal) specifications of the MAA, the test vectors and code of the ISO standards, and in some of the compilers and verification tools used by us.

Chapter 4 describes the formalization and the functional testing of a communication protocol and presents our new on-the-fly conformance test case generation tool, named TESTOR [MMS18], developed on top of the CADP toolbox [GLMS13]. As a case-study, we reconsider the formalization of the draft Transport Layer Security (TLS) handshake version 1.3 [IET18], a protocol responsible for the authentication and exchange of keys necessary to establish or resume a secure communication. Our TLS model has been vali-

dated using TESTOR.

In Chapter 5, we leverage the conformance test generation for asynchronous systems to automatically derive realistic scenarios (input constraints and oracles), which are necessary ingredients for the unit testing of individual synchronous components, and are difficult and error-prone to design manually. We illustrate our approach on a simple, but relevant example of an autonomous car modeled as a GALS system.

Finally, Chapter 6 concludes by summarizing our main contributions and discussing directions for future work.

Chapter 2

Formal Modelling and Validation of GALS Systems

This chapter presents first some preliminary notions that we use in the rest of this thesis. It also defines the perimeter of this thesis by listing aspects considered to be out of scope, together with the reasons justifying this choice.

The rest if this chapter is structured in three parts, corresponding to the different techniques required for the analysis of GALS systems: (1) techniques for synchronous components; (2) techniques for the asynchronous communication; and (3) techniques for complete GALS systems. For each part we present tools, languages, and an overview of the ones we used. A reader familiar with our languages and tools can safely skip the overviews.

2.1 Preliminaries

In order to describe formally the behavior of discrete and concurrent systems, one may use Labelled Transition Systems (LTS). In this section we define informally and formally LTS, and we describe three approaches to validate them, namely equivalence checking, model checking, and testing.

2.1.1 Labelled Transition Systems (LTS)

An LTS is a state-transition graph, in which the states do not provide information except for the indication of the initial state. The information is represented in the labels (or actions) related to transitions. Formally, an LTS (Q, A, T, q_0) consists of a finite set of states Q, a set of actions A, a transition relation $T \subseteq Q \times A \times Q$, and an initial state $q_0 \in Q$. A transition $(q_1, a, q_2) \in T$ (also noted $q_1 \xrightarrow{a} q_2$) indicates that the system can move from

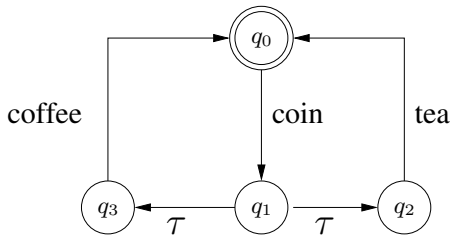


Figure 2.1: The LTS corresponding to a simple coffee/tea vending machine

state q_1 to state q_2 by performing action a. Internal transitions, i.e., transitions whose label is not visible, are labeled by the action τ. An LTS is said nondeterministic if there is more than one initial state or if it exists a state $q \in Q$ and an action $a \in A$, where more than one state can be reached from q by performing the same action a. An example LTS is shown in Figure 2.1, modelling the simple behavior of a coffee vending machine. The set of states of this LTS is $Q = \{q_0, q_1, q_2, q_3\}$. The set of actions is $A = \{$*coin, coffee, tea*,$\tau\}$. The initial state is q_0. An LTS can be represented explicitly, such as the textual AUTomaton (AUT) format [FGK+96] and the Binary Coded Graph (BCG) format [Gar91]; or implicitly, by providing the post function iterating over the outgoing transitions of a given state, for instance using the Open/Caesar API [Gar98].

The LTS model of a system represents exhaustively its behavior, which can be described intuitively as follows. The LTS starts with its initial state. From this initial state, the LTS evolves according to its transition relation. Very often, more than one transition will be possible from a given state. In that case, a transition is selected non-deterministically. Coming back to Figure 2.1, from the initial state (q_0), only one transition is possible. So, the LTS of the vending machine will always perform a coin action first and reach q_1. In q_1, there are two internal steps (τ). The LTS system will non-deterministically choose one of them and then either perform a coffee or a tea action.

Individual execution sequences in an LTS are called *traces*. A trace of size $n \in \mathbb{N}$ is a sequence of actions $a_1, a_2, \ldots, a_n \in A$ such that $q_0 \xrightarrow{a_1} q_1 \in T, q_1 \xrightarrow{a_2} q_2 \in T, \ldots, q_{n-1} \xrightarrow{a_n} q_n \in T$. A trace starts always with the initial state (q_0). A trace is potentially infinite, due to the presence of cycles in the LTS. The traces of the LTS in Figure 2.1 consist of (infinite) repetitions of one or both of the following cycles:

i) $q_0 \xrightarrow{coin} q_1 \xrightarrow{\tau} q_2 \xrightarrow{tea} q_0$

ii) $q_0 \xrightarrow{coin} q_1 \xrightarrow{\tau} q_3 \xrightarrow{coffee} q_0$

In this thesis, we also use a particular kind of LTS with the distinction between input and output actions (LTS $(A_i \cup A_o)$), called Input-Output Labelled Transition Systems (IOLTS) [Tre99]. An IOLTS is an LTS (Q, A, T, q_0) where the set of actions is partitioned in $A = A_{\mathrm{I}} \cup A_{\mathrm{O}} \cup \{\tau\}$, where $A_{\mathrm{I}}, A_{\mathrm{O}}$ are the subsets of input and output actions, and τ is the internal (unobservable) action. Input (resp. output) actions are noted $?a$ (resp.

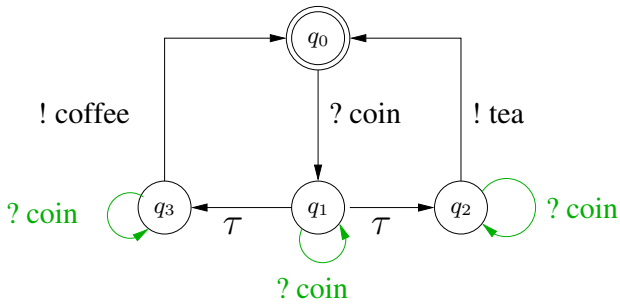


Figure 2.2: The IOLTS (and the IOTS in green) corresponding to the LTS in Figure 2.1

$!a$). An example of IOLTS is illustrated in black in Figure 2.2, modelling the behavior of a vending coffee machine. The machine *receives* coins, represented as input (`?coin`) and *delivers* beverages, represented as outputs (`!tea` and `!coffee`) in the LTS. In order to avoid any bias in the observation of the input actions of an IOLTS, the IOLTS can be *input enabled*. This particular kind of IOLTS, with the property that any input action is always enabled in any state, is called *IOTS* [Tre99]. An example of IOTS is illustrated in Figure 2.2 with the additional input transitions in green.

2.1.2 Equivalence checking

Equivalence checking is a verification method consisting in comparing two LTSs modulo an equivalence relation. There exist several equivalence relations in the literature, such as strong equivalence, branching equivalence [vGW89], trace equivalence [FM91], weak trace equivalence [BHR84], etc. We focus on two of them, namely the *branching* equivalence and the *weak trace* equivalence.

Two LTS $M = (Q, A, T, q_0)$ and $M' = (Q', A', T', q'_0)$ are equivalent according to an equivalence relation $\sim$ (written $M \sim M'$), if and only if their initial states are equivalent modulo $\sim$ (written $q_0 \sim q'_0$).

Branching equivalence Two states p and q are branching equivalent (called $p \sim_{bra} q$), if and only if, it exists a relation R_{bra} such that $pR_{bra}q$ and:

1. for each transition $p \xrightarrow{a} p' \in T$
   - either $a = \tau$ and $p'R_{bra}q$
   - or it exists a sequence $q \xrightarrow{\tau^*} q' \xrightarrow{a} q'' \in T'^*$ such as $pR_{bra}q'$ and $p'R_{bra}q''$
2. for each transition $q \xrightarrow{a} q' \in T'$
   - either $a = \tau$ and $pR_{bra}q'$
   - or it exists a sequence $p \xrightarrow{\tau^*} p' \xrightarrow{a} p'' \in T^*$ such as $p'R_{bra}q$ and $p''R_{bra}q'$

Weak trace equivalence Two states p and q are weak trace equivalent (called $p \sim_{wtr} q$), if and only if:

1. for all $n \geq 0$ and all sequences $p \xrightarrow{\tau*.a_1 \ldots \tau*.a_n} p' \in T^*$, there exists a sequence $q \xrightarrow{\tau*.a_1 \ldots \tau*.a_n} q' \in T'^*$
2. for all $n \geq 0$ and all sequences $q \xrightarrow{\tau*.a_1 \ldots \tau*.a_n} q' \in T'^*$, there exists a sequence $p \xrightarrow{\tau*.a_1 \ldots \tau*.a_n} p' \in T^*$

The CADP (Construction and Analysis of Distributed Processes) verification toolbox [GLMS13] provides the tools BISIMULATOR [BDJM05, MO08] and BCG_CMP [Vas10] to compare LTSs, the tool REDUCTOR [Vas04] to reduce LTSs according to the strong or weak trace equivalence, and the tool BCG_MIN [BHH+06] to minimize LTSs according to strong or branching equivalence.

2.1.3 Model checking

Model checking [CGP01] is a verification method establishing that a model of a system (respectively, component) satisfies a property. A property defines how the system should be designed and which features it should provide. Informally, model checking produces a Boolean answer to the question: *Does a given system satisfy a property ?* Most model checking approaches add a diagnostic that answers to the question: *Where is this property not satisfied by this system ?* Concretely, model checking defines a satisfaction relation between a system (respectively, component) and a property as a mathematical binary relation that is true if and only if the property holds for the system, i.e., if the system correctly implements the property. The system is described by a finite state model (e.g., an LTS) and a property is expressed in temporal logic.

Examples of properties are simple assertions, stating that a predicate on system variables holds whenever the execution reaches a particular control location (e.g., the transition `coffee` is reachable whenever a coin with the value `20` was inserted), or termination properties (e.g., "the vending machine system terminates on a `coffee` or `tea` transition"). Principally, properties are classified as liveness, fairness and safety. A liveness property expresses that something good eventually happens, e.g., all transition sequences starting after an action `coin` eventually lead to an action `tea` or `coffee`. A fairness property expresses that, under certain conditions, something should always happen, e.g., after an action `coin`, all fair execution sequences (which avoid the infinite repetition of cycles) will lead to an action `tea` or `coffee`. A safety property expresses that something bad never happens, e.g., if the action `coin` happens, then while there is no action tea or coffee, no other action `coin` may happen.

Numerous model checkers exist for synchronous and asynchronous systems, verifying in the supported modeling formalism and property languages (stochastic, probabilistic, timed, etc.) including (in alphabetical order): CADP [GLMS13], CPAchecker [BK11], Cubi-

cle [CGK+12], DREAM [PCdVA12], FDR [GABR14], Java Pathfinder [VHBP00], KIND 2 [CMST16], LESAR [SSC+04], mCRL2 [CGK+13], NuSMV [KTK09], Prism [HKNP06], SAL 2 [dMOR+04], SLAM [BR01], SPIN [Hol97], TLA+ [Lam93], UPPAAL [BLL+95], XEVE [Bou98], and Zing [AQR+04].

2.1.4 Testing and coverage criteria

Exhaustive testing is infeasible in practice, because the systems are complex and their corresponding LTSs are large, the space of possible test cases being generally too big to cover exhaustively in a reasonable time. Thus, strategies are required to decide when to stop testing. Coverage criteria serve to decide when to stop testing, by measuring the amount of the system already tested. More particularly, a coverage criterion is a measure used to describe the degree to which the internal structure of a system has been exercised when a particular a set of test cases, called a *test suite*, runs. Coverage criteria could also guide automatic test generation, otherwise the test generation is simply random. One can distinguish two complementary coverage methodologies. Functional coverage tries to measure how scrupulously a system under test is tested against its informal specification, e.g., system requirements. Functional coverage is helpful to reveal functionality defects. Structural coverage tries to quantify how much the internal structure of a system under test is exercised during its test process. Structural coverage is helpful for detecting unused fragments of the system, as well as highlighting the fragments that have not been properly exercised.

In this section, we focus on a particular structural coverage for model based testing, called *model coverage* [UPL12]. Model coverage quantifies how much of the internal structure of the model (e.g., a LTS) is explored and unexplored during the test process. Five coverage criteria defined with respect to an LTS for the model coverage of a concurrent system are proposed in [TLK92]:

1. *all labels* criterion, which covers all labels of the LTS at least once, a same label may occur on several transitions in the LTS;
2. *all states* criterion, which covers all states of the LTS at least once;
3. *all transitions* criterion, which covers at least once all transitions in the LTS;
4. *all proper paths* criterion, which covers all finite trace in the LTS without any identical states;
5. and *all paths* criterion, which covers all paths of the LTS at least once, which is impossible to achieve in finite time if the LTS contains cycles.

A specific model coverage, called *dataflow coverage criteria* [FW88, Lak06], can be applied to any modelling notation that contains variables. For instance, the academic tool Lustructu [Lak06, LP05] includes these criteria for the synchronous dataflow language Lustre.

One can also directly specify custom criteria in the model test coverage module of the SCADE environment [Ber07].

For large systems, it is not realistic to describe explicitly the LTS of the system. Such large LTSs are described using higher level formalism, and in the next sections we present synchronous, asynchronous, and GALS specific higher level formalisms.

2.2 Synchronous Models and their Validation

The synchronous paradigm [Hal93] for developing reactive systems is based on the *synchronous hypothesis*: the time is discrete, and the inputs of the system are only changed at a clock tick. Thus, defining the outputs of the system by equation on the inputs and the current state is sound if the outputs are computed faster than the sampling rate. Synchronous models enable to describe, simulate and verify reactive systems and to compile the code or the hardware ensuring the same behavior described in the model, if the synchronous hypothesis is verified. Synchronous programs have a cyclic behavior: at each cycle, all inputs are read, processed and all outputs are transmitted simultaneously and theoretically instantly. In the state of the art, one can principally distinguish two synchronous paradigms of programming:

- The synchronous dataflow languages as Lustre [HCRP91]. The dataflow model is based on a block diagram description. A block diagram can be described by a system of equations. A system is made up of a network of operators acting in parallel and at a pre-established pace.
- The control oriented languages as Esterel [BG92, PEB07]. This programming paradigm is particularly suited to the description of systems whose behavior changes frequently and depending on the data.

In this thesis we will use the synchronous data flow language Lustre taking advantage of its connection to robust testing tools, actively supported and maintained by developers. Moreover, Lustre is at the basis of industrial environments, such as SCADE [Ber07].

2.2.1 Overview of the Lustre language

In this section we give an overview of the Lustre language, readers familiar with Lustre can safely skip this section. Lustre [HCRP91] is a synchronous language based on the dataflow model. At each cycle, the value of the previous inputs can be memorized. The outputs of a program at a given instant may depend on these memorized inputs, but not on future inputs.

The relations between inputs and outputs are expressed using constants, operators, auxiliary variables, and functions. Lustre provides some predefined types, i.e., Boolean (`bool`),

integer numbers (`int`), real numbers (`real`), and strings (`string`). For example, the following Lustre constant `average_temperature` is defined as the integer 20, and could be used to specify the desired average temperature in a room for a heater.

```
(* temperature in Celsius *)
const average_temperature:int = 20;
```

The user can also define record, array and enumeration types. For example, the following Lustre enumeration `seasons` corresponds to the four seasons:

```
type seasons = enum {Winter, Spring, Summer, Autumn};
```

In addition to built-in operators for the predefined types, and all classical arithmetic and logic operators, Lustre enables to define functions. For example, the following Lustre function `season_to_temperature` associates a temperature to be set by the heater for each season.

```
function season_to_temperature (s: season)
        returns (temperature: int);
let
  temperature = if s = Winter then average_temperature + 2
                else if s = Spring then average_temperature
                else if s = Summer then average_temperature - 2
                else average_temperature + 1;
tel;
```

Lustre offers also the possibility to define new, complex operators, called *nodes*. Each description written in Lustre is built up of a network of nodes. A node describes the relation between its input and output parameters using a system of equations. A node has a set of unordered equations that define each of the output parameters according to the actual or previous input parameters. Being a data flow language, Lustre handles infinite sequences of values, called *streams*.

In addition, Lustre has two specific flow manipulation operators for the nodes:

- the memory operator **pre** ("previous") refers to the value of an input or output variable at the previous cycle;
- the initialization operator $->$ ("followed by") initializes a stream.

The code below represents a Lustre node corresponding to a simple synchronous heater. The heater increases gradually the temperature (`new_room_temperature`), taking into account the previous temperature (**pre** `new_room_temperature`) and the expected temperature of a room (the variable `expected_temperature`). The input `actual_room_temperature`, denoting the actual temperature of the environment (the room), is only used to initialize the value of the room temperature. The input `season`, denoting the actual season, is always taken into account to compute the expected temperature according to

the season. The output `new_room_temperature` is initialized with the actual temperature (`actual_room_temperature`).

```
node heater (actual_room_temperature: int, season: seasons)
    returns (new_room_temperature: int);
var expected_temperature: int;
let
  expected_temperature = season_to_temperature (season);
  new_room_temperature = actual_room_temperature ->
            if pre new_room_temperature < expected_temperature then
              pre new_room_temperature + 1
            else pre new_room_temperature;
tel
```

For a more detailed presentation of Lustre see [JRH19].

2.2.2 Synchronous system validation

In this section, we present tools enabling the validation of reactive systems, more particularly model checking and testing tools.

Synchronous models can be verified by using model checking. One can distinguish two kinds of synchronous model checkers. Symbolic model checkers use logical representations of sets of states, such as Binary Decision Diagrams (BDD) to describe regions of the state space that satisfy the properties being evaluated such as: LESAR [SSC+04], NuSMV [KTK09], SLAM [BR01] and XEVE [Bou98]. Model checkers based on satisfiability modulo theories (SMT) solvers reason about infinite state models containing real numbers and unbounded arrays, such as: KIND 2 [CMST16], Cubicle [CGK+12], and SAL 2 [dMOR+04]. In this thesis, we did not use synchronous verification to validate synchronous components, but we used synchronous test generation tools.

Model based testing of synchronous models includes automating test data generation. Automating the test data generation consists in generating inputs for testing a synchronous system. Often the inputs are generated from a model and a description of constraints on the inputs of this model. The generation could also be guided by a scenario, in order to generate relevant tests. The test generation can use a white box approach, such as Gatel [MA00], or black box approach, such as Lutess v1 [dBORZ99], Lutess v2 [SP07], Lurette [JRB06], and Testium [MDP14, MDP+16]. In this thesis, we focus on automating the test data generation for testing the synchronous components of a GALS system.

Table 2.1 compares synchronous test data generators according to: (1) the testing paradigm (white or black box); (2) the programming paradigm automating the test sequence generation, more particularly Constraint Logic Programming (CLP) or Binary Decision Diagrams (BDD); (3) the presence of a scenario guiding the test sequence generation; and (4) the presence of nondeterminism in the input constraint descriptions.

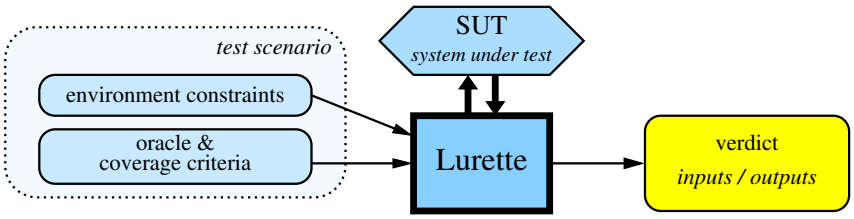


Figure 2.3: Overview of the Lurette testing tool

Table 2.1: Synchronous test generation tools

tool	test paradigm	programming paradigm	scenario language	nondeterminism	time
Lutess v1	black	BDD	behavioral model	no	no
Lutess v2	black	CLP	no	no	no
Lurette	black	BDD	Lutin scenario	yes	Lustre
Gatel	white	CLP	no	no	Lustre
Testium	black	CLP	SPTL scenario	yes	no

In this thesis, we use the black box testing tool Lurette for validating synchronous components, taking advantage of its connection with the Lustre language.

2.2.3 Overview of the Lurette testing tool

In this section we give an overview of the testing tool Lurette [JRB06], readers familiar with it can safely skip this section. Using formal specifications of the input constraints and an oracle, Lurette automates both, the generation of appropriate inputs for the SUT (system under test) and the decision about the test result.

Figure 2.3 (inspired by [JHR13, Figure 3]) gives an overview of Lurette. The tool takes two inputs:

(i) a specification of the environment constraints in Lutin [RRJ08] to dynamically constrain the inputs;

(ii) an oracle in Lustre [HCRP91] implementing the test decision and the coverage-computation of the generated and executed input sequences, and the coverage criteria to evaluate these input sequences generated.

Lurette interacts with the SUT, logs the generated sequence of inputs and their corresponding outputs in a file, and displays the test decisions. The admissible test inputs are generally provided by a formal specification of the system environment. The correct corresponding/respective outputs to an input vector are generally provided as formal properties of the system.

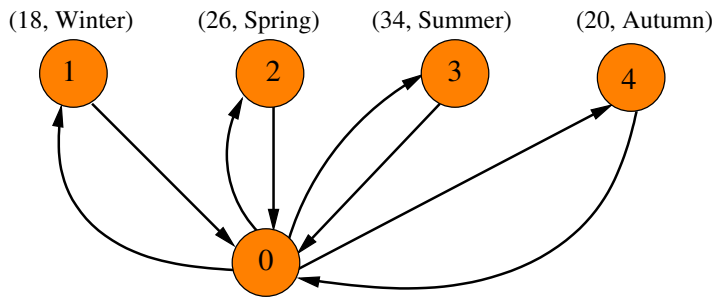


Figure 2.4: Simple scenario automaton with the inputs values of the heater (`actual_room_temperature` and `season`)

Lutin input constraints

The test sequence is generated uniformly and randomly by Lurette. Lurette builds and submits to the SUT an input vector respecting the input constraints description. Then the tool calculates a new input vector for SUT, based on the current state of the input constraints and the last output of the SUT.

Lutin is a language to program stochastic reactive systems. It has been designed to model input constraints and perform automated testing of reactive systems with Lurette. Lutin supports the numerical and Boolean types of Lustre, as well as all Lustre statements, such as variable assignment, conditional (**if-then-else**), previous (**pre**) and initialization operators (−>). In addition, Lutin has three specific operators: sequence (**fby**, for "followed by"), Kleene star (**loop**), and probabilistic choice (|). Consider for example the following Lutin node corresponding to the simple scenario defined by the automaton[1] in Figure 2.4, where the input values `actual_room_temperature` and `season` of the node `heater` (see Section 2.2.1) can possibly take 18 and Winter respectively, or 26 and Spring, etc. Note that the first `actual_room_temperature` input is only taken into account in the first activation of the heater, then for the next iterations only the `season` input values are taken into account. Because Lutin does not provide structured datatypes, we have to add four constants for the four seasons.

```
node input_constraints () returns (actual_room_temperature, season: int) =
  let Winter:int = 1, Spring:int = 2, Summer:int = 3, Autumn:int = 4 in
  loop {
    | actual_room_temperature = 18 and season = Winter
    | actual_room_temperature = 26 and season = Spring
    | actual_room_temperature = 34 and season = Summer
    | actual_room_temperature = 20 and season = Autumn
  }
```

At each iteration, Lurette generates inputs by executing one transition of the Lutin node.

[1]This automaton is not an LTS.

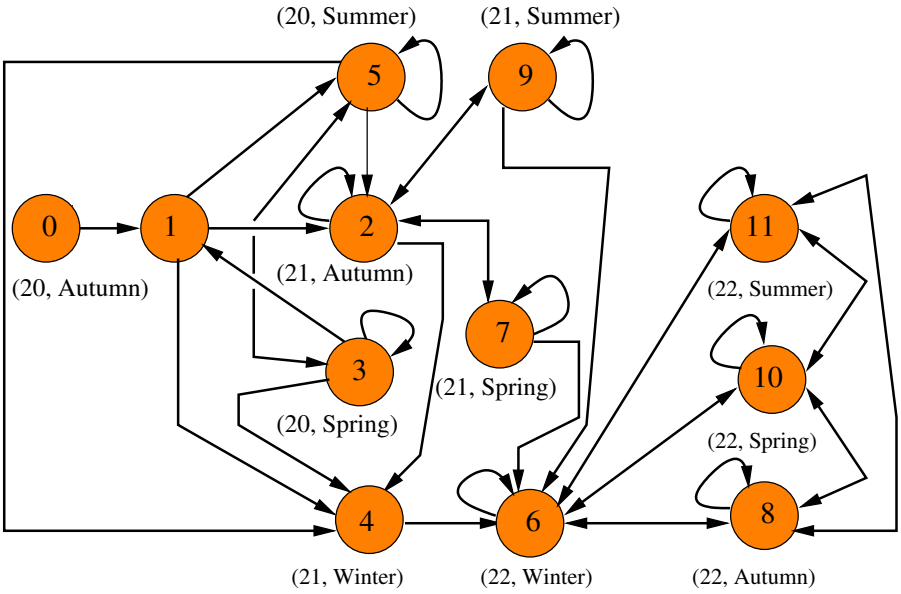


Figure 2.5: Part of the oracle automaton with output `new_room_temperature` and input `season`, corresponding to an initial value of 20 for `actual_room_temperature`

Each input can only be a Boolean, or a numerical type (integer, real), mostly because of the data types provided by the Lutin language. Note that if the input values are not explicitly constrained in the Lutin specification, random numbers will be generated.

Lustre oracle

In order to determine whether the outputs generated by the SUT are correct or not, one should define an oracle. An oracle should contain all possible pairs of inputs with the corresponding expected outputs, and signal an error for each unexpected output. In addition to checking correctness, Lurette supports additional Boolean outputs (called coverage variables) to measure coverage. While testing the SUT, Lurette records the coverage variables that were at least once true, and computes the ratio of covered versus uncovered variables. The list of coverage variables with their status is stored in a file, and is updated by any subsequent run of Lurette for the same SUT, input constraints, and oracle.

For instance, the temperature of the heater (`new_room_temperature`) can only increase, and only if the previous room temperature (**pre** `new_room_temperature`) is lower than the expected temperature (`expected_temperature`), defined for each season. Because the heater takes into account only the first actual room temperature (`actual_room_temperature`), the possible values of the heater (`new_room_temperature`) depend on the value of the first actual room temperature and on the next room temperature (`new_room_temperature`). Consider for example all possible combinations of the input value (`season`) and the output value (`new_room_temperature`) of the node `heater`: if we start with an actual temperature ini-

tialized with 20 in the automaton[2] illustrated in Figure 2.5, then according to the season, all possible temperatures of the heater can be explored. The oracle should take into account all these possible values, and also for all other possible initializations. The following Lustre code corresponds to an excerpt of the oracle from the automaton illustrated in Figure 2.5. As the Lutin node `input_constraints`, the oracle takes the inputs and outputs of the heater (`season, temperature`) as input and gives as outputs the verdict `res`, i.e., whether the observed outputs are those expected from the inputs. For instance, if the first `actual_room_temperature` is 20, the temperature can only increase until 22, according to the seasons and the previous temperature of the heater. The following oracle also computes a coverage variable `cover_seasons`, measuring the coverage of all seasons at least once by the generated input vectors.

```
const Winter = 1;
const Spring = 2;
const Summer = 3;
const Autumn = 4;

node oracle (season, temperature: int)
returns (res, cover_seasons: bool);
var s1, s2, s3, s4: bool;
let
  res = true ->
    (* if the first actual_room_temperature is 20 *)
        ((season = Winter and temperature = 21) or
         (season = Winter and temperature = 22)
     or (season = Spring and temperature = 20) or
        (season = Spring and temperature = 21) or
        (season = Spring and temperature = 22)
     or (season = Summer and temperature = 20) or
        (season = Summer and temperature = 21) or
        (season = Summer and temperature = 22)
     or (season = Autumn and temperature = 20) or
        (season = Autumn and temperature = 21) or
        (season = Autumn and temperature = 22)
     (* the cases for the other first actual_room_temperature values *)
     or ...);

  (* each intermediate variables s1 - s4 are covering one season *)
  s1 = false -> if season = Winter then true else pre s1;
  s2 = false -> if season = Spring then true else pre s1;
  s3 = false -> if season = Summer then true else pre s1;
  s4 = false -> if season = Autumn then true else pre s1;

  (* check the coverage of each season at least once *)
```

[2]This automaton is not an LTS.

```
  cover_seasons = false ->
    if s1 and s2 and s3 and s4 then true else pre cover_seasons;
tel
```

For a more detailed presentation of Lurette see [JRB06]. In the next section, we describe asynchronous model and validation for the asynchronous communication between synchronous components.

2.3 Asynchronous Models and their Validation

Process calculi [AFV01] provide a formal means for the high-level description of interactions, communications, and synchronizations between a set of independent components of a concurrent system. They also provide rules to describe, manipulate, and analyze processes. The interactions between independent processes are represented by message-passing on communication gates (not modification of shared variables). Interleaving semantics is used for the compositions of processes, i.e., if two processes P_1 and P_2 execute respectively an action a_1 and a_2 in parallel, then the global behavior, represented by an LTS, will have two possible paths: a_1 then a_2 and a_2 then a_1. In the asynchronous semantics two actions never occur at the same time.

The ideas behind process algebra have been implemented in several languages including: CCS [Mil89], CSP [Hoa85], ACP [BK85], LOTOS [BB88], mCRL2 [GMR+06], and LNT [CCG+19]; and tools including: CADP [GLMS13], Concurrency Workbench [CPS89, CPS93], FDR [GABR14], and mCRL2 [CGK+13].

In this thesis, we use LNT as specification language for obtaining LTSs models, mostly because LNT is concise, expressive, easily readable, and user-friendly. Also LNT is the recommended specification language of the CADP toolbox [GLMS13], which provides several useful validation tools.

2.3.1 Overview of the LNT language

LNT is an improved version of formal specification language LOTOS [BB88], providing more user-friendly notations than LOTOS, which combines traits from process calculi, functional languages, and imperative languages. In order to let the reader understand the LNT excerpts presented in Chapters 4 and 5, we present here a short introduction to the LNT language. The reader familiar with LNT can safely skip this section.

LNT allows the definition of data types, functions, and processes.

LNT provides some predefined types, i.e., Boolean (**bool**), natural numbers (**nat**), integer numbers (**int**), real numbers (**real**), characters (**char**) and strings (**string**). The user can

also declare abstract data types, through the use of the **type** keyword. For instance, we define the enumeration type `drink`, corresponding to the different types of drinks of a vending machine:

```
type drink is
  coffee, tea, water, milk
end type
```

It is also possible to define record types, which have a single constructor regrouping several types. For instance, the following definition of the type `prices`:

```
type price is
  price (class: drink, cost: real)
  with ''get'', ''set''
end type
```

The predefined functions **get** and **set** enable to access and respectively modify one field of the record type `price`. For instance, the following LNT instructions of the `ristretto` price illustrate two different ways to access and modify a field of this type:

```
ristretto := price (coffee, 20);
-- get the cost of the ristretto
get_cost (ristretto);
ristretto.cost;
-- modify the cost of the ristretto
set_cost (ristretto, 25);
ristretto.{cost => 2}
```

Note that comments in LNT can be single line comments, in the form *−−comment*, or several lines, in the form *(∗comment ∗)*.

LNT provides syntactic sugar to define types corresponding to the **list**, **sorted list**, **set**, or **array** of another type. We can for instance have the following LNT list type `coin`, corresponding to all possible coin values (from 5 cents to 2 euros), and the type `coins` corresponding to a list of coins:

```
type coin is
  5, 10, 20, 50, 1, 2
end type

type coins is
  list of coin
end type
```

The latter is syntactic sugar for:

```
type coins is
cons (head: coin, tail: coins)
end type
```

Similarly, `{}` is converted to nil, and `{0}` is converted to `cons (0, nil)`.

Functions define operations on types. The predefined data types are already provided with predefined functions. Functions are defined with the **function** keyword. The passing of parameters to a function can be done by copy (parameter **in**), or by reference (parameter **out** or **in out**). By default, a parameter is passed by copy. A function can return a value of a certain type. Recursive calls are allowed. Control structures, such as conditional branching or loops, are available. Various statements can be used inside a function, for instance correct termination (**null**), sequential composition (**;**), function return (**return**), variable declaration (**var**), conditional construct (**if**), breakable loop construct (**loop**), for loop construct (**for**). Value assignments to variables are performed with the `:=` operator. For instance, the LNT function `n_coins` takes as input a number `n` and a coin `value`, and computes a list of `n` coins of this `value`:

```
function n_coins (n: int, value: coin) : coins is
  var cash: coins in
    cash := {};
    loop L
      if n <= 0 then break L
      else
        cash := cons (value, cash);
        n := n - 1
      end if
    end loop;
    return cash
  end var
end function
```

The **case** operator allows pattern matching to be performed on values of any type. As part of the pattern matching, the keyword **any** matches any value. The following function returns the change if the cash passed as an argument is bigger than the price of the drink, all coins of the cash are matched in order to compute the sum:

```
function give_change (cost: real, cash: coins) : real is
  var sum: real in
    sum := 0;
    loop L in
      if cash == {} then break L
      else
        case head(cash) in
          5 -> sum := sum + 0.05
        | 10 -> sum := sum + 0.10
        | 20 -> sum := sum + 0.20
        | 50 -> sum := sum + 0.50
        | 1 -> sum := sum + 1
        | 2 -> sum := sum + 2
```

```
        end case;
        cash := tail(cash)
      end if
    end loop;
    return sum - cost
  end var
end function
```

Note that LNT pattern matching should be exhaustive; the existence of a formal semantics is guaranteed by static semantic constraints [Gar15] ensuring that all possible cases are taken into account in the pattern matching.

A process executes actions. An action is either internal or a rendezvous synchronization on a *gate*. The internal action in LNT is designated by `i`. External actions, i.e., realized on a gate, are the observable events of the execution of a process. The formal semantics of LNT defines how, from the specification of a process, one can build the LTS that represents the possible behaviors of this process. An action on a gate is defined by the identifier of the gate, possibly followed by a list of offers in parentheses. A gate is used to handle input/output communications and synchronizations. Offers allow to exchange data on a gate, and they can be an emission of an expression prefixed by an optional `!`, or the reception of a value prefixed by `?`. Here is for example a behavior that performs an internal action, followed by an action on the gate `cup` where the first offer denotes the sending of a coin, and the second one the receiving a drink.

```
var d: drink in
  i;
  cup (10, ?d);
end var
```

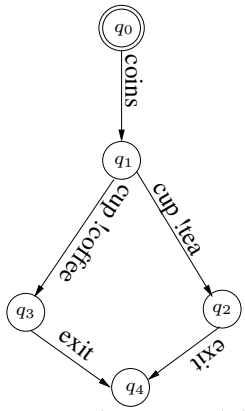
Gates can be untyped (**any**) or typed by a channel. The channel defines the gate profiles, i.e., the possible offer combinations accepted by the gate. A channel is defined using the **channel** keyword and must state one or more profiles, which are lists of parameters that declare the number and types of authorized offers. Note that two gates are compatible only if they have the same type, or they are both untyped (**any**). The channel `none` is predefined and allows any profile offers. For instance, the following LNT channels define one offer with the drink type and one offer with two possible types, respectively.

```
-- one offer with the drink type is allowed
channel single is (drink) end channel
-- several profiles allowed
channel change is
  (coins),
  (token)
end channel
```

Processes in LNT are defined with the **process** keyword. A process receives as argument a list of gates on which it can perform actions. Every gate corresponds to a channel. In order to make explicit the gates available for communication and synchronization between processes, each processes has a list of gates parameters (between square brackets). Similar to a function, a process can also receive a list of values as arguments (between parentheses). Statements presented for functions also apply to processes. A process terminates by executing implicitly the special action `exit`.

A process can make nondeterministic choices between multiple actions using the **select** operator. This operator allows to introduce several possible behaviors, which are delimited by []. For instance, the following LNT process `vending_machine` and its corresponding LTS make a nondeterministic choice between the action on the gate `cup (coffee)` and the one on the gate `cup (tea)`. Once it has realized an action of a branch of this non-deterministic choice, it is forced to perform the rest of this branch.

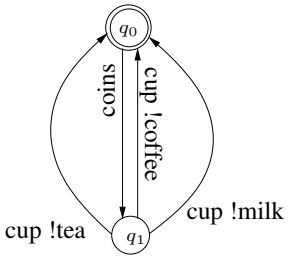
```
process vending_machine [coins: none, cup: drink] is
  coins;
  -- nondeterministic choice (tea or coffee)
  select
     cup (coffee)
   []
     cup (tea)
  end select
end process
```



Note that this corresponds to a state in the LTS with multiple outgoing transitions, where each transition corresponds to one of the choices in a **select** construct.

Another construction can also perform nondeterminism in an expression. The keyword **any** assigns any value of a type, possibly with a restriction introduced by the clause **where**. For example, the following LNT process (and its corresponding LTS) can cyclically perform an action on the gate `cup` passing a random type of drink different from water:

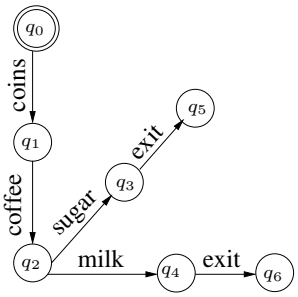
```
process vending_machine [coins: none, cup: drink] is
  var kind: drink in
    loop
      coins;
      -- nondeterministic choice of drink
      kind := any drink where (kind != water);
      cup (kind)
    end loop
  end var
end process
```



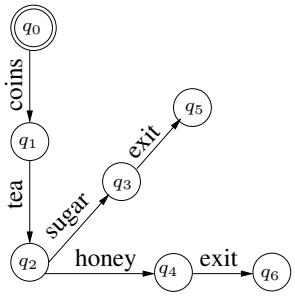
The parallel composition operator can be used to handle communication between differ-

ent processes. Concretely, the parallel composing operator **par** is used to declare multiple processes that run independently, in parallel, while specifying on which gates the processes must synchronize their actions. Two or more processes synchronize and exchange information during a *rendezvous*. A rendezvous involving an arbitrary number superior to two processes is called *multiway rendezvous*. One can distinguish three kinds of treatments on actions during a parallel composition. The actions on the gates for which no synchronization is imposed are performed independently by each process. The termination action (`exit`) is always synchronized between all processes in a parallel composition. Only actions which are offered by every process in the parallel composition can be synchronized. The following example shows the parallel composition of the `tea_m` and `coffee_m` processes, which must synchronize their actions on gates `coins`, `sugar`, `milk` and `honey`:

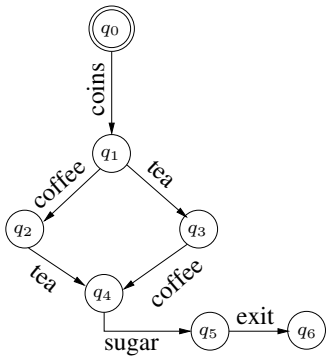
```
-- coffee manager
process coffee_m [coins, coffee,
                  sugar, milk: none] is
  coins;
  coffee;
  select
    sugar
  []
    milk
  end select
end process
```




```
-- tea manager
process tea_m [coins, tea,
               sugar, honey: none] is
  coins;
  tea;
  select
    sugar
  []
    honey
  end select
end process
```



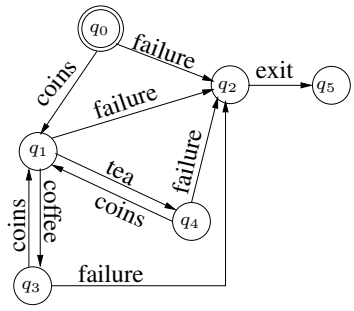
```
-- parallel composition
par coins, sugar, milk, honey in
   coffee_m [coins, cofee, sugar, milk]
|| tea_m [coins, tea, sugar, honey]
end par
```



The first action on gate `coins` is common to both processes. Then the actions on gates `coffee` and `tea` do not have to be synchronized, so they can be realized in any order. Actions on gates `sugar`, `milk`, and `honey` must be synchronized. When the process `coffee_m` is ready for an action on the gate `sugar` or the gate `milk`, and the process `tea_m` is ready for action on gate `sugar` or gate `honey`, so the only action that is possible for both processes is the one on the `sugar` gate. Finally, the two processes synchronize their `exit` action.

Last, but not least, the disruption operator is also provided by LNT. The **disrupt** behavior starts a behavior, however at any moment, this behavior can be interrupted by another one. For instance, the following LNT disrupt behavior starts with the correct behavior of the `vending_machine` (`coins`, etc.), which executes normally. However, at any moment, the correct behavior can be interrupted, in which case the execution of the unexpected behavior (an electrical `failure`) starts and the correct behavior is disrupted.

```
process vending_machine [coins, failure: none, cup: drink] is
  disrupt
    -- correct behavior
    loop
      coins;
      select
        cup (coffee)
      []
        cup (tea)
      end select
    end loop
  by
    -- unexpected behavior
    failure
  end disrupt
end process
```



Note, if the correct behavior successfully terminates before any action has taken place in the unexpected behavior, the disrupt behavior terminates, meaning that the possibility to be interrupted by an electrical `failure` disappears.

For a more detailed presentation of LNT see [CCG+19, GLS17].

2.3.2 Verification of asynchronous systems

In this section, we present model checking tools enabling the verification of asynchronous systems, more particularly the property specification languages of the model checker used in this thesis. Numerous model checkers exist for asynchronous systems, including: CADP [GLMS13], CPAchecker [BK11], DREAM [PCdVA12], FDR [GABR14], Java Pathfinder [VHBP00], mCRL2 [CGK+13], Prism [HKNP06], SPIN [Hol97], TLA+ [Lam93], and UPPAAL [BLL+95].

In this thesis we will use the EVALUATOR 4.0 [MT08] model checker of CADP. EVALUATOR 4.0 can verify temporal logic properties on the fly on an LTS, and exhibits full diagnostics (portions of the LTS) illustrating the truth value of temporal logic formulas. The temporal logic formulas are written in MCL [MT08]. We also used the executable temporal Logic tool called XTL [MG98] of the CADP toolbox. We included the MCL formulas and the XTL calls in an SVL (Script Verification Language) [GL01] script, in order to automate the model verification and test generation method presented in Chapter 5. This script avoids the need to explicitly name certain CADP tools, as SVL is responsible for invoking the appropriate tools to conduct the required verifications. The SVL tool executes verification scenarios expressed in the scripting language SVL. CADP tools are traditionally called from the command line, and each tool has several options. The SVL language makes it easy to describe verification scenarios which involve several CADP tools. In addition, it is possible to insert any UNIX shell command within an SVL scenario, which facilitates the automation of formal verifications.

Overview of the MCL language

In this section we give an overview of the MCL language, readers familiar with MCL can safely skip this section. MCL (Model Checking Language) [MT08] is an action-based, branching-time temporal logic suitable for expressing properties of concurrent systems, including properties parameterized by data values. MCL is interpreted on Labelled Transition Systems (LTSs).

MCL language extends the temporal property specification formalism μ-calculus [Koz83], with regular expressions over transition sequences similar to those of PDL [FL79], data-handling constructs inspired from functional programming languages (quantified variables and fixed point parameters), and a generalization of the infinite looping operator of PDL-Δ [Str82] specifying fairness. MCL allows a concise and readable formulation of temporal properties, notably when these properties are parameterized by data values.

We give below the MCL formulation of the four temporal properties defined in Section 2.1.3 and one additional MCL properties, in order to illustrate the MCL predicates used in this thesis.

a) The following MCL formula encodes a liveness property expressing the existence of a

sequence leading to an action, e.g., one can potentially get a `TEA` after inserting a `COIN` in the vending machine:

< **true*** . `COIN` . **true***. `TEA` > **true**

b) The following MCL formula encodes a liveness property expressing inevitability using fixed point operators. It states that all transition sequences starting at the current state lead to `TERMINATE` actions after a finite number of steps, e.g., the vending machine system terminates on a `COFFEE` or `TEA` transition:

mu `X` . (< **true** > **true and** [**not** `TERMINATE`] `X`)

where `TERMINATE` is an MCL macro denoting the termination actions (`COFFEE` or `TEA`):

macro `TERMINATE () = ( COFFEE` **or** `TEA )` **end_macro**

c) The following MCL formula encodes a fairness property, that expresses reachability of actions by considering only fair execution sequences, e.g., after every `COIN` inserted, all fair execution sequences (i.e., by skipping cycles) will lead to the reception of a `TERMINATE` action:

[**true*** . `COIN` . (**not** `TERMINATE`)*] < **true*** . `TERMINATE` > **true**

d) The following MCL formula encodes a safety property, which expresses that something bad never happens, e.g., if the action `COIN` occurs, then while there is no action `TEA` or `COFFEE`, no other action `COIN` may happen:

[**true***. `COIN` . (**not** (`COFFEE` **or** `TEA`))*. `COIN`] **false**

e) It is also possible to use action predicates in the MCL formulas, for instance the following MCL formula encodes a safety property, which expresses that a kid should not get a caffeine drink, denoted by the predicate `caffeine(drink)`:

[**true** *. `KID` . {`CUP` ?`drink`:**String where not** (`caffeine(drink)`)}] **false**

For a more detailed presentation of MCL see [MT08].

Overview of the XTL language

In this section we give an overview of the XTL language, readers familiar with XTL can safely skip this section.

The XTL (eXecutable Temporal Logic) tool [MG98] is a tool that allows the parsing of BCG (an explicit representation of LTS) [Gar91], and can perform operations at a lower level than classical temporal logic. Since XTL is a programming language, it can also be used to define non-standard temporal operators and, more generally, to perform any

computation on an LTS model. In this thesis, we used it to traverse the LTS and print various information (e.g., list of labels going out of certain states).

XTL provides the standard predefined types with their usual operators, such as **boolean**, **integer**, **character**, **string**, etc. XTL provides also the types corresponding to the elements of a LTS model, such as the states and sets thereof (**state** and **stateset**), the transitions and sets thereof (**edge** and **edgeset**), the labels and sets thereof (**label** and **labelset**).

XTL expressions are built from classical functional programming operators, such as function calls, or pattern-matching. For example, the following XTL code computes the set of labels corresponding to the reception of a cup containing a drink different than coffee. The variable `D` is initialized by pattern-matching with the corresponding value contained in the label, and is used in the **where** clause, which allows additional filtering using a Boolean condition:

```
{ A:label where A -> [ CUP ?D:String where D <> "COFFEE" ] }
```

In addition, XTL provides four specific operators for accessing and exploring the LTS and its elements:

- the **init** operator returns the initial state of the LTS;
- the **succ** and **pred** operators access successors and predecessors of a state respectively;
- the **in** and **out** operators define incoming and outgoing transitions of a state respectively;
- the **source** and **target** operators access the origin and the destination states of a transition respectively.

An XTL expression can also contain quantifiers (**exists**) and conditional branching (**if**) or loop (**for**) control structures. For binary operators, such as +, `*`, `<=`, both prefix and infix notations are available. For instance, the definition of a *corner state*, i.e., the state corresponding to the target of an input transition (`COIN`) and the source of an output transition (`CUP`), can be implemented by the following XTL function:

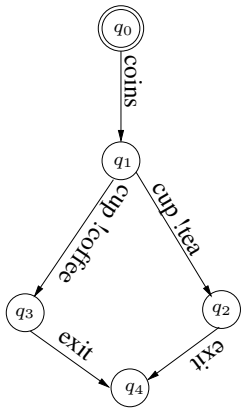
```
def corner_state (s: state): boolean =
  exists e:edge among in (s) in e -> [COIN ...] end_exists and
  exists e:edge among out (s) in e -> [CUP ...] end_exists
end_def
```

XTL being a functional language, its loop control structures operate by performing iterative computations over accumulation variables. For example, the following XTL loop **for** prints all corner states of an LTS: The evaluation of the **for** proceeds as follows. First, the accumulation variables, more precisely the state `s` and an **action**, are initialized with the initial state (**init**) and the empty action (**nop**) respectively. Then, for all corner states, an iteration is performed, that consist in assigning the state name to the accumulator (`s`) and printing the information through the second accumulator (**fby**).

```
for s:state where corner_state (s)
  apply (replace, fby)
  from (init, nop)
  to (s,
  printf ("the state ") fby print (s) fby
  printf (" is a corner state\n"))
end_for
```

For instance, the XTL loop execution on the following small LTS prints:

```
the state 1 is a corner state
```



For certain **for** loops, such as the previous one, XTL also provides an abbreviated form:

```
<| fby on s:state where corner_state (s) |> (
  printf ("the state ") fby
  print (s) fby
  printf (" is a corner state\n")
)
```

As in MCL, one can also define macros in XTL, for instance the following example of XTL macro definition implements the `drink_transition`, characterizing all edges labeled by a `drink` (`coffee` or `tea`) action:

```
macro drink_transition (e) =
  (((e) -> [TEA]) or ((e) -> [COFFEE]))
end_macro
```

For a more detailed presentation of XTL see [Mat98, MG98].

2.3.3 Asynchronous model-based testing (the ioco theory)

After verifying the formal model of a system (e.g., using model checking), one can check the conformance of the system under test (SUT) against this formal model. This can be carried out by Model-Based Testing (MBT), which generates tests from the model. The

purpose of such test suites is either to check if the SUT is correct, or if the model does not correspond to the SUT.

Conformance testing aims at extracting from a formal model (M) of a system a set of test cases to assess whether an actual implementation of the SUT conforms to the model, using black-box testing techniques (i.e., without knowledge of the actual code of the SUT). This technique is particularly suited for nondeterministic concurrent systems, where the behavior of the SUT can be observed and controlled by a tester only via dedicated interfaces, called points of control and observation. Often, the formal model is an IOLTS[3] (Input/Output Labelled Transition System), where transitions between states of the system are labeled with an action classified as input, output, or internal (i.e., unobservable, usually denoted by τ). In this setting, the most prominent conformance relation is input-output conformance (**ioco**) [dT01, Tre08]. In this section, we present the **ioco** relation and a few tools that use variants of the **ioco** conformance relation. Other model-based tools for combinatorial and statistical testing, or white box testing are described in [UPL12].

According to this relation, the SUT conforms to the model if and only if the SUT passes all tests generated from the model. In our context, we are interested in MBT using LTSs and the input/output conformance (**ioco**) relation proposed by the conformance testing theory. The SUT behaves as an input-enabled LTS (i.e., IOTS), from all states every input is possible. Intuitively, a SUT (with initial state s) **ioco**-conforms to a model (with initial state m), if and only if:

- when s produces an output $!x$ after a trace σ, m can also produce the output $!x$ after the trace σ;
- and when s cannot produce any output after a trace σ, m cannot produce either any output after this trace σ, which denotes the quiescence δ.

Formally, **ioco** relates an SUT $\in IOTS$ and a model $M \in IOLTS$:

$$\mathbf{ioco} \subseteq IOTS(A_i, A_o) \times LTS(A_i \cup A_o)$$

The **ioco** relation is defined in terms of observing outputs as follows:

$$SUT\ \mathbf{ioco}\ M =_{def} \forall \sigma \in \mathit{Straces(m)}\colon\ \mathit{out(s\ after\ \sigma)} \subseteq \mathit{out(m\ after\ \sigma)}$$

where:

- $p \xrightarrow{\delta} p = \forall !x \in A_o \cup \{\tau\}.p \not\xrightarrow{!x}$
- $\mathit{Straces(m)} = \{\sigma \in (A \cup \{\delta\})^* | m \overset{\sigma}{\Rightarrow}\}$
- $p\ \mathit{after}\ \sigma = \{p' | p \overset{\sigma}{\Rightarrow} p'\}$
- $\mathit{out}(P) = \{!x \in A_o | p \xrightarrow{!x}, p \in P\} \cup \{\delta | p \xrightarrow{\delta} p, p \in P\}$, where P is a set of state.

[3]More details about IOLTS are available in Section 2.1.1

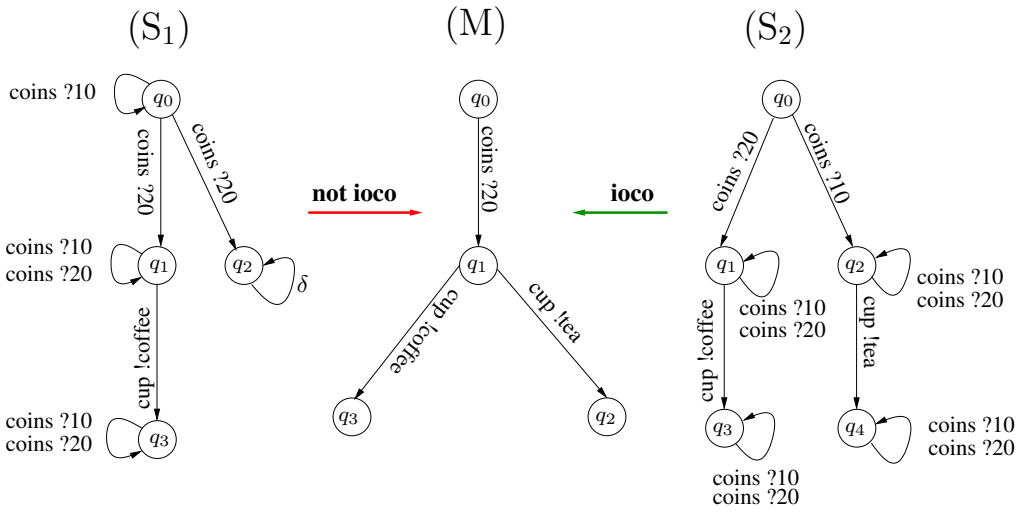


Figure 2.6: Examples of the implementation relation **ioco**.

Essential properties of test suites are established for the ioco relation, i.e., *soundness* and *exhaustiveness* [Tre96]. The **ioco** relation is *sound*, i.e., if the SUT **ioco** M then the SUT passes the generated tests. The **ioco** relation is *exhaustive*, i.e., if the SUT passes all the generated tests then the SUT **ioco** the IOLTS model. Although in practice, it may be infeasible to generate all the tests (the test suite necessary for exhaustiveness may even be infinite).

A test suite is *valid* if this test suite detects the non conformity:

$$s \; not \; \mathbf{ioco} \; m \implies \exists t : \; s \; fails \; t$$

and if the test suite never fails with a correct SUT:

$$s \; \mathbf{ioco} \; m \implies \forall t, s \; passes \; t$$

where s represents the system under test, m the model, and t a test of the test suite.

For instance, in the examples of the implementation relation **ioco** shown on Figure 2.6 (inspired by [Tre08, Figure 8]), the model M specifies that after input *coins ?20* output *cup !coffee* must occur, which is expressed as: *out(M* ***after*** *coins ?20)* $= \{$*cup !coffee*$\}$. The system under test S_2 satisfies this requirement, but S_1 does not: *out (*S_1 ***after*** *coins ?20)* $= \{$*cup !coffee, δ*$\} \not\subseteq \{$*cup !coffee*$\}$. Effectively S_1 is not **ioco** M, because the system under test should not produce more outputs than allowed by the model. Note that S_2 **ioco** M, even if *out (*S_2 ***after*** *coins ?10)* $= \{$*cup !tea*$\}$ because *coins ?10* $\notin$ *Straces(M)*.

Although model-based conformance testing has been intensively studied, there are only a few tools that use variants of the **ioco** conformance relation and that are still actively developed [BFS05]. In this section we will compare the main ones: AGEDIS [HN04], JTorX [Bel10, Bel14], STG [CJRZ02], TGV [JJ05], TorX [TB03], TorXakis [MPS+09, Tre17], Uppaal-Cover [HP06], Uppaal-Tron [LMNS05], and Uppaal-Yggdrasil [KLN+15].

Table 2.2 compares these **ioco** conformance tools by taking into account the following six tool features: (1) online conformance testing (i.e., a simultaneous execution of the model and the SUT or/and offline); (2) a test selection to generate relevant tests; (3) ont-the-fly tests generation; (4) handling nondeterministic models; (5) handling timed models; and (6) handling infinite models.

Table 2.2: Comparison of **ioco** model-based testing tools

tool	online/ offline	coverage criteria	test purpose	on-the-fly	nondeter.	timed LTS
AGEDIS	offline	yes	yes	yes	yes	no
JTorX	both	yes	no	yes	yes	no
STG	offline	no	yes	no	yes	no
TGV	offline	no	yes	yes	yes	no
TorX	both	no	no	yes	yes	no
TorXakis	online	no	yes	yes	yes	no
Uppaal-Cover	both	yes	no	yes	no	yes
Uppaal-Tron	both	no	no	yes	yes	yes
Uppaal-Yggdrasil	both	no	yes	yes	no	yes

Note that TGV and AGEDIS tools perform on-the-fly only the extraction of complete test graphs, but not the extraction of controllable test cases.

In this thesis, in order to face the state-space explosion of the complex GALS model we needed an on-the-fly model-based testing tool with test selection, the test selection description could contain data, so we decided to extend the TGV approach.

The TGV approach

In this section we give an overview of the TGV approach, readers familiar with the TGV approach can safely skip this section.

We assume that the behavior of the SUT can also be represented as an IOLTS, even if it is unknown (the so-called testing hypothesis [JJ05]). In the sequel, we consider the same running example as [JJ05], whose IOLTS model M is shown on Figure 2.7.

Input actions of the SUT are controllable by the environment, whereas output actions are only observable. Testing allows one to observe the execution traces of the SUT, and also to detect *quiescence*, i.e., the presence of deadlocks (states without successors), outputlocks (states without outgoing output actions), or livelocks (cycles of internal actions). The quiescence present in an IOLTS L (either the model M or the SUT) is modeled by a *suspension automaton* $\Delta(\mathrm{L})$, an IOLTS obtained from L by adding self-loops labeled by a special output action δ on the quiescent states. The SUT conforms to the model M

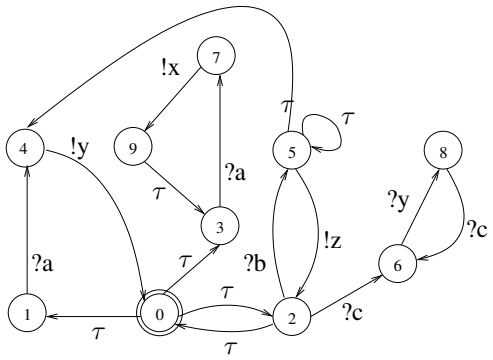


Figure 2.7: Model M of the running example [JJ05]

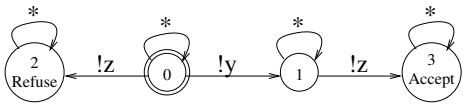


Figure 2.8: Test purpose TP of the running example [JJ05]

modulo the **ioco** relation [Tre96] if after executing each trace of $\Delta(\mathrm{M})$, the suspension automaton $\Delta(\mathrm{SUT})$ exhibits only those outputs and quiescences that are allowed by the model. Since two sequences having the same observable actions (including quiescence) cannot be distinguished, the suspension automaton $\Delta(\mathrm{M})$ must be determinized before generating tests.

The test generation technique of TGV is based upon *test purposes*, which allow one to guide the selection of test cases. A test purpose guides the exploration of the specification model to have more focused test cases. Formally, a test purpose for a model $\mathrm{M} = (Q^{\mathrm{M}}, A^{\mathrm{M}}, T^{\mathrm{M}}, q_0^{\mathrm{M}})$ is a deterministic and complete (i.e., in each state all actions are accepted) IOLTS $\mathrm{TP} = (Q^{\mathrm{TP}}, A^{\mathrm{TP}}, T^{\mathrm{TP}}, q_0^{\mathrm{TP}})$, with the same actions as the model $A^{\mathrm{TP}} = A^{\mathrm{M}}$. TP is equipped with two sets of trap states $Accept^{\mathrm{TP}}$ and $Refuse^{\mathrm{TP}}$, which are used to select desired behaviors and to cut the exploration of M, respectively. In the TP shown on Figure 2.8, the desired behavior consists of an action !y followed by !z and is specified by the accepting state q_3; notice that the occurrence of an action !z before a !y is forbidden by the refusal state q_2.

In a TP, a special transition of the form $q \xrightarrow{*} q'$ is an abbreviation for the complement set of all other outgoing transitions of q. These $*$-transitions facilitate the definition of a test purpose (which has to be a complete IOLTS) by avoiding the need to explicitly enumerate all possible actions for all states. Test purposes are used to mark the accepting and refusal states in the IOLTS of the model M. In TGV, this annotation is computed by a synchronous product [JJ05, Definition 8] $\mathrm{SP} = \mathrm{M} \times \mathrm{TP}$. Notice that SP preserves all behaviors of the model M because TP is complete and the synchronous product takes into account the special $*$-transitions. When computing SP, TGV implicitly adds a self-looping $*$-transition to each state of the TP with an incomplete set of outgoing transitions. To

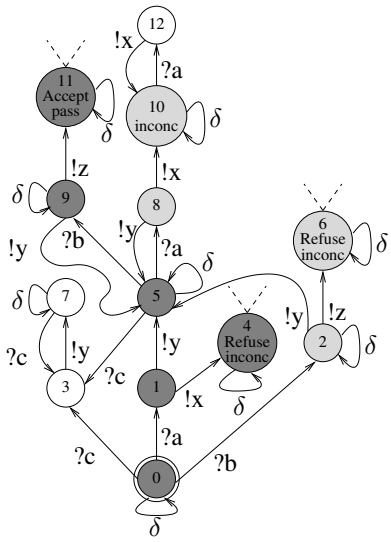


Figure 2.9: The visible behavior SP_{vis}, complete test graph CTG (gray), and a test case TC (dark gray) of the running example [JJ05]

keep only the visible behaviors and quiescence, SP is suspended and determinized, leading to $\mathrm{SP}_{vis} = det(\Delta(\mathrm{SP}))$. Figure 2.9 shows an excerpt of SP_{vis} limited to the first accepting and refusal states reachable from $q_0^{\mathrm{SP}_{vis}}$.

A *test case* is an IOLTS $\mathrm{TC} = (Q^{\mathrm{TC}}, A^{\mathrm{TC}}, T^{\mathrm{TC}}, q_0^{\mathrm{TC}})$ equipped with three sets of trap states $\mathbf{Pass} \cup \mathbf{Fail} \cup \mathbf{Inconc} \subseteq Q^{\mathrm{TC}}$ denoting verdicts. The actions of TC are partitioned into $A_{\mathrm{I}}^{\mathrm{TC}}$ and $A_{\mathrm{O}}^{\mathrm{TC}}$ subsets[4]. A test case TC must be *controllable*, meaning that in every state, no choice is allowed between two inputs or an input and an output (i.e., the tester must either inject a single input to the SUT, or accept all the outputs of the SUT). Intuitively, a TC denotes a set of traces containing visible actions and quiescence that should be executable by the SUT to assess its conformance with the model M and a test purpose TP. From every state of the TC, a verdict must be reachable: **Pass** indicates that TP has been fulfilled, **Fail** indicates that SUT does not conform to M, and **Inconc** indicates that correct behavior has been observed but TP cannot be fulfilled. An example of TC (dark gray states) is shown on Figure 2.9. Pass verdicts correspond to accepting states (e.g., q_{11}). Inconclusive verdicts correspond either to refusal states (e.g., q_4 or q_6) or to states from which no accepting state is reachable (e.g., state q_{10}). Fail verdicts, not displayed on the figure, are reached from every state when the SUT exhibits an output action (or a quiescence) not specified in the TC (e.g., an action !z or a quiescence in state q_1).

In general, there are several test cases that can be generated from a given model and test purpose. The union of these test cases forms the Complete Test Graph (CTG), which is

[4]In TGV [JJ05], the actions of test cases are symmetric w.r.t. those of the model M and the SUT, i.e., $A_{\mathrm{O}}^{\mathrm{TC}} \subseteq A_{\mathrm{I}}^{M}$ (TC emits only inputs of M) and $A_{\mathrm{I}}^{\mathrm{TC}} \subseteq A_{\mathrm{O}}^{\mathrm{SUT}} \cup \{\delta\}$ (TC captures outputs and quiescences of SUT). To avoid confusion, we consider here that inputs and outputs of TC are the same as those of M and SUT.

an IOLTS having the same characteristics as a TC except for controllability. Figure 2.9 shows the CTG (light and dark gray states) corresponding to M and TP, which is not controllable (e.g., in state q_5 the two input actions ?a and ?b are possible). Formally, a CTG is the subgraph of SP_{vis} induced by the states L2A (*lead to accept*) from which an accepting state is reachable, decorated with pass and inconclusive verdicts. A controllable TC exists iff the CTG is not empty, i.e., $q_0^{\mathrm{SP}_{vis}} \in$ L2A [JJ05].

The execution of a TC against the SUT corresponds to a parallel composition TC || SUT with synchronization on common observable actions, verdicts being determined by the trap states reached by a maximal trace of TC || SUT, i.e., a trace leading to a verdict state. Quiescent livelock states (infinite sequences of internal actions in the SUT) are detected using timers, and lead to inconclusive verdicts. A TC may have cycles, in which case global timers are required to prevent infinite test executions. For a more detailed presentation of TGV see [JJ05].

In this thesis, we will present a new tool for on-the-fly conformance test case generation tool based on the TGV approach, but enabling online testing by generating controllable test cases completely on the fly and a more versatile description of test purposes than the TGV ones using the LNT language.

2.4 GALS Models and Validation

A GALS (Globally Asynchronous and Locally Synchronous) system consists of several independent synchronous components that evolve concurrently, each at its own pace, and communicate all together asynchronously. A GALS system is a particular asynchronous system, but with additional difficulty when faithfully modelling it as a composition of synchronous components, with all the synchronous constraints (one clock, no interleaving during a component execution, synchronous loop). To model it formally and manage the complexity of the underlying state space (and hence of the analysis), one has to consider the following aspects:

C1. atomicity of synchronous components, i.e., the execution of the synchronous component can not be interrupted or reduced before its end;

C2. interleaving of the execution of synchronous components, i.e., the order of synchronous component executions is not fixed;

C3. nondeterminism induced by the absence of a shared clock and by the unreliable communication media (messages can be delayed, lost, duplicated, or reordered);

C4. data constraints on the inputs and/or outputs of components, which, combined with nondeterminism allow to calibrate the description of the system;

C5. activation constraints on the components, which allow to control the degree of asyn-

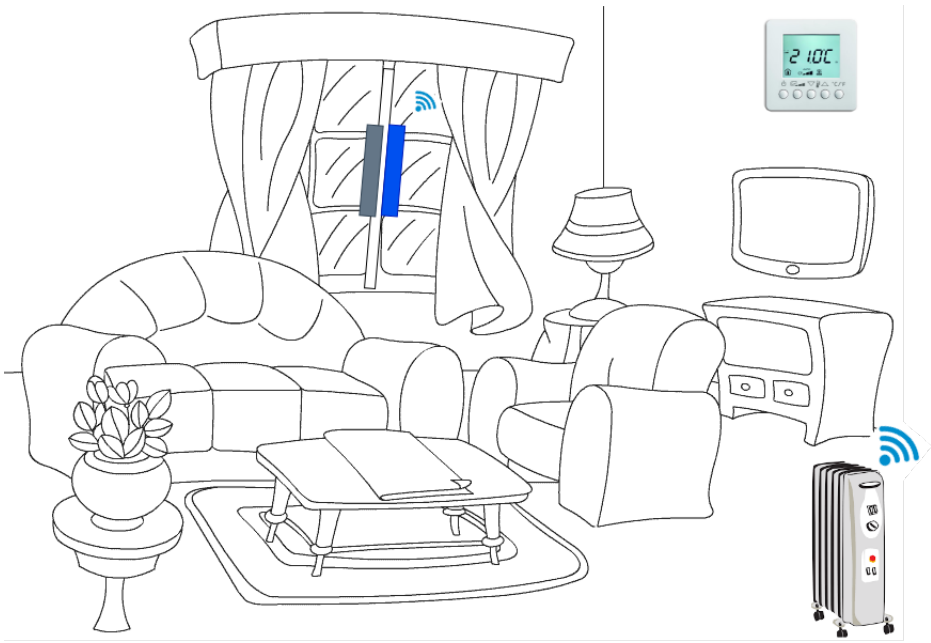


Figure 2.10: Simple smart room temperature management

chrony between components, and to describe realistic situations (failure and activation strategies, etc.) in an abstract way.

For instance, Figure 2.10 presents an example of a GALS system, a smart room temperature management system, whose goal is to control and monitor the room temperature of a smart home. The system has one smart window contact component, which detects if the window is open more than ten minutes, and a smart heater component, which regulates the room temperature and stops heating if the window is open for too long. Both components (the window contact and the smart heater) are independent, evolve concurrently at their own pace, and the window contact communicates the status of the window (open for more than ten minutes or closed) to the smart heater. In this smart room temperature management system, the smart window contact and the smart heater are two synchronous reactive systems: their behavior is deterministic and atomic (their executions are never interrupted). At any time, the smart window contact can detect that the window is open for more than ten minutes, and inform the smart heater; nondeterminism and interleaving are required to model this behavior. The smart heater increases his temperature gradually; to model this behavior, it is important to constrain the data (corresponding to the component inputs). The smart window contact information should always be taken into account as soon as possible; activation constraints enable to model this behavior.

Both synchronous and asynchronous languages have been used to model GALS systems. Each approach, applied individually, is unable to model all behavioral subtleties of GALS systems. Several combinations of synchronous and asynchronous languages to model GALS systems have been studied, such as CRSM-Promela [RSD+04], SystemJ [MGS12], Signal-

Promela [DMK+06], and SAM-LNT [GT09]. Table 2.3 compares these GALS modeling approaches by taking into account the five concepts described above, more particularly: (C1) the modelling of the synchronous atomicity; (C2-C3) the nondeterminism in order to model communication protocols, or an existing communication protocol; and (C4-C5) the modelling of activation strategies and of data constraints on the components.

Table 2.3: Approaches to model GALS systems

name	synchronous atomicity	communication		constraints	
		nondeterministic	protocol	data	activation
CRSM Promela	Esterel synch. semantics	yes	no	yes	no
SAM-LNT	functions	yes	user defined	yes	no
Signal Promela	Promela atomic constraints	yes	LTTA	yes	auto clock constraints
SYSTEMJ	Esterel synch. semantics	no	no	yes	no
GRL	synch. blocks	yes	user defined	yes	yes

Due to the different semantics and abstraction level, a direct modeling with only synchronous or asynchronous languages could be complex. Instead, using a single GALS-specific language enables to reduce that complexity. Although GALS models have been intensively studied, as far as we know GALS Representation Language (GRL) [JLM16, Jeb16] is the only existing DSL with formal semantics specially designed for GALS systems.

Thus, in this thesis, we use GRL to describe the global behavior of GALS systems. GRL is equipped with the GRL2LNT [JLM16, Jeb16] translator to LNT [GLS17], which provides a connection to the CADP verification toolbox [GLMS13].

2.4.1 An overview of the GRL language

We present here an overview of the GRL language, the reader familiar with GRL can safely skip this section. GRL [JLM16, Jeb16] is a formal language designed to model the behavior of GALS systems. GRL enables the behavioral modelling of synchronous components (determinism, atomicity), asynchronous communication (nondeterminism, asynchronous concurrency), and constraints involving both component activations and the data carried by component inputs. Concretely, GRL integrates the synchronous reactive model underlying dataflow languages and the asynchronous interleaving semantics of concurrency underlying process algebras.

GRL provides constructed data types, constants, statements built upon standard algorithmic control structures, and functions. GRL supports all the basic types of LNT (see

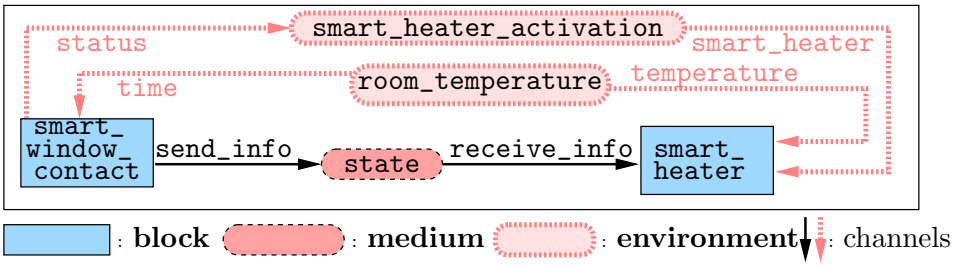


Figure 2.11: Architecture of the GRL model of the room temperature management

Section 2.3.1), such as **bool**, **int**, **real** numbers, and character **string**, as well as user-defined LNT types (bounded or unbounded), such as records, unions, lists, sets, and arrays. For instance, the following LNT type defines the enumerated data type `window_status`. The **with** clause specifies the predefined functions for this type (in our case, the comparison operations).

```
type window_status is
  open,
  close,
  already_sent
  with "!=", "=="
end type
```

The constants are defined by the keyword **const** and are necessarily initialized. Here is an example of numerical constant `max_open_time`, corresponding to the maximum duration (in minutes) for which the window could be open without a reaction of the smart heater:

```
const max_open_time: nat := 10;
```

The GRL model reflects the modular specification, connecting the synchronous components by a communication network. Figure 2.11 shows the architecture of a formal GRL model for the smart room temperature management seen in Figure 2.10. Each of the two synchronous components (`smart_window_contact` and `smart_heater`) is represented in GRL as a **block**, depicted as a (light blue) rectangle with solid border in Figure 2.11. These blocks exchange data via asynchronous communication media (`state`), which is represented in GRL as a **medium**, depicted as a (pink) ellipse with dashed border in Figure 2.11. The interaction between blocks has also to respect environmental constraints, represented in GRL as an **environment** (`room_temperature`), depicted as a (light pink) ellipse with thick dashed border in Figure 2.11. The overall model of the GALS is represented in GRL as a **system**, which describes the composition and interactions of blocks, media, and environments.

Block

A GRL block defines the deterministic code executed at each *activation* (i.e., clock pulse) by the synchronous component. GRL blocks behave as deterministic and atomic synchronous components that interleave with other blocks. For defining blocks, GRL provides a set of basic synchronous language constructs. GRL block statements are inspired by those of LNT, such as variable assignment, sequential composition, conditional (if-then-else), loops (for, while), and extended with block invocations, and communication primitives. Because the initial idea behind GRL blocks was to generate them from a synchronous language, one has to manually implement these three classical synchronous languages operators:

1) The *synchronous parallel* operator is not provided by GRL, the composition between GRL blocks is sequential and require to define a correct order between blocks.

2) The *delay operator* is not explicitly present, but GRL makes explicit the internal state through the usage of static variables. One should manage the access and update the block's internal states or the delay operators could be encoded in GRL libraries.

3) The *explicit loop*, for and while loops are also missing in GRL. One can encode bounded operators in GRL libraries.

Therefore, GRL is less user-friendly for describing synchronous components[5] than using a full-fledged synchronous language, such as Lustre [HCRP91]. For instance, the smart window contact of the smart room temperature management could be modeled by a GRL block `smart_window_contact` (defined below). This block has three **static** variables, which define its internal states:

- `opening_time` to keep track of the (abstract) time when the window is open from the previous activation (this time is considered to be initially null);
- `previous_info` to keep track of the perception information (open window or closed one) computed and sent during the previous activation;
- `is_open` to keep track of the perception information open or not even if the window is open for less than `max_open_time` minutes.

At each activation, the `smart_window_contact` block receives the current time (**in** parameter `time`). It then computes the current window status `send_info` indicating if the window is open (opened for more than `max_open_time` minutes), or closed. If there is a change between `previous_info` and `info` (the window was not open for more than `max_open_time` minutes, or did not close after being open), both the variable `previous_info` and the output `send_info` are updated; otherwise the output is set to the particular value `already_sent` indicating that the status of the window did not change. At the end of the activa-

[5]Historically, GRL was defined as a pivot language between synchronous programs with graphical syntax (e.g., function-block diagrams) and LNT, and evolved subsequently into a plain language for GALS systems.

tion, the computed window status info is sent to the connected medium (**send** parameter `send_info`), and it sends the status to the environment managing the activation of the block `smart_heater` through the **out** `status` channel. The function `perception_window_open` detects if the window is open or not.

```
block smart_window_contact (in time: nat, out status: window_status)
                           [send send_info: window_status] is
  static var opening_time: nat := 0,
         var previous_info: window_status := already_sent;
         var is_open: bool := false;
         var info: window_status
  -- detects if the window is open
  info := perception_window_open ();
  if previous_info == info then
    -- the same status as the previous activation
    info := already_sent
  elsif (info == open) and (previous_info == close) and (is_open == false) then
    -- the window has been opened since the last activation
    opening_time := time;
    info := previous_info;
    is_open := true
  elsif (info == open) and (time - opening_time > max_open_time) then
    -- the window has been open for more than max_open_time
    opening_time := 0;
    previous_info := open;
    info := open;
    is_open := false
  else
    -- the window has been closed since the last activation
    -- (less than max_open_time minutes ago)
    previous_info := close;
    info := close;
    is_open := false
  end if;
  send_info := info;
  status := info
end block
```

Medium

Synchronous blocks interact with each other via asynchronous communication media. GRL media do not impose any communication protocol, but provide enough expressiveness to model general communication mechanisms. Explicitly representing these media makes it possible to finely model a large panel of behaviors (i.e., message buffering, message loss,

nondeterminism, etc.). A medium is connected to each block by at most two channels, called **receive** and **send** channels. Note that a **receive** channel corresponds to the reception of some value in a variable prefixed by "?". Channels are tuples of variables, by which the synchronizations between components (rendezvous) take place. Each channel has an associated Boolean condition (tested with a **when** clause), stating whether a message is available or should be sent. GRL media statements are those of the blocks and extended with nondeterministic statements. For instance, the following GRL medium `state` enables the block `smart_window_contact` to send the current perception window status (via the **receive** channel `send_info`) to the block `smart_heater` (via the **send** channel `receive_info`); the transmission takes place only when the window status information has not already been sent.

```
medium state [receive send_info: window_status,
              send receive_info: window_status] is
  static var buffer: window_status := close
  select
    when ?send_info ->
      if send_info != already_sent then buffer := send_info end if
  []
    when receive_info -> receive_info := buffer
  end select
end medium
```

Environment

GRL environments model the external environment of blocks; by providing their inputs and receiving their outputs. Concretely, an environment is a loop, where at each iteration at most one block is activated. Blocks and environments are connected through input and output channels, with a primitive communication data signal, i.e., a data signal is executed by the environment, if an interaction on a channel occurs (receiving **in** or sending **out**). Note that an environment may be nondeterministic, e.g., it may contain nondeterministic choice, modelled in GRL using the **select** statement, as in LNT. For instance, the following GRL environment `room_temperature` ensures that: (1) the (abstract) time, which corresponds to the number of activations (`ticks`), is shared with the block `smart_window_contact` (by sending this information to `smart_window_contact` as input `time`); (2) the evolution of the (abstract) temperature is changed gradually and shared with the block `smart_heater` (by sending it as input `temperature`).

```
-- Environment constraining the data carried by blocks
environment room_temperature (out time: nat, out temperature: nat) is
  static var ticks: int := 0,
             pre_temperature: nat := 20
  loop
    ticks := ticks + 1;
```

```
    select
      -- The evolution of the temperature is abstract and changes gradually
      when temperature ->
        case pre_temperature is
          18 -> select
                   temperature := 18
                 [] temperature := 19
                 end select
        | 19 -> select
                   temperature := 18
                 [] temperature := 19
                 [] temperature := 20
                 end select
        | 20 -> select
                   temperature := 19
                 [] temperature := 20
                 [] temperature := 21
                 end select
        | 21 -> select
                   temperature := 20
                 [] temperature := 21
                 end select
        | any -> temperature := 20
        end case;
        pre_temperature := temperature
    []
      when time -> time := ticks
    end select
  end loop
end environment
```

Note that in order to execute its body several times, an environment should contain a loop, whereas the synchronous loop is built-in in block definitions, therefore, the body of a block contains only the code executed at each activation.

Block activations are particular implicit inputs of blocks, enabling an environment to precisely control the activations of synchronous blocks. GRL enables to model various activations, by using the keyword **enable**. In the following environment example, the activation signal **enable** implements the permission for the block `smart_heater` to execute only if the window is closed or open for less than ten minutes (`window_status`)):

```
-- Environment constraining the block activations
environment smart_heater_activation (block smart_heater
                                     in status: window_status) is
  loop
    when status -> if status == false then enable smart_heater end if
```

```
  end loop
end environment
```

System

GRL systems define GALS as sets of blocks, environments, and media. The system behavior can be refined to model detailed communication strategy and constraints. Also to reduce the complexity, one can model in a system the behavior of blocks separately, or a primary system behavior with simple communication media and test case scenarios. The following GRL **system** describes the complete model of the smart room temperature management, built by composing the blocks `smart_window_contact` and `smart_heater`, the medium `state`, and the environment `room_temperature`, introduced by the keywords **block list**, **medium list**, and **environment list**, respectively. The component aliasing `smart_heater` (`heater`) is required for the environment `smart_heater_activation` to control its activation.

```
system main (send_info, receive_info, status: window_status,
             time, temperature: nat) is
  alias smart_heater as heater
  block list
    smart_window_contact (time, ?status) [?send_info],
    heater (temperature) [receive_info]
  medium list
    state [send_info, ?receive_info]
  environment list
    room_temperature (?time, ?temperature),
    smart_heater_activation (heater, status)
end system
```

GRL defines the semantics of a GALS system as an LTS (Labeled Transition System), whose states represent the memories of all blocks, media, and environments of the system [Jeb16, Chapter 4]. The initial state is the initial memory, in which each static variable has its (mandatory) initial value. Each transition going out of a state corresponds to the atomic execution of a block, which consists in reading the values of input and receive channels, executing the code of that block activation, and producing the values for the outputs and send channels of the block. The transition is labeled with all these values, and its target state corresponds to the updated memories of the participating components, i.e., the block and its connected media and environments. Thus, the atomic executions of synchronous blocks are interleaved in the LTS, which reflects the GALS nature of the system.

For technical reasons, we rely in this thesis on the semantics of GRL as induced by the current translation-based implementation of GRL. In this semantics [Jeb16, Chapter 5], the execution of a synchronous block activation is split into an input transition followed

by an output transition. These two transitions are executed atomically, i.e., without interleaving with any other transitions corresponding to an activation of another block. GRL is equipped with the GRL2LNT [JLM16, Jeb16] translator to LNT [GLS17], which provides a connection to the CADP tools. For a more detailed presentation of GRL see [Jeb16, JLM16].

2.4.2 Verification of GALS systems

Milner's proposal [Mil83] to encode asynchronism in a synchronous process calculus has been used to specify GALS systems using synchronous programming approaches [HB02, GTL03, MGT+04, HM06]. These approaches do not retain a finer modeling of the overall GALS system and consider only the synchronous verification. The global aspects of a GALS system and the asynchronous validation are missing.

Following a bottom-up approach [GMM12], manually defining contracts for the synchronous components (to be verified locally, for instance using SCADE [Ber07]), the overall GALS system can be verified by translating the network of component contracts and verification properties into Promela (for verification with SPIN [Hol03]) or timed automata (for timing analysis with UPPAAL [BDL+01]).

A graphical tool set [RSD+04], based on the specification of the synchronous components as communicating reactive state machines, translates the system and its properties specified as observers into Promela for verification with SPIN. This tool focuses on the verification of the overall GALS system.

Encapsulating synchronous components makes the overall GALS system amenable for analysis with asynchronous verification tools. This approach has been followed for a combination of the synchronous language SAM, the asynchronous language LNT, and the CADP toolbox [GT09], and a combination of the synchronous language SIGNAL, the asynchronous language Promela, and the model checker SPIN [DMK+06].

Focusing on communication media for GALS hardware circuits, the asynchronous connections between synchronous blocks can be encoded into variants of Petri nets dedicated to the analysis of hardware circuits [BSY17]. On the contrary, our approach targets the testing of synchronous components of more generic GALS systems, relying on less precise models of the communication signals.

As far as we know, we propose the first approach of Model Based Testing for GALS systems.

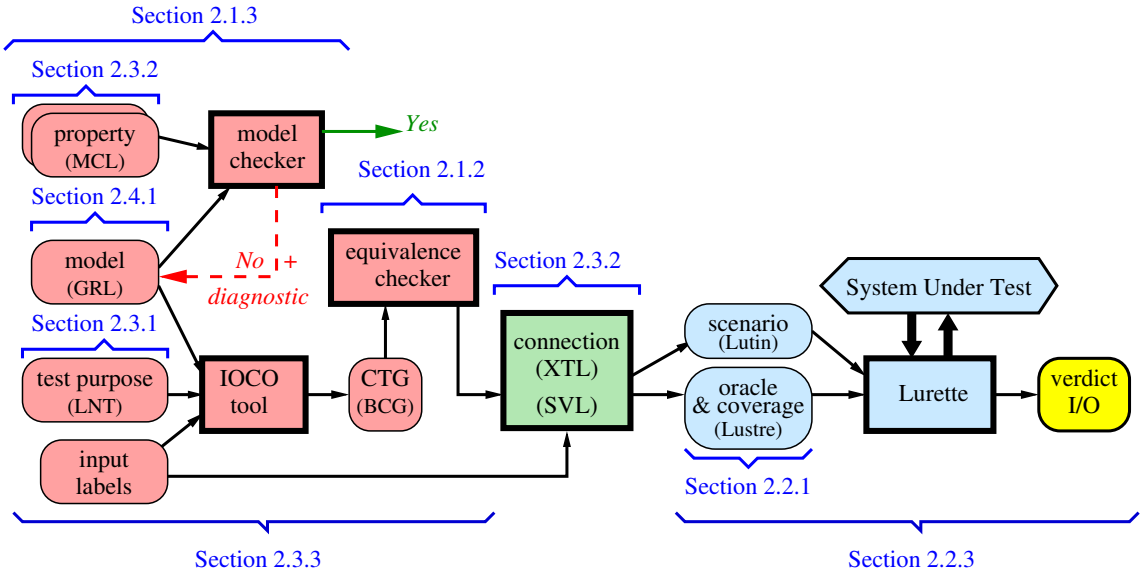


Figure 2.12: Overview of the tools and languages selected for this thesis

⋆⋆⋆

In this thesis, we retain a fine modeling of the synchronous components in the overall GALS system and consider the validation of both, the overall GALS system and its synchronous components. The overall flow of our approach is illustrated in Figure 2.12. For modelling and validating the synchronous components, we will use the synchronous data flow language Lustre (see Section 2.2.1) and the synchronous Lurette testing tool (see Section 2.2.3). For the global GALS behavior model, we chose the formal GRL language (see Section 2.4.1). We will verify the global asynchronous model using model checking (see Section 2.1.3) with MCL formulas (see Section 2.3.2), and test it using a new **ioco** model-based testing tool (see Section 2.3.3) similar in spirit to the TGV tool (see Section 2.3.3). In order to connect the synchronous and asynchronous techniques, we will also use equivalence checking (see Section 2.1.2) and the XTL tool (see Section 2.3.2). We encapsulated all tool invocations in an SVL script (see Section 2.3.2).

Chapter 3

Validation Techniques for Synchronous Components

In this chapter we illustrate formal techniques for the functional testing of a synchronous component, using a case study of a synchronous dataflow algorithm: The Message Authenticator Algorithm (MAA), a pioneering cryptographic function designed in the mid-80s at the National Physical Laboratory (NPL, United Kingdom). Part of this work has been published in [GM17] and in [GM18].

The chapter is organized as follows. Section 3.1 presents the MAA, both from a technical and an historical perspective. Section 3.2 presents the modeling of the MAA using the synchronous dataflow language Lustre. Section 3.3 precises how the Lustre model has been validated. Section 3.4 gives an overview of our supplementary work on three other formal models describing the full MAA. Finally, we give summarizing remarks on our work and synchronous testing.

3.1 The Message Authenticator Algorithm (MAA)

In data security, a Message Authentication Code (MAC) is a short sequence of bits that is computed from a given message; a MAC ensures both the authenticity and integrity of the message, i.e., that the message sender is the stated one and that the message contents have not been altered. A MAC is more than a mere checksum, as it must be secure enough to defeat attacks; its design usually involves cryptographic keys shared between the message sender and receiver.

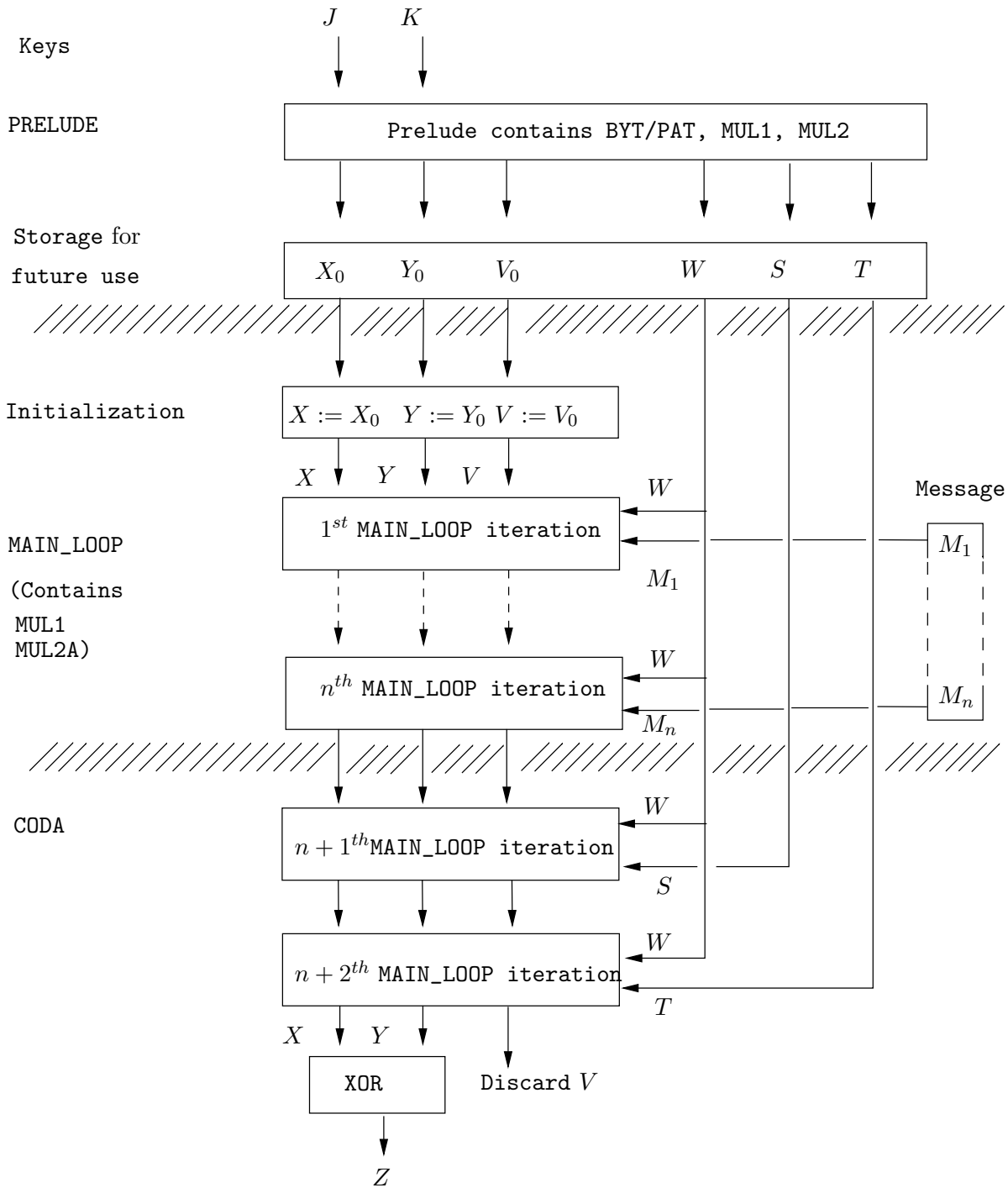


Figure 3.1: MAA data flow

One of the first MAC algorithms to gain widespread acceptance was the Message Authenticator Algorithm (MAA, also known as the Message Authentication Algorithm) [Dav85, DC88, Pre11] designed in 1983 by Donald Davies and David Clayden at the National Phys-

ical Laboratory (NPL) in response to a request of the UK Bankers Automated Clearing Services. The MAA was adopted by ISO in 1987 and became part of the international standards 8730 [ISO86] and 8731-2 [ISO87]. Later, cryptanalysis of the MAA revealed various weaknesses, including feasible brute-force attacks, existence of collision clusters, and key-recovery techniques [PvO96, RPD96, PRvO97, PvO99, Pre11]. For this reason, the MAA was withdrawn from ISO standards in 2002.

Nowadays, message authentication codes are computed using different families of algorithms based on either cryptographic hash functions (HMAC), universal hash functions (UMAC), or block ciphers (CMAC, OMAC, PMAC, etc.). Contrary to these modern approaches, the MAA was designed as a standalone algorithm that does not rely on any preexisting hash function or cipher. In this section, we briefly explain the principles of the MAA. More detailed explanations can be found in [Dav85, DC88] and [MvOV96, Algorithm 9.68].

The MAA was intended to be implemented in software and to run on 32-bit computers. Hence, its design intensively relies on 32-bit words (called blocks) and 32-bit machine operations. Figure 3.1 (inspired by [DC88, Figure 21]), gives an overview of the data flow of the MAA. The MAA takes as inputs a key and a message. The key has 64 bits and is split into two blocks J and K. The message is seen as a sequence of blocks. If the number of bytes of the message is not a multiple of four, extra null bytes are added at the end of the message to complete the last block. The size of the message should be less than 1,000,000 blocks; otherwise, the MAA result is said to be undefined; we believe that this restriction, which is not inherent to the algorithm itself, was added in the second ISO standard [ISO92] to provide MAA implementations with an upper bound (four megabytes) on the size of memory buffers used to store messages.

The MAA produces as output a block, which is the MAC value computed from the key and the message. The fact that this result has only 32 bits proved to be a major weakness enabling cryptographic attacks; MAC values computed by modern algorithms now have a much larger number of bits. Apart from the aforementioned restriction on the size of messages, the MAA behaves as a totally-defined function; its result is deterministic in the sense that, given a key and a message, there is only a single MAC result, which neither depends on implementation choices nor on hidden inputs, such as nonces or randomly-generated numbers.

The MAA calculations rely upon conventional 32-bit logical and arithmetic operations, among which: `AND` (conjunction), `OR` (disjunction), `XOR` (exclusive disjunction), `CYC` (circular rotation by one bit to the left), `ADD` (addition), `CAR` (carry bit generated by 32-bit addition), `MUL` (multiplication, sometimes decomposed into `HIGH_MUL` and `LOW_MUL`, which denote the most- and least-significant blocks in the 64-bit product of a 32-bit multiplication). On this basis, more involved operations are defined, among which `MUL1` (result of a 32-bit multiplication modulo $2^{32} - 1$), `MUL2` (result of a 32-bit multiplication modulo $2^{32} - 2$),

`MUL2A` (faster version of `MUL2`), `FIX1` and `FIX2` (two unary functions[1] respectively defined as $x \rightarrow$ `AND`(`OR`(x,A),C) and $x \rightarrow$ `AND`(`OR`(x,B),D), where A, B, C, and D are the four hexadecimal block constants A = `02040801`, B = `00804021`, C = `BFEF7FDF`, and D = `7DFEFBFF`). The MAA operates in three successive phases:

- The `PRELUDE` takes the two blocks J and K of the key and converts them into six blocks X_0, Y_0, V_0, W, S, and T. This phase is executed once. After the prelude, J and K are no longer used.
- The `MAIN_LOOP` successively iterates on each block M_n of the message (M_1, ..., M_n). This phase maintains three variables X, Y, and V (initialized to X_0, Y_0, and V_0, respectively), which are modified at each iteration. The main loop also uses the value of W, but neither S nor T.
- The `CODA` adds the blocks S and T at the end of the message and performs two more iterations on these blocks. After the last iteration, the MAA result is XOR(X, Y), called Z.

In 1987, the second ISO standard [ISO87, Section 5] introduced an additional feature (called mode of operation), which concerns messages longer than 256 blocks and which, seemingly, was not present in the early MAA versions designed at NPL.

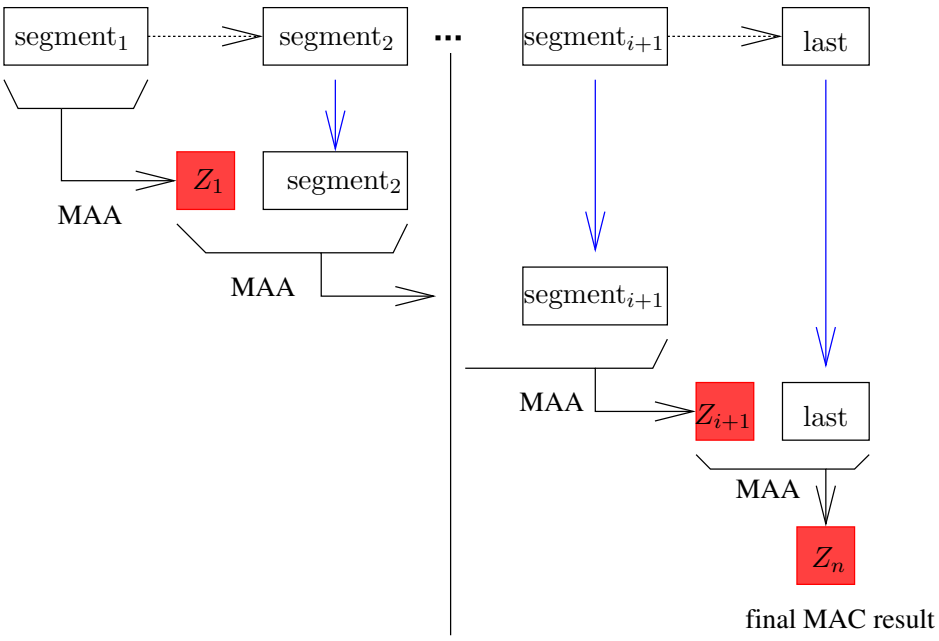


Figure 3.2: Mode of operation of the MAA

Figure 3.2 gives an overview of "the mode of operation". Each message longer than 256 blocks must be split into segments of 256 blocks each, with the last segment possibly

[1] The names `FIX1` and `FIX2` are borrowed from [Mun91b, pages 36 and 77].

containing less than 256 blocks. The above MAA algorithm (prelude, main loop, and coda) is applied to the first segment, resulting in a value noted Z_1. This block Z_1 is then inserted before the first block of the second segment, leading to a 257-block message to which the MAA algorithm is applied, resulting in a value noted Z_2. This is done repeatedly for all the n segments, the MAA result Z_i computed for the i-th segment being inserted before the first block of the $(i+1)$-th segment. Finally, the MAC for the entire message is the MAA result Z_n computed for the last segment.

From the point of view of formal methods, the MAA is interesting because of its pioneering nature, because its definition is freely available and stable, and because it is involved enough while remaining of manageable complexity. Over the past decades, various formal specifications of the MAA have been developed, either non-executable ones using VDM in 1990 [PO90, PO91], using Z in 1991 [Lai91], and LOTOS in 1990/1991 [Mun91b, Mun91a] or executable ones using LOTOS, written by Hubert Garavel and Philippe Turlier in 1992 (LOTOS-92), and using LNT written by Wendelin Serwe in 2016 (LNT-16). For such formalism, the usual examples often deal with syntax trees, which are explored using standard traversals (breadth-first, depth-first, etc.); contrary to such commonplace examples, cryptographic functions (and the MAA, in particular) exhibit more diverse behavior, as they rather seek to perform irregular computations than linear ones.

3.2 Modelling the MAA in Lustre

We considered the MAA models LOTOS-92 and LNT-16 and extended them to model the full MAA algorithm in the synchronous formal language Lustre [HCRP91] (more details about Lustre are available in Chapter 2 Section 2.2.1). We decided to take advantage of Lustre connection to robust testing tools, actively supported and maintained by developers. A large part of our Lustre model contains general definitions, half of which are largely independent of the MAA. The whole Lustre model of the MAA is presented in the Appendix A. The architecture of this model follows the dataflow shown on Figure 3.1.

We chose to represent blocks as words of four bytes, rather than thirty-two bits. So doing, the logical operations on blocks (`AND`, `OR`, `XOR`, and `CYC`) are easy to define using bitwise and bytewise manipulations. The bits, bytes and blocks are represented with the three following Lustre data structures:

```
type Bit = enum {X0,X1};

type Octet = struct {x1: Bit; x2: Bit; x3: Bit; x4: Bit;
                    x5: Bit; x6: Bit; x7: Bit; x8: Bit};

type Block = struct {o1: Octet; o2: Octet; o3: Octet; o4: Octet};
```

Then, we defined a set of functions to implement the corresponding logical operations

on bits, bytes, and blocks. The arithmetical operations (`ADD`, `CAR`, and `MUL`) have been implemented using 8-bit, 16-bit, and 32-bit adders and multipliers, more or less inspired from the theory of digital circuits. Thus, the structure `Pair` shown below, represents results of the multiplication of two blocks.

```
type Pair = struct {w1: Block; w2: Block};
```

A message is a list of blocks, each block of the message is an input of the main `MAC` node in Lustre, shown in Figure 3.3. Long messages (i.e., containing more than one block) are hard coded in functions, as for instance, the twenty-words message in the Appendix A.16.

The `MAC` node takes as input the message to encode, and a key, which corresponds to two words (called K and J). A local variable `init` defines the beginning of a message, and another variable `n` stores the number of blocks, in order to ensure the implementation of the "mode of operation" of the MAA, i.e., segmentation of messages larger than 1024 bytes (as explained in Section 3.1). The MAC gives as outputs the result of the intermediate computations (`prelude`, `mainloop`) in the auxiliary blocks (`X, Y, V, W, S, T`), and the result of its main computation, which is represented with a variable (`Z`) containing the MAC for the given input (one key and a message). At each call of the Lustre node `MAC`, one new block of the message is processed as follows:

- for the 1st block of the message, the auxiliary variables (`X`, `Y`, and `V`) are computed with an iteration of the `mainLoop` function with the initial values (`X0, Y0, V0`) computed by the function `prelude` shown in Figure 3.3;
- for the 257th block of the message, the auxiliary variables are computed with an iteration of the `mainLoop2` function, which consists of two iterations of the `mainLoop`, one on the result of the previous coda (`Z`), and one on this block message;
- and, for the other blocks of the message, the auxiliary variables are computed by taking into account the previous auxiliary variables (`pre X`, `pre Y`, and `pre V`) computed by the previous block.

Keys could also be represented using the type Pair, but we prefer introducing the following dedicated structure `Key` to clearly distinguish between keys and, e.g., results of the multiplication of two blocks:

```
type Key = struct {K: Block; J: Block};
```

We defined the "multiplicative" functions used for MAA computations, most of which were present in [DC88] or have been later introduced in [MvOV96]. The three principal low-level operations are MUL1, MUL2, and MUL2A, with its auxiliary functions. We also defined the higher-level functions that implement the MAA algorithm on one segment (maximum 1024-byte), namely the prelude, the inner loop, the coda, as well as the principal function MAC (that computes the 4-byte signature of a message).

It turned out that Lustre enables an elegant modeling of the involved data structures, such

```
1  node MAC (KJ: Key, word: Block)
2          returns (X, Y, V, W, S, T, Z: Block; n: int);
3  var X0, Y0, V0: Block;
4  let
5    -- identify the number of the block
6    init = true -> false;
7    n = if init then 1 else if pre n = 256 then 0 else pre n + 1;
8    -- prelude, initialisations
9    X0, Y0, V0, W, S, T = prelude (KJ.J, KJ.K);
10   -- main loops
11   X, Y, V = if init then
12               mainLoop (X0, Y0, V0, W, word)
13             else if n = 0 then
14               -- mode of operation
15               mainLoop2 (X0, Y0, V0, W, pre Z, word)
16             else mainLoop (pre X, pre Y, pre V, W, word);
17   -- coda
18   Z = coda (X, Y, V, W, S, T);
19 tel;
```

Figure 3.3: The node MAC

as enumeration (e.g., the enumeration `Bit`) and structures (e.g., the type Byte defined as `Octet`). We briefly discuss below some of our choices for modeling the MAA in Lustre.

Usage of local variables. Local variables are essential to store computed results that need to be used several times, thus avoiding identical calculations to be repeated. Lustre allows to freely define and assign local variables; the compiler guaranties that each variable is duly assigned before used. However, Lustre forbids successive assignments to the same variable. For instance, the `MUL2` function can be expressed in Lustre as follows:

```
function MUL2 (w1, w2 : Block) returns (w: Block);
var w1w2, w3w4, w5w6: Pair; w3: Block;
let
  w1w2 = mulBlock (w1, w2);
  w3w4 = ADDC (w1w2.w1, w1w2.w1);
  w3 = addBlock (w3w4.w2, addBlock (w3w4.w1, w3w4.w1));
  w5w6 = ADDC (w3, w1w2.w2);
  w = addBlock (w5w6.w2, addBlock (w5w6.w1, w5w6.w1));
tel;
```

Functions computing several results. There are several such functions in the MAA; let us consider the `prelude` function which takes two block parameters `J` and `K` and returns six block parameters `X`, `Y`, `V`, `W`, `S`, and `T`. By exploiting the fact that Lustre

```
1  function prelude (J, K: Block) returns (X, Y, V, W, S, T: Block);
2  var P: Octet; J1, J12, J14, J16, J18, J22, J24, J26, J28: Block;
3      K1, K12, K14, K15, K17, K22, K24, K25, K27, K19, K29: Block;
4      H4, H0, H5, H6, H7, H8, H9: Block;
5  let
6    J1, K1 = BYT (J, K);
7    P = PAT (J, K);
8    J12, J14, J16, J18, J22, J24, J26, J28 = preludeJ (J1);
9    K12, K14, K15, K17, K19, K22, K24, K25, K27, K29 = preludeK (K1);
10   H4, H6, H8 = preludeHJ (J14, J16, J18, J24, J26, J28);
11   H0, H5, H7, H9 = preludeHK (K15, K17, K19, K25, K27, K29, P);
12   X, Y = BYT (H4, H5);
13   V, W = BYT (H6, H7);
14   S, T = BYT (H8, H9);
15 tel;
```

Figure 3.4: The function prelude

functions may return one or several parameters, we can elegantly define the `prelude` function in Lustre as shown on Figure 3.4. The definitions of the auxiliary functions `preludeJ`, `preludeK`, `preludeHJ`, and `preludeHK` are in Appendix A.12. This function is invoked as follows (as it is done in the MAC node):

```
X, Y, V, W, S, T = prelude (J, K);
```

Our complete Lustre model of the MAA (see Appendix A) has 1271 lines. In the next section, we present the validation process of this model, based on the testing framework available for Lustre.

3.3 Testing the MAA Model

To validate our Lustre model, we defined four sets of test vectors derived from the specification in [DC88]:

(T1) implements the 36 checks listed in Tables 1, 2, and 3 of [DC88]. These test vectors specify, for a few given keys and messages, the expected values of intermediate calculations (e.g., MUL1, MUL2, MUL2A, etc.).

(T2) is based upon Table 4 of [DC88], checking if the main loop of MAA (as described on page 10 of [DC88]) is correctly implemented on six groups of checks (three single-block messages and one three-block message); T2 corresponds to 56 tests.

(T3) is based upon Table 5 of [DC88], checking if the MAA signature is correctly computed

on four groups of checks, with two different keys and two different messages; T3 corresponds to 64 tests.

(T4) checks all intermediary values of the algorithm with a message of 20 blocks containing only zeros directly taken from Table 6 of [DC88], T4 corresponds to 45 tests.

In all these series of test vectors [DC88], which corresponds to 201 tests, we found mistakes, which we documented and for which we gave corrections in [GM18, Annex A].

We automate the test execution process, by using the testing tool Lurette [JRB06] taking advantage of its connection with the Lustre language.

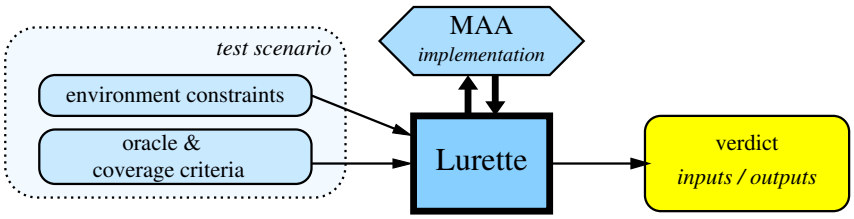


Figure 3.5: Overview of the Lurette testing tool

Figure 3.5 gives an overview of Lurette in the context of testing our MAA model. Lurette takes two inputs: (i) an input constraints in Lutin [RRJ08]; (ii) an oracle implementing the test decision, in our case the 12 possible pairs (one key K J, and a message of two blocks), and the expected resulting MAC; and some parameters controlling the execution and the coverage of the generated input sequences, e.g., the number of steps (n) in the execution sequence. The Lurette tool is also connected to a system under test, in our case the Lustre implementation of the MAA. More details about Lurette are available in Section 2.2.3 (Chapter 2).

As seen below, we consider hard-coded messages, and for using Lurette, the inputs of the system under test must only use the datatypes provided by Lutin (Boolean, reals, and integers). Thus, we rewrote the Lustre `MAC` node as below: it has two inputs, a natural number corresponding to the id of one key-message pair and a Boolean expressing the beginning of a new message segment. In the rewritten Lustre node (Figure 3.6), the function `get_mess_key` (line 6) takes as input an id, and returns the corresponding key and a message (the possible messages blocks and pairs are hard-coded in this function).

The Lutin code in Figure 3.7 constrains the input of the MAA to one out of four messages, containing each one two words.

Note that if the input values are not explicitly constrained in the Lutin environment, random numbers will be generated. The environment contains the 12 possible pairs of keys and messages. Although larger and more complex environments could be written, this task is tedious and error-prone, in particular due to the representation of messages and keys by natural numbers.

```
1  node MAC (id: int; init: bool)
2          returns (X, Y, V, W, S, T, Z: Block; n: int);
3  var X0, Y0, V0: Block;
4     KJ: Key; Mn: Block;
5  let
6    Mn, KJ = get_mess_key (id);
7    n = if init then 1
8        else if pre n = 256 then 0
9        else pre n + 1;
10   -- initialisations
11   X0, Y0, V0, W, S, T = prelude (KJ.J, KJ.K);
12   -- mainloops
13   X, Y, V = if init then
14               mainLoop (X0, Y0, V0, W, Mn)
15             else if n = 0 then
16               -- mode of operation
17               mainLoop2 (X0, Y0, V0, W, pre Z, Mn)
18             else mainLoop (pre X, pre Y, pre V, W, Mn);
19   -- coda
20   Z = coda (X, Y, V, W, S, T);
21 tel;
```

Figure 3.6: The MAC node rewritten

```
1 node environment () returns (id: int; init: bool) =
2 loop {
3 |id = 1 and init = true fby id = 12 and init = false
4 |id = 2 and init = true fby id = 22 and init = false
5 |id = 3 and init = true fby id = 32 and init = false
6 |id = 4 and init = true fby id = 42 and init = false
```

Figure 3.7: Environment

Table 3.8 contains the test vector (T3), more precisely, four groups of checks, with two different keys and two different messages.

MAA Variables	Message 1	Message 2	Message 3	Message 4
J	00FF 00FF	00FF 00FF	5555 5555	5555 5555
K	0000 0000	0000 0000	5A35 D667	5A35 D667
P	FF	FF	00	00
X0	4A64 5A01	4A64 5A01	34AC F886	34AC F886
Y0	50DE C930	50DE C930	7397 C9AE	7397 C9AE
V0	5CCA 3239	5CCA 3239	7201 F4DC	7201 F4DC
W	FECC AA6E	FECC AA6E	2829 040B	2829 040B
M1	5555 5555	AAAA AAAA	0000 0000	FFFF FFFF
X	48B2 04D6	6AEB ACF8	2FD7 6FFB	8DC8 BBDE
Y	5834 A585	9DB1 5CF6	550D 91CE	FE4E 5BDD
M2	AAAA AAAA	5555 5555	FFFF FFFF	0000 0000
X	4F99 8E01	270E EDAF	A70F C148	CBC8 65BA
Y	BE9F 0917	B814 2629	1D10 D8D3	0297 AF6F
S	51ED E9C7	51ED E9C7	9E2E 7B36	9E2E 7B36
X	3449 25FC	2990 7CD8	B1CC 1CC5	3CF3 A7D2
Y	DB91 02B0	BA92 DB12	29C1 485F	160E E9B5
T	24B6 6FB5	24B6 6FB5	1364 7149	1364 7149
X	277B 4B25	28EA D8B3	288F C786	D048 2465
Y	D636 250D	81D1 0CA3	9115 A558	7050 EC5E
Z	F14D 6E28	A93B D410	B99A 62DE	A018 C83B

Figure 3.8: Tests (T3) inspired by Table 5 of [DC88].

In our example, correct tests consist of the right output (a correct MAC value), given an input vector (the id of the pair of one key and a message). A small example of oracle for the previous Lutin code is given by the Lustre code of Figure 3.9, containing the test decisions and the expected output: the auxiliary block values of X, Y and MAC value for each block of message given in the Table 3.8.

The Lustre and Lutin model for the sets of test vectors (T1), (T2), (T3), (T4) has 1875 lines and contains 9 types (structures), 442 constants, 100 functions, and 2 nodes. Our Lustre model of the MAA was validated by the 201 tests successfully.

```
1  node oracle (id: int; init: bool; X, Y, Z: Block; n: int)
2              returns (res: bool);
3  let
4    res = true ->
5       -- X, Y: 1st MainLoop iteration
6     ((id = 1 and init and X = x48B204D6 and Y = x5834A585) or
7       -- X, Y: 2nd MainLoop iteration
8      (id = 12 and not init and X = x4F998E01 and Y = xBE9F0917) or
9      -- Z: coda
10     (id = 12 and Z = xF14D6E28) or
11     -- X, Y: 1st MainLoop iteration
12     (id = 2 and init and X = x6AEBACF8 and Y = x9DB15CF6) or
13     -- X, Y: 2nd MainLoop iteration
14     (id = 22 and not init and X = x270EEDAF and Y = xB8142629) or
15     -- Z: coda
16     (id = 22 and Z = xA93BD410) or
17     -- X, Y: 1st MainLoop iteration
18     (id = 3 and init and X = x2FD76FFB and Y = x550D91CE) or
19     -- X, Y: 2nd MainLoop iteration
20     (id = 32 and not init and X = xA70FC148 and Y = x1D10D8D3) or
21     -- Z: coda
22     (id = 32 and Z = xB99A62DE) or
23     -- X, Y: 1st MainLoop iteration
24     (id = 4 and init and X = x8DC8BBDE and Y = xFE4E5BDD) or
25     -- X, Y: 2nd MainLoop iteration
26     (id = 42 and not init and X = xCBC865BA and Y = x0297AF6F) or
27     -- Z: coda
28     (id = 42 and Z = xA018C83B)
29     -- All test vectors
30     and CHECK () = true);
31 tel;
```

Figure 3.9: Oracle

3.4 Formal models of the Message Authenticator Algorithm (MAA)

Since Lustre does not implement the list data type, we had to to consider only messages with fixed size in our Lustre model. Therefore, we also undertook the translation of the LOTOS-92 and the LNT-2016 models of the MAA, in the term rewriting language REC proposed in [DRB+09, Sect. 3] and [DRB+10, Sect. 3.1], and in the process algebra languages: LOTOS and LNT[2].

First, we undertook the translation of LOTOS-92 into a term rewrite system (REC-17). This system was encoded in the simple language REC proposed in [Gar89, Sect. 3] and [DRB+09, Sect. 3.1], which was slightly enhanced to distinguish between free constructors and non-constructors. Contrary to higher-level languages such as LOTOS or LNT, REC is a purely theoretical language that does not allow to import external fragments of code written in a programming language. Thus, all types (starting by the most basic ones, such as Bit and Bool) and their associated operations were exhaustively defined "from scratch" in the REC language. To check whether the MAA calculations are correct or not, the REC model was enriched with the 201 test vectors used also to validate the Lustre model, and 2 others to validate the "mode of operation" [GM17, Annexes B.18 to B.21]. The resulting REC model has 1575 lines and contains 13 sorts, 18 constructors, 644 nonconstructors, and 684 rewrite rules. Using a collection of translators developed at Inria Grenoble, the REC model was automatically translated into various languages: AProVE (TRS), Clean, Haskell, LNT, LOTOS, Maude, mCRL2, OCaml, Opal, Rascal, Scala, SML, Stratego/XT, and Tom. Using the interpreters, compilers, and checkers available for these languages, it was shown [GM17, Sect. 5] that the REC model terminates, that it is confluent, and that sall the 203 tests pass successfully.

Then, we also performed a major revision of LOTOS-92 based upon the detailed knowledge of the MAA acquired during the development of the REC model. Our goal was to produce an executable LOTOS model (LOTOS-17) as simple as possible, even if it departed from the original not executable model LOTOS-91 written by Harold B. Munster. Many changes were brought: the Nat sort was replaced almost everywhere by the Block sort; about seventy operations were removed, while a dozen new operations were added; the Block constructor evolved by taking four bytes rather than thirty-two bits; the Prelude operation is executed only once per message, rather than once per segment; the detection of messages larger than 1,000,000 blocks is now written directly in C; etc. These changes led to a 266-line LOTOS model (see Annex C) with two companion C files (157 lines in total) implementing the basic operations on blocks. Interestingly, all these files taken together are smaller than the original model LOTOS-91, demonstrating that executability and conciseness are not necessarily antagonistic notions. Our LOTOS model was validated by the CÆSAR.ADT compiler, which implements all the syntactic and semantic checks

[2]More details about LNT are available in Chapter 2 Section 2.3.1).

stated in the definition of LOTOS [ISO89]. The C code generated from the LOTOS model passed the test vectors specified in [ISO90, Annexes E.3.4 and E.4].

Finally, we entirely rewrote LNT-16 in order to obtain a simpler model. First, the same changes as for LOTOS-17 were applied to the LNT model. Also, the sorts Pair, TwoPairs, and ThreePairs, which had been introduced by Harold B. Munster to describe functions returning two, four, and six blocks, have been eliminated; this was done by having LNT functions that return their computed results using "out" or "in out" parameters (i.e., call by result or call by value result) rather than tuples of values; the principal functions (e.g., `MUL1`, `MUL2`, `MUL2A`, `PRELUDE`, `CODA`, `MAC`, etc.) have been simplified by taking advantage of the imperative style LNT, i.e., mutable variables and assignments; many auxiliary functions have been gathered and replaced by a few larger functions (e.g., `PreludeJ`, `PreludeK`, `PreludeHJ`, and `PreludeHK`) also written in the imperative style. These changes resulted in a 268-line LNT model with a 136-line companion C file, which have nearly the same size as LOTOS-17 , although the LNT version is more readable and closer to the original MAA model [DC88], also expressed in an imperative style. As for Lustre, the LNT model was then enriched with the same collection of test vectors, and supplementary test vectors intended to specifically check for certain aspects (byte permutations and message segmentation) that were not enough covered by the above tests; this was done by introducing a makeMessage function acting as a pseudo-random message generator. Finally, the remaining test vectors of [ISO90, Annexes E.3.4 and E.4], which were too lengthy to be included in the Lustre model and the REC-17, have been stored in text files and can be checked by running the C code generated from the LNT model. This makes LNT-17 the most complete formal model of the MAA as far as validation is concerned. Our LNT model was validated by the LNT2LOTOS translator, which implements the syntactic checks and (part of) the semantic checks stated in the definition of LNT [CCG+19] and generates LOTOS code, which is then validated by the CÆSAR.ADT compiler, therefore performing the remaining semantics checks of LNT. The C code generated by the CÆSAR.ADT compiler passed the test vectors specified in [ISO92, 17, Annex A], in [ISO90, Annexes E.3], in [ISO90, Annexes E.3.4 and E.4], and the supplementary test vectors based on an additional function building messages.

⋆⋆⋆

The modelisation of the MAA and its validation enabled us to discover various mistakes in prior (informal and formal) specifications of the MAA:

1. a mistake in the test vectors for the function PAT (part of the function Prelude), given in [ISO92, Annex A] and [DC88], the correction and more details are available in [GM17, Annex A];
2. a mistake in the test vectors for the full MAA algorithm, given in [ISO90, Annex E], the correction and more details are available in [GM18, Annex A];
3. an error was found in the main C program provided by [DC88], which computed an incorrect MAC value, as the list of blocks storing the message was built in reverse

order;

4. three others errors were found in the Lustre v6 toolbox: (i) lv6 compiler generating invalid C code if a variable identifier uses quotes ('); (ii) another issue with constant structures and the compiler lv6; and (iii) Lurette tool ignoring -2c-exec option; all these errors have been communicated to the Lustre v6 developers and have been quickly fixed by Erwan Jahier;

5. another error was found in the external implementation in C of the function `HIGH_MUL`, which computes the highest 32 bits of the 64-bit product of two blocks and is imported by the LOTOS and LNT models - this illustrates the risks arising when formal and non-formal codes are mixed.

It is however fair to warn the reader that testing our Lustre MAA model was a tedious task: in order to automate the tests validation with a Lutin environment and an oracle, we had to define more than 400 constants. An automation of this testing process is therefore highly beneficial, as it will be illustrated in Chapter 5.

Chapter 4

Validation of Communication Protocols between Components

In this chapter we illustrate the formalization and the functional testing of communication protocols between synchronous components, taking as a case study the TLS (Transport Layer Security) handshake version 1.3 [DR06, IET18], a protocol responsible for the authentication and the exchange of keys necessary to establish or resume a secure communication. The TLS [BMMW18] model has been validated using a new on-the-fly conformance test case generation tool named TESTOR [MMS18].

This chapter is organized as follows. Section 4.1 gives an overview of the TLS 1.3 handshake, from a technical perspective. Section 4.2 presents the LNT model of the TLS 1.3 handshake. Section 4.3 presents our new on-the-fly conformance test case generation tool TESTOR and describes various experiments to validate it. Section 4.4 describes our validation approach for the TLS handshake, using TESTOR to generate test cases.

4.1 Transport Layer Security Handshake Protocol

Security services are frequently used in fields like online banking, e-government and online shops. With increased availability of such services, the number of security risks rises both for users and providers alike. In order to ensure a secure communication between peers in terms of authenticity, privacy, and data integrity, cryptographic protocols are applied to regulate the data transfer. These protocols provide a standardized set of rules and methods for the interaction between peers. Transport Layer Security (TLS) [DR06] is a widely used security protocol, designed as a successor of Secure Sockets Layer (SSL) [Wea06]. Both protocols encompass a set of rules for the communication between client and server and rely on public-key cryptography to ensure integrity of exchanged data.

In this chapter, we focus on the TLS handshake protocol, which enables a TLS client

and server to establish a secure, authenticated communication link. The TLS handshake consists of four steps: (i) consent on the version of the protocol to use and choose cryptographic algorithms; (ii) exchange and validate certificates to authenticate each other; (iii) generate a shared secret key; and (iv) abort the handshake with an alert if something goes wrong. We present below the main TLS 1.3 handshake messages exchanged, as well as the exchange process between the server and the client(s).

4.1.1 Handshake TLS 1.3 interactions

The TLS messages make up the interaction between a client and a server and exchange values between them. The most important feature is the negotiation of cryptographic parameters that ensures privacy and integrity of exchanged information. During the protocol, client and server agree on used protocol version, exchange random values and select cryptographic algorithms for encryption and decryption of transferred data. Both peers exchange keys and certificates and after the handshake is finished, they can start to encrypt and exchange application data.

A summary of the interactions and the message exchanges from the client side and the server side is respectively depicted in the state machines [IET18] shown on Figure 4.1 and Figure 4.2. The state machines are transition-based (i.e., the state names are for sake of readability only), and conditional actions are represented in brackets ([ ]).

Note that these state machines do not represent the client's and server's `Alert` message interactions. The server and the client have to abort the handshake with an `Alert` message if one of the TLS 1.3 requirements textually described in the draft TLS 1.3 handshake [IET18] is not respected. As an example, we describe here three requirements taken from the draft TLS 1.3 handshake [IET18]:

- The handshake messages should be sent in one of the orders represented in a path of the state machines given in Figure 4.1 and Figure 4.2. If a message is sent in the wrong order, the handshake connection will be aborted with an "`unexpected_message`" alert.
- The TLS 1.3 handshake refuses renegotiation without a `hello retry request` message, thus the `client hello` message can only be exchanged in the beginning of the protocol or after receiving a `hello retry request` message. If renegotiation takes place without a `hello retry request` message, the handshake is aborted with an "`unexpected_message`" alert.
- When the client receives a `hello retry request` message, the client should check that cryptographic information contained in the `hello retry request` is different from the information in the initial `client hello` message. If not, the handshake is aborted with an "illegal_parameter" alert.

The difficulty in modelling the TLS was the extraction of definitions from the informal definition of the TLS 1.3 handshake requirements in [IET18]. Since this informal specification is not self-contained, it refers to many documents, e.g., the alert management.

In the sequel, we will not consider the internal processing of the TLS handshake but we will focus instead on the TLS messages, leaving the information exchange to be handled by the execution framework and the SUT. In the next section, we detail the handshake messages exchanged.

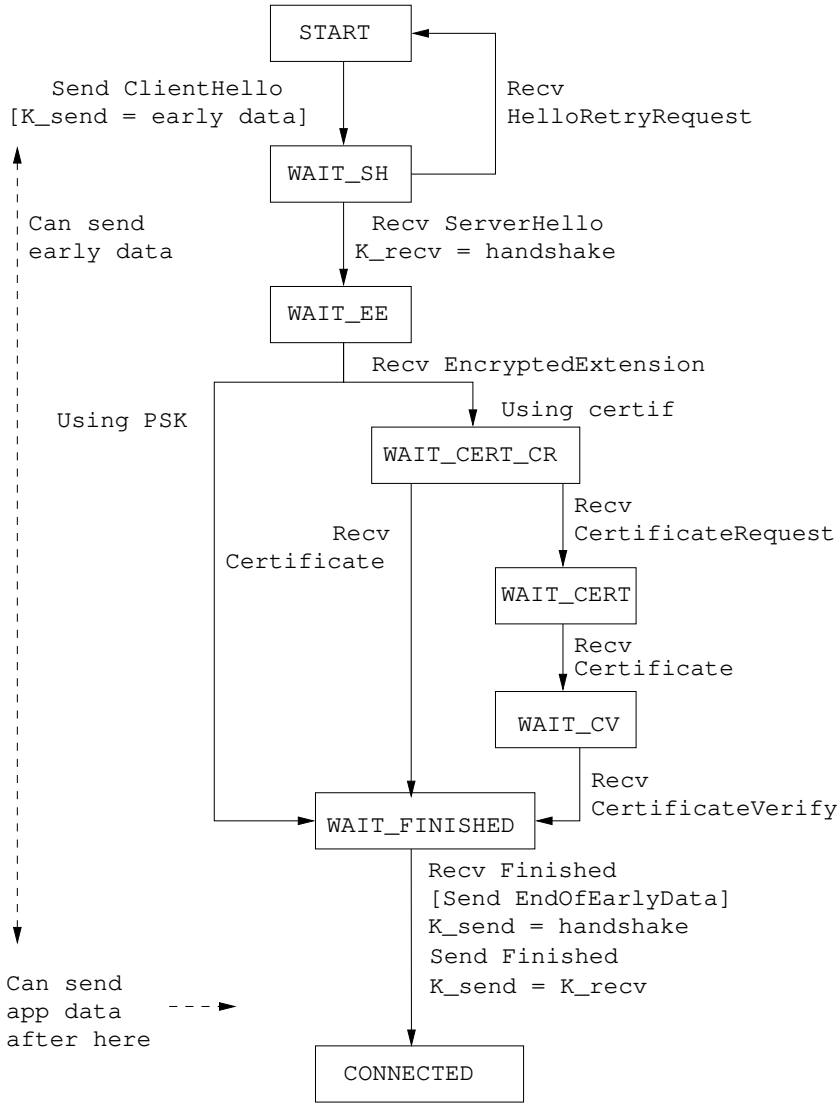


Figure 4.1: TLS handshake client

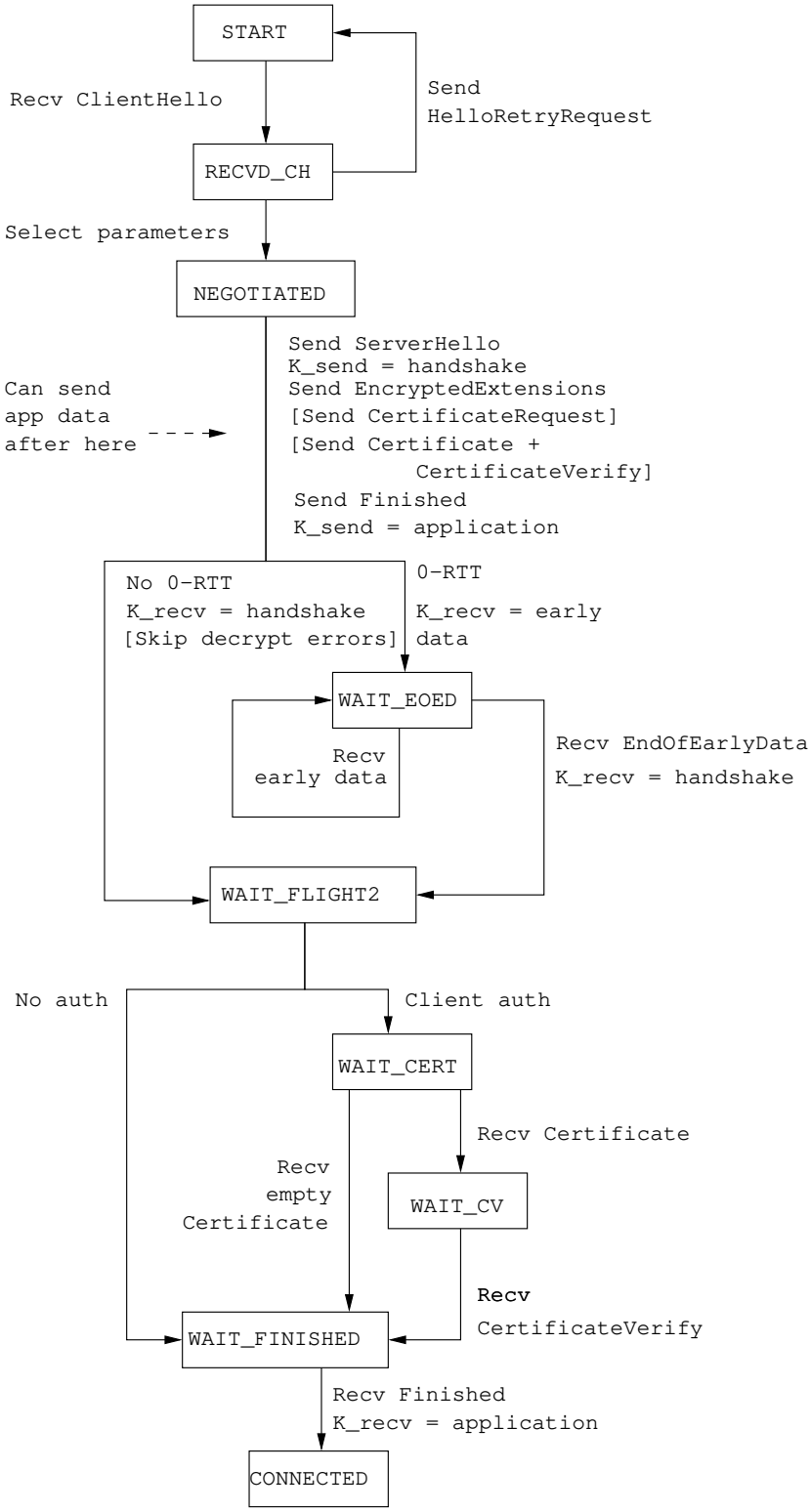


Figure 4.2: TLS handshake server

4.1.2 Main TLS 1.3 handshake messages

The handshake itself consists of different message types, so-called TLS messages, and corresponding parameters that are part of these messages. Every such parameter comprehends specific values, where some of them are assigned dynamically during the handshake procedure. The main messages exchanged during the TLS handshake steps are:

i. The client and the server agree upon the version of the protocol and cryptographic algorithms to use by exchanging the `client hello`, `hello retry request`, `server hello`, and `encrypted extensions` messages.

   - The `client hello` is always the first message, and a client should resend a `client hello` message only if the server responded to it by a `hello retry request` message. It contains the client's cryptographic information: the supported version of protocol, the pre-shared keys, the list of symmetric cipher options, and the extended functionalities.
   - The `server hello` is the response (message) from the server to the `client hello` message if the server was able to select an acceptable set of handshake parameters based on the `client hello`. It contains the cryptographic information selected: the protocol version, the list of symmetric cipher, and the server extensions.
   - The `hello retry request` is the response (message) from the server to the `client hello` message if the server was not able to select an acceptable set of parameters. It contains the same cryptographic information as the `server hello` message.
   - The `encrypted extensions` message is sent by the server immediately after the `server hello` message. This is the first message that is encrypted using keys derived from the server_handshake_traffic_secret. It contains extensions that can be protected.

ii. An authentication with a certificate between the server and the client can be requested by exchanging the `certificate request`, `certificate`, and `certificate verify` messages.

   - The `certificate request` message must be sent by the server directly after the `encrypted extensions` message if the server requests a certificate. It contains an identification of the certificate request and a set of extensions describing the parameters of the certificate.
   - The `certificate` message is sent by the server and by the client. It contains their respective certificate to be used for authentication, and any other supporting certificates.
   - The `certificate verify` message is directly sent by the server and by the client after the respective certificate message. It contains a signature using the private

key corresponding to the public key in the `certificate` message.

iii. The server and the client generate and share a secret key by exchanging the `finished` message. The `finished` message is sent by the server, then by the client. It contains a message authentication code (MAC).

iv. At any step of the handshake, the client or the server can abort the handshake, and close the connection if a failure happened. To do so, they should exchange an `alert` message. The `alert` message contains the description of the alert.

A number of TLS handshake messages contain encoded extensions. There are 21 types of extensions defined in [IET18] (e.g., "supported_versions", "cookie", "negotiated groups", etc.). For instance, the "supported_versions" extension is a list of supported versions in preference order, and it is used by the client to indicate which versions of TLS it supports and by the server to indicate which version it is using. The client sends its extension requests in the `client hello` message and the server sends its responses in the `server hello` extension, `encrypted extensions`, `hello retry request` extensions and `certificate` messages.

4.2 Formal Model of the TLS Handshake in LNT

Taking the draft specification of the TLS [IET18] protocol version 1.3 as starting point, we formalize the handshake protocol of TLS in the LNT language [GLS17, CCG+19]. In this section, we give a brief description of our model, which encompasses the handshake messages, and illustrate then the handshake interactions. We discuss the challenges to specify the TLS handshake in LNT with concrete examples. The full LNT model is given in [BMMW18][1].

4.2.1 Handshake interactions

Our modeling of the communication between client and server is based on the state machines (Figures 4.1 and 4.2) and the handshake TLS 1.3 requirements discussed in Section 4.1.1.

The server and the client are modeled by two processes [BMMW18] communicating by *rendezvous* on gates, i.e., the communication is blocked by both sending and receiving messages: the one waiting for a rendezvous is suspended and terminates immediately after the rendezvous takes place. Concretely, the server and the client processes correspond to their respective state machines (Figure 4.1 and Figure 4.2) extended with the management of the `Alert` message. Each kind (server or client) of handshake message is implemented

[1] `http://mars-workshop.org/repository/018-TLS.html`

as a process, and two additional processes by kind of handshake message sent by the client and the server. In total there are 15 processes.

The alert management is in charge of handling handshake errors. If a handshake requirement is not respected, the handshake should be aborted with an alert message. We defined the following *AlertType*, an enumeration of all possible alert messages. Each process takes as parameter an alert type, which is initialized by an "undefined" alert in the main process. If a handshake requirement is not respected in a process, the corresponding alert type should be assigned to the out alert parameter, and the handshake aborted with the corresponding alert message.

```
type AlertType is
  missing_extension,
  unexpected_message,
  unsupported_certificate,
  ...
  undefined
end type
```

During the modeling process of the handshake interactions, we took advantage of the powerful disruption operation of the LNT language. Consider for instance the following requirement: the TLS 1.3 handshake refuses renegotiation without a `hello retry request` message. The `client hello` message can only arrive at the beginning of the handshake, or right after a `hello retry request` message. In all other cases, if a `client hello` message arrives, the handshake should be aborted with an alert. To implement this requirement we used the LNT operator **disrupt**, which allows at any time a possible disruption of a block of code by another block. In the following LNT code, we have for instance the possible disruption of a `content` behavior by a `client hello` message (`ClientHello`), followed by an alert.

```
disrupt
   ... content
by
  -- TLS 1.3 refuses renegotiation without a Hello Retry Request
  ClientHello [clientHello_c] (false, !?CH_p, HRR_P, ?alert);
  alert := unexpected_message;
  -- abort the handshake with an "unexpected_message" alert
  alert_c (alert)
end disrupt
```

4.2.2 Handshake messages

The handshake messages and their encryption information are defined as types. Our model contains 43 types, with simple types as enumerations, lists, or more sophisticated ones, such

as "union like" types. Below is an example of structure-like type containing one constructor with several fields.

```
type ClientHello is
  ClientHello (legacy_version: ProtocolVersion, random: Random32,
    legacy_session_id: SessionId, cipher_suite: Ciphers,
    legacy_compression_methods: CompressionMethods,
    extensions: Extensions)
end type
```

The main challenge was the definition of "union-like" types, regrouping several types, the difficulty being to build this abstract tree of types from the informal TLS handshake description. The advantage is that, once this was done, it became easy to bring new extensions to our model. For instance, one of the encryption parameters of the `client hello` message is of type `Extensions`, which is a list of elements of type `Extension`, i.e., a tuple containing an extension type and an extension data.

```
type Extensions is
  list of Extension
with "cons", "remove"
end type
```

```
type Extension is
  Extension (extension_type: ExtensionType,
             extension_data: ExtensionData)
end type
```

LNT provides some predefined functions, which simplify the modeling task by avoiding the definition of classical useful functions. For instance, the LNT definition of the `client hello` message described in Section 4.1.2, uses three predefined functions to support notation x.f and compare values of the ClientHello type (`cons` and `remove`). `Extension` type is defined by an enumeration of 21 `extension` types. `Extension data` is a type regrouping several constructors, each of them having its own parameters and corresponding to an extension type. Most of the extensions are optional, thus we implemented 9 of the 21 respective extension data constructors in our model.

```
type ExtensionType is
  signature_algorithms,
  supported_versions,
  cookie,
  ...
end type
```

```
type ExtensionData is
  Cookie (c: Cookie),
  CertificateType (ct: CertificateType),
  SupportedVersions (sv: supportedVersions),
  ...
end type
```

Consider for instance the definition of one of the mandatory extensions in TLS 1.3, the supported version extension. We want to model a `supported-version extension` for the protocol "TLS 1.2" in LNT. To do so, we need an extension with the supported version type, and with an extension data using the constructor `SupportedVersions`, which takes as a parameter a `supportedVersions`, i.e., a list of protocol versions.

```
Extension(                        type SupportedVersions is      type ProtocolVersion is
  supported_version,                list of ProtocolVersion        TLS12,
  SupportedVersion ({TLS12})      end type                         ...
)                                                                end type
```

Our complete LNT model of the TLS handshake 1.3 (see in [BMMW18, Appendix 16]) has 918 lines and contains 15 processes, 8 functions, 42 types, and 10 channel definitions.

4.3 TESTOR: On-the-Fly Conformance Test Case Generation

In this section, we present TESTOR[2], a tool for on-the-fly conformance test case generation guided by test purposes. Following the approach of TGV [JJ05], a test purpose characterizes some state(s) of the model as accepting. TESTOR extends the algorithms of TGV to extract controllable test cases completely on the fly (i.e., during test case execution against the SUT), making TESTOR suitable for online testing. The tool is built following a modular architecture on top of the OPEN/CAESAR [Gar98] generic environment for on-the-fly graph manipulation provided by CADP [GLMS13].

4.3.1 Architecture

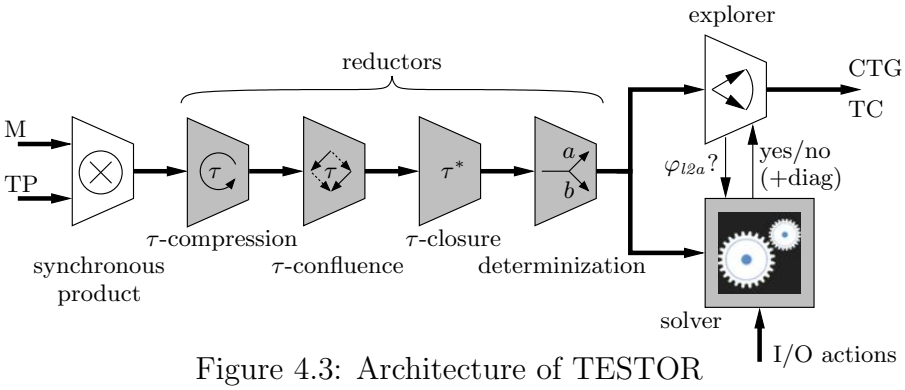


Figure 4.3: Architecture of TESTOR

TESTOR takes as input a formal model (M), a test purpose (TP), and a predicate specifying the input/output actions of M. Depending on the chosen options, it produces as output either a complete test graph (CTG), or a test case (TC) extracted on the fly. TESTOR has a modular component-based architecture consisting of several on-the-fly IOLTS transformation components, interconnected according to the architecture shown on Figure 4.3.

[2]The TESTOR tool is available at http://convecs.inria.fr/software/testor

The boxes represent transformation components and the arrows between them denote the implicit representations (*post* functions) of IOLTSs. The synchronous product and the explorer are the only components newly developed, all the other ones (represented in gray on Figure 4.3) being already available in the libraries of the OPEN/CAESAR [Gar98] environment of CADP.

The first component produces the synchronous product (SP) between the model M and the test purpose TP. Following the conventions of TGV [JJ05], the synchronous product supports $*$-transitions and implements the implicit addition of self-looping $*$-transitions. Note that the synchronous product is optional, one can instead use the multiway rendezvous [Hoa78, HP06, GS17] to compositionally annotate the model. The next four reduction components progressively transform SP into $\mathrm{SP}_{vis} = det(\Delta(\mathrm{SP}))$ as follows:

(i) τ-compression produces the suspension automaton $\Delta(\mathrm{SP})$ by squeezing the strongly connected components of τ-transitions and replacing them with δ-loops representing quiescence;

(ii) τ-confluence eliminates redundant interleavings by giving priority to confluent τ-transitions, i.e., whose neighbor transitions (going out from the same source state) do not bring new observational behavior;

(iii) τ-closure computes the transitive reflexive closure on τ-transitions;

(iv) the resulting τ-free IOLTS is determinized by applying the classical subset construction.

The reduction by τ-compression is necessary for τ-confluence (which operates on IOLTSs without τ-cycles) and is also useful as a preprocessing step for τ-closure (whose algorithm is simpler in the absence of τ-cycles). Although τ-confluence is optional, it may reduce drastically the size of the IOLTS prior to τ-closure, therefore acting as an accelerator for the whole test selection procedure when SP contains large diamonds of τ-transitions produced by the interleavings of independent actions [Mat05]. The first three reductions [Mat05] are applied only if TESTOR detects the presence of τ-transitions in SP.

The determinization produces as output the post function of the IOLTS SP_{vis}, whose states correspond to sets of states of the τ-free IOLTS produced by τ-closure. SP_{vis} is processed by the explorer component, which builds the CTG or the TC by computing the corresponding subgraph whose states are contained in L2A (*lead to accept*). The reachability of accepting states is determined on the fly by evaluating the PDL [FL79] formula $\varphi_{l2a} = \langle \mathsf{true}^* \rangle \mathit{accept}$ on the states visited by the explorer, where the atomic proposition *accept* denotes the accepting states. This check is done by translating the verification problem into a Boolean equation system (BES) and solving it on the fly using a BES solver component [Mat06].

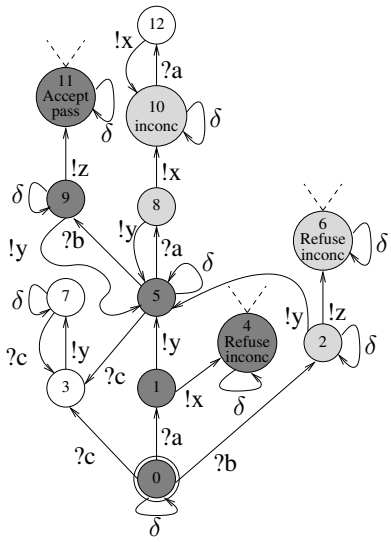


Figure 4.4: The visible behavior SP_{vis}, complete test graph CTG (gray), and a test case TC (dark gray) of the running example [JJ05].

4.3.2 On-the-fly test selection algorithm

We describe below the algorithm used by the explorer component to extract the CTG or a (controllable) TC from the SP_{vis} IOLTS on the fly.

Basically, the CTG is the subgraph of SP_{vis} containing all states in L2A, extended with some states denoting verdicts. The accepting states (which are by definition part of L2A) correspond to pass verdicts. For every state $q \in$ L2A, the output transitions $q \xrightarrow{!a} q'$ with $q' \notin$ L2A lead to inconclusive verdicts, and the output transitions other than those contained in SP_{vis} lead to fail verdicts. To compute the CTG, the explorer component performs a forward traversal of SP_{vis} and keeps the states $q \in$ L2A, which satisfy the formula φ_{l2a}. The check $q \models \varphi_{l2a}$ is done by solving the variable X_q of the minimal fixed point BES $\{X_q = (q \models accept) \vee \bigvee_{q \xrightarrow{b} q'} X_{q'}\}$ denoting the interpretation of φ_{l2a} on SP_{vis}. The resolution is carried out on the fly using the algorithm for disjunctive BESs proposed in [Mat06]. If the CTG is not empty (i.e., $q_0^{\mathrm{SP}_{vis}} \models \varphi_{l2a}$), then it contains at least one controllable TC [JJ05].

The extraction of a TC uses a similar forward traversal as for generating the CTG, extended to ensure controllability, i.e., every state q of TC either has only one outgoing input transition $q \xrightarrow{?a} q'$ with $q' \in$ L2A, or has all output transitions $q \xrightarrow{!a} q''$ of SP_{vis} with $q'' \in$ L2A. The essential ingredient for selecting the input transitions on the fly is the diagnostic generation for BESs [Mat00], which provides, in addition to the Boolean value of a variable, also the minimal fragment (w.r.t. inclusion) of the BES illustrating the value of that variable. For a variable X_q evaluated to true in the disjunctive BES underlying φ_{l2a}, the diagnostic (witness) is a sequence $X_q \xrightarrow{b_1} X_{q_1} \xrightarrow{b_2} \cdots \xrightarrow{b_k} X_{q_k}$ where $q_k \models accept$.

This induces a sequence of transitions $q \xrightarrow{b_1} q_1 \xrightarrow{b_2} \cdots \xrightarrow{b_k} q_k$ in SP_{vis} leading to an accepting state. Since all states $q, q_1, ..., q_k$ also belong to L2A, this diagnostic sequence is naturally part of the TC under construction.

More precisely, the TC extraction algorithm works as follows. If $q_0^{\mathrm{SP}_{vis}} \models \varphi_{l2a}$, the diagnostic sequence for $q_0^{\mathrm{SP}_{vis}}$ is inserted in the TC (otherwise the algorithm stops because the CTG is empty). For the TC illustrated on Figure 4.4, this first diagnostic sequence is $q_0 \xrightarrow{\mathrm{?a}} q_1 \xrightarrow{\mathrm{!y}} q_5 \xrightarrow{\mathrm{?b}} q_9 \xrightarrow{\mathrm{!z}} q_{11}$. Then, the main loop consists in choosing an unexplored transition of the TC and processing it.

- If it is an input transition $q \xrightarrow{?a} q'$, nothing is done, since the target state $q' \in$ L2A by construction. Furthermore, the presence of this transition in the TC makes its source state q controllable. This is the case, e.g., for the transition $q_0 \xrightarrow{\mathrm{?a}} q_1$ in the TC shown on Figure 4.4.

- If it is an output transition $q \xrightarrow{!a} q'$, each of its neighboring output transitions $q \xrightarrow{!a'} q''$ is examined in turn. If the target state $q'' \notin$ L2A, the transition is inserted in TC and q'' is marked with an inconclusive verdict. This is the case, e.g., for the transition $q_1 \xrightarrow{\mathrm{!x}} q_4$ in the TC on Figure 4.4. If $q'' \in$ L2A, the transition in inserted in the TC, together with the diagnostic sequence produced for q''. This is the case, e.g., for the transition $q_9 \xrightarrow{\mathrm{!y}} q_5$ in the TC on Figure 4.4.

The insertion of a diagnostic sequence in the TC stops when it meets a state q that already belongs to the TC, since by construction the TC already contains a sequence starting at q and leading to an accepting state. This is the case, e.g., for the diagnostic sequence starting at state q_5 in the TC on Figure 4.4. In this way, the TC is built progressively by inserting the diagnostic sequences produced for each of the encountered states in L2A.

During the forward traversal of SP_{vis}, the explorer component continuously interacts with the BES solver, which in turn triggers other forward explorations of SP_{vis} to evaluate φ_{l2a}. The repeated invocations of the solver have a cumulated linear complexity in the size of the BES (and hence, the size of SP_{vis}), because the BES solver keeps its context in memory and does not recompute already solved Boolean variables [Mat06].

4.3.3 Examples of different ways to express a test purpose

A natural TP for the TLS handshake is to search for a sequence corresponding to the exchange of handshake messages between the client and the server, for instance, the client sends a `client hello` message. If the server is not satisfied with this request, it responds with a `hello retry request`, and the client responds with a new `client hello` message. Finally, if the server is satisfied, it responds with a `server hello` message. Using the LNT language [GLS17, CCG+19], one would be tempted to write this TP as the process

PURPOSE_A below, simply containing the desired sequence of the four handshake messages (on gates CLIENT_HELLO, HELLO_RETRY_REQUEST, CLIENT_HELLO, and SERVER_HELLO):

```
process PURPOSE_A [CLIENT_HELLO: CH, HELLO_RETRY_REQUEST: HRR,
                   SERVER_HELLO: SH, T_ACCEPT: none] is
  CLIENT_HELLO (CTLS12, 28byteRand, T_NULL, {}, T_NULL, {});
  HELLO_RETRY_REQUEST (TLS12, TLS_AES_128_GCM_SHA256,
              {Extension (signature_algorithms, SignatureSchemeList (sas))});
  CLIENT_HELLO (CTLS12, 28byteRand, T_NULL, {TLS_AES_128_GCM_SHA256}, T_NULL,
              {Extension (signature_algorithms, SignatureSchemeList (sas))});
  SERVER_HELLO (TLS12, 28byteRand, TLS_AES_128_GCM_SHA256,
              {Extension (signature_algorithms, SignatureSchemeList (sas))});
  loop T_ACCEPT end loop
end process
```

Following the conventions of TGV, we mark accepting (respectively, refusal) states by a self-loop labeled with T_ACCEPT (respectively, T_REFUSE).

However, PURPOSE_A is not complete: e.g., initially only one action out of the possible set {CLIENT_HELLO(), HELLO_RETRY_REQUEST (...), SERVER_HELLO (...), ...} is specified. Thus, when computing the synchronous product with the model, PURPOSE_A is implicitly completed by self-loops labeled with "*" (as explained in Section 2.3.3), yielding a significantly more complex TC than expected. For instance, the implicit *-transition in the initial state allows the tester to perform the sequence "CLIENT_HELLO (...); HELLO_RETRY_REQUEST (...); CLIENT_HELLO (...); HELLO_RETRY_REQUEST (...); CLIENT_HELLO (...); SERVER_HELLO (...)" rather than the expected simple sequence "CLIENT_HELLO (...); HELLO_RETRY_REQUEST (...); CLIENT_HELLO (...); SERVER_HELLO (...)". To force the generation of a TC corresponding to the simple sequence, it is necessary to explicitly complete the TP with transitions to refusal states, as shown by the LNT process PURPOSE_B, where gate OTHERWISE stands for the special label "*":

```
process PURPOSE_B [CLIENT_HELLO: CH, HELLO_RETRY_REQUEST: HRR,
                   SERVER_HELLO: SH, T_ACCEPT, T_REFUSE, OTHERWISE: none] is
   select -- refuse any rendezvous but "CLIENT_HELLO (...)"
      CLIENT_HELLO (CTLS12, 28byteRand, T_NULL, {}, T_NULL, {})
   [] OTHERWISE; loop T_REFUSE end loop
   end select;
   select -- refuse any rendezvous but "HELLO_RETRY_REQUEST (...)"
      HELLO_RETRY_REQUEST (TLS12, TLS_AES_128_GCM_SHA256,
              {Extension (signature_algorithms, SignatureSchemeList (sas))})
   [] OTHERWISE; loop T_REFUSE end loop
   end select;
   select -- refuse any rendezvous but ``CLIENT_HELLO (...)''
     CLIENT_HELLO (CTLS12, 28byteRand, T_NULL, {TLS_AES_128_GCM_SHA256}, T_NULL,
              {Extension (signature_algorithms, SignatureSchemeList (sas))})
```

```
   [] OTHERWISE; loop T_REFUSE end loop
   end select;
   select -- refuse any rendezvous but "SERVER_HELLO (...)"
     SERVER_HELLO (TLS12, 28byteRand, TLS_AES_128_GCM_SHA256,
               {Extension (signature_algorithms, SignatureSchemeList (sas))});
     loop T_ACCEPT end loop
   [] OTHERWISE; loop T_REFUSE end loop
   end select
end process
```

The modular architecture of TESTOR also makes the description of test purposes more convenient, replacing the specific synchronous product of TGV by the LNT parallel compositions and taking advantage of the multiway rendezvous, a powerful primitive to express communication and synchronization among a set of distributed processes. This approach based on the multiway-rendezvous supports data handling, something which is necessary for defining test purposes to validate. Instead of using the dedicated synchronous product, it is also possible to take advantage of the multiway rendezvous [Hoa78, GS17] to compositionally annotate the model, relying on the LNT operational semantics [CCG+19, Appendix B] to cut undesired branches. For instance, the same effect as the synchronous product with `PURPOSE_B` can be obtained by skipping the left-most component "synchronous product" of Figure 4.3, i.e., feeding the τ-reduction steps with the IOLTS described by the following LNT parallel composition:

```
par CLIENT_HELLO, HELLO_RETRY_REQUEST, SERVER_HELLO in
   SERVER [CLIENT_HELLO, HELLO_RETRY_REQUEST, SERVER_HELLO]
|| PURPOSE_A [CLIENT_HELLO, HELLO_RETRY_REQUEST, SERVER_HELLO, T_ACCEPT]
end par
```

Note that `SERVER` denotes a process implementing the Server behavior.

This approach based on the multiway rendezvous even supports data handling. For instance, to receive the hello request message (variable `H`), and to use the information directly in the client hello message (`H.cipher_suite` and `H.extensions`), one has just to replace in the above parallel composition the call to `PURPOSE_A` by a call to the process `PURPOSE_C`:

```
process PURPOSE_C [CLIENT_HELLO: CH, HELLO_RETRY_REQUEST: HRR,
                   SERVER_HELLO: SH, T_ACCEPT: none] is
   var H: HelloRetryRequest, S: ServerHello in
      CLIENT_HELLO (CTLS12, 28byteRand, T_NULL, {}, T_NULL, {});
      HELLO_RETRY_REQUEST (?H);
      CLIENT_HELLO (CTLS12, 28byteRand, T_NULL,
                  H.cipher_suite, T_NULL, H.extensions);
      SERVER_HELLO (?S);
      loop T_ACCEPT end loop
   end var
end process
```

Table 4.1: Run-time performance for selected examples

example	TESTOR				TGV			
	test case		CTG		test case		CTG	
	time	mem.	time	mem.	time	mem.	time	mem.
EnergyBus	3	81	182	181	2	137	52	858
EnergyBus (with REFUSE)	1	67	1	66	0	66	0	43
ACE UniqueDirty	45	121	346	451	75	159	3047	643
ACE SharedDirty	384	510	342	529	3821	746	3920	746
ACE SharedClean	298	415	325	523	2820	628	3474	663
ACE Data Inconsistency	24	116	580	711	24	142	6701	894
DES	22109	300	>1week		>43GB		>220GB	
DES (with REFUSE)	27344	332	27	86	24	6177	24	6176
DES (with data)	2	74	4	100	not applicable			

Execution time is given in seconds and memory usage in MB.

4.3.4 Experimental comparison of TESTOR and TGV

TESTOR follows TGV's implementation of the **ioco**-based testing theory [Tre92, Tre08], using the same IOLTS processing steps, adding only the τ-confluence reduction. For each step, TESTOR uses components developed, tested, and used in other tools for more than a decade. We focus on performance aspects and we compare TESTOR to TGV. For this purpose, we conducted several experiments with models and test purposes, both automatically generated and drawn from academic examples and realistic case studies. In this section we present a summary of the evaluation process and experimentation (more details are available in [MMS18]).

For assessing the correctness of TESTOR, we checked that each TC is included in the CTG, and we compared the TCs and CTGs generated by TESTOR to those generated by TGV. For each pair of model and TP, we measured the runtime and peak memory usage of computing a TC or CTG (using TESTOR and TGV), excluding the fixed cost of compiling the LNT code (model and TP) and generating the executable. The experiments presented in this chapter were carried out using the `petitprince` cluster located in Luxembourg [3], each machine of which is equipped with 2 Intel Xeon E5-2630L CPUs, 32GB RAM, and running 64-bit Debian GNU/Linux 8 and CADP 2017-i. Each measurement corresponds to the average of ten executions.

Test Purposes taken from case studies

Table 4.1 summarizes the results for some selected examples. The first two have been kindly provided by Alexander Graf-Brill, and correspond to initial versions of TPs for his EnergyBus model [GBHG14]; both aim at exhibiting a particular boot sequence, the second one using `REFUSE` transitions. The next four examples have been used by STMicroelectronics to verify a cache-coherence protocol [KS15]. The last three correspond to the three test purposes for an asynchronous implementation of the DES (Data Encryption Standard) [Ser15] and check the correctness of a simplified[4] version of the asynchronous implementation of the DES (Data Encryption Standard). These examples cover a large spectrum of characteristics: from no τ-transitions (ACE) to huge confluent τ-components (DES), from few visible transitions (DES) to many outgoing visible transitions (EnergyBus), and a test selection more or less guided via refusal states.

We observe that TESTOR requires less memory than TGV for all examples, but most significantly for the DES. However, although TESTOR is several orders of magnitude slower than TGV for the DES when using the synchronous product (DES TPs without and with REFUSE), TESTOR requires only two seconds to generate a TC or CTG when using an LNT parallel composition with the DES TP with data handling. This is because the LNT parallel composition, handled by the LNT compiler, enables more aggressive optimizations. Thus, using LNT parallel composition to annotate the model's accepting and refusal states is not only more convenient (thanks to the multiway rendezvous) and data aware, but also much more efficient — it is even possible to generate a TC for the original DES model (167 million states, 1.5 billion transitions) in less than 40 minutes.

For the ACE examples, TESTOR is both faster and requires less memory than TGV. This is partly due to an optimization of TESTOR, which deactivates the various reductions of τ-transitions. For a fair comparison, we also run experiments forcing the execution of these reductions. For the extraction of a TC, this increases the execution time by a factor of two and the memory requirements by a factor of three. For the computation of a CTG, this increases the memory requirements by a factor of one and a half, without modifying the execution time significantly.

[3]The `petit prince` cluster is part of the Grid'5000 testbed, which is supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

[4]The S-boxes are executed sequentially rather than in parallel and the gate `SUBKEY` is left visible to separate the iterations of the DES algorithm and thus significantly reduce the size of τ-components. For the extraction of TC for the TP using `REFUSE` transitions, from the full version of the DES, TESTOR would run for several weeks and TGV would require more than 700 GB of RAM.

Automatically generated test purposes

To evaluate the performance, we used a collection of 9791 LTSs with up to 50 million transitions, taken from the non-regression test-base of CADP. For each LTS M of the collection, we automatically generated two TPs: one to test the reachability of an action and another to test the presence of an execution sequence. For the former TP, we sorted the actions of the LTS alphabetically, and checked the reachability of the first action, considering the second half of the action set as inputs. For the latter TP, we used the EXECUTOR tool[5] to extract a sequence of up to 1000 visible actions, which we transformed into a TP, considering all actions whose ranking is an odd number as inputs. Technically, this transformation consists in adding to each state of the sequence a self-loop labeled with τ and a $*$-transition to a refusal state. From the generated pairs (M, TP) we eliminated those for which the automatic generation of a TP failed (for instance, due to special actions that would require particular treatment) and those for which the computation of a TC or CTG took too much time or required too much memory by either TESTOR or TGV. This led to a collection of 13,142 pairs (M, TP) for which both tools could extract a TC. For 12,654 of them, both tools also could compute the CTG.

As for the case studies, we observe that TESTOR and TGV choose different tradeoffs between computation time and memory requirements. On average, TESTOR requires 0.3 times less memory and runs 1.3 (respectively 0.5) times faster to compute a TC (respectively the CTG). When considering only the 1005 pairs with more than 500,000 transitions in the LTS, the average numbers show a larger difference. On average for these larger examples, to compute a CTG, TESTOR requires 1.4 times less memory, but runs 3.5 times longer; to compute a TC, TESTOR requires 2.7 times less memory and runs 0.7 times faster.

Also, while both tools required the exclusion of examples due to excessive runtime, we excluded several examples due to insufficient memory for TGV, but not for TESTOR. Given that TCs are usually much smaller than CTGs, the on-the-fly extraction of a TC by TESTOR is generally faster and consumes less memory than the generation of the CTG. We also observed that the CTGs produced by TESTOR are sometimes smaller than (although strongly bisimilar to) those produced by TGV. While trying to understand these results in more detail, we found examples where each tool is one or two magnitudes faster or memory-efficient than the other.

Indeed, the benefits of the different reductions applied in the tools depend heavily on the characteristics of the example, most notably the sizes of the various subgraphs explored (τ-components, L2A). For instance, when the model M does not contain any τ-transition, there is no point in applying the reductions (τ-compression, τ-confluence, and τ-closure).

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

4.3.5 Comparison of TESTOR and other MBTs

In the following, we compare TESTOR to the most closely related tools.

TorX [TB03] and JTorX [Bel10] are online test generation tools, equipped with a set of adapters to connect the tester to the SUT. The latest versions support test purposes (TPs), but they are used differently than in TESTOR. Indeed, JTorX yields a two-dimensional verdict [Bel14]: one dimension is the **ioco** correctness verdict (pass or fail), and the other dimension is an indication whether the test objective has been reached. This contrasts with TESTOR, which generates test cases (TCs) ensuring by construction that the execution stays inside the lead to accept states (L2A), and stopping the test execution as soon as possible with a verdict: **fail** if non-conformance has been detected, **pass** if an accepting state has been reached, or **inconclusive** if leaving L2A is unavoidable.

Uppaal is a toolbox for the analysis of timed systems, modeled as timed automata extended with data. Three test generation tools exist for Uppaal timed automata. Uppaal-Tron [LMNS05] is an online test generation tool, taking as input a specification and an environment model, used to constrain the test generation. Uppaal-Tron is also equipped with a set of adapters to derive and execute the generated tests on the SUT. Contrary to TESTOR, the TCs generated from Uppaal-Tron can be irrelevant, because the generation is not guided by TPs. Uppaal-Cover [HP06] generates offline a comprehensive test suite from a deterministic Uppaal model and coverage criteria specified by observer automata. Uppaal-Cover attempts to build a small test suite satisfying the coverage criteria, by selecting those TCs satisfying the largest parts of the coverage criteria. In contrast to TESTOR and Uppaal-Tron, Uppaal-Cover generates offline tests. Offline generation does not face the state-space explosion, but also limits the expressiveness of the specification language (e.g., nondeterministic models are not allowed). Uppaal-Yggdrasil [KLN+15] generates offline test suites for deterministic Uppaal models, using a three-step strategy to achieve good coverage: (i) a set of reachability formulas, (ii) random execution, and (iii) structural coverage of the transitions in the model. The guidance of the test generation by a temporal logic formula is similar to the use of a TP. However, the TPs supported by TESTOR (and TGV) can express more complex properties than reachability, and enable one to control the explored part of the model (using refusal states).

On-the-fly test generation tools also exist for the synchronous dataflow language Lustre [HCRP91], e.g., Lutess [dBORZ99], Lurette [JRB06], and Gatel [MA00]. Contrary to TESTOR, these tools do not check the **ioco** relation, but randomly select TCs, satisfying constraints of an environment description and an oracle.

In IOLTS, actions are monolithic, which does not fit for realistic models that involve data handling. STG (Symbolic Test Generator) [CJRZ02] breaks the monolithic structure of actions, enabling access to the data values, and generates tests on the fly, handling data values symbolically. This enables more user-friendly TPs and more abstract TCs, because not all possible values have to be enumerated. However, the complexity of symbolic com-

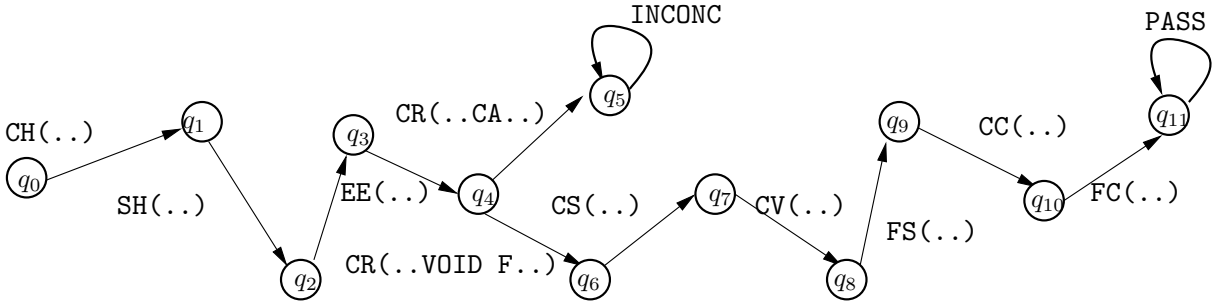


Figure 4.5: TC I: TLS handshake with classical TLS 1.3 order

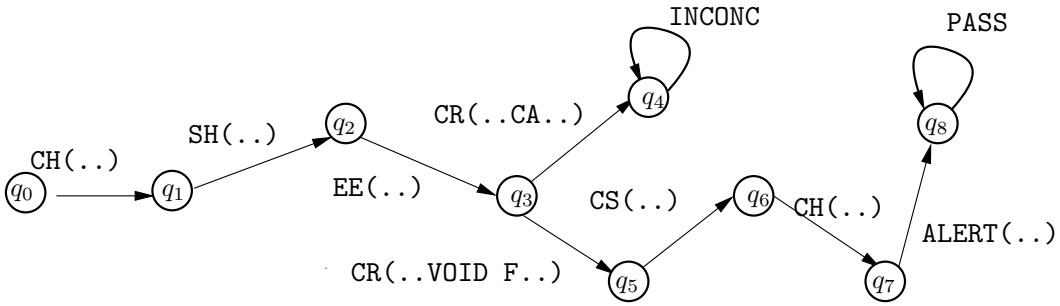


Figure 4.6: TC II: TLS handshake aborted with an unexpected Alert

putation is not negligible in practice. When using the LNT parallel composition, TESTOR can handle data (see example in Section 4.3.3) without the cost of symbolic computation, but still has to enumerate data explicitly when generating the TC. T-Uppaal [MLN04] uses symbolic reachability analysis to generate tests on the fly and then simultaneously executes them on the SUT. The complexity of symbolic algorithms turns out to be expensive for online testing.

4.4 Testing the TLS Handshake Model with TESTOR

We validated our LNT model using TESTOR, taking advantage of its ability to specify data in the test purposes. We defined three test purposes corresponding to three requirements from the draft TLS 1.3 handshake [IET18]:

I. The protocol messages must be sent in the right order, using classical TLS 1.3 order (without `hello retry request` message).

II. The handshake must be aborted with an "unexpected_message" alert, if there is a client renegotiation.

III. The protocol messages are sent in the right order with an incorrect key shared (with `hello retry request` message).

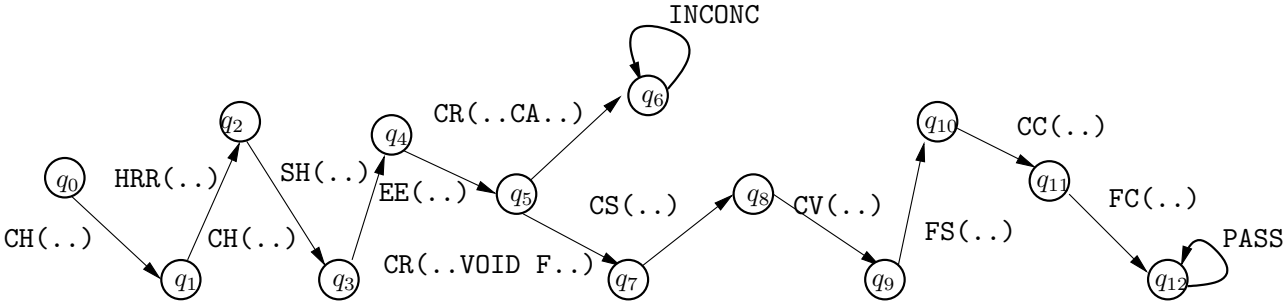


Figure 4.7: TC III: TLS handshake with renegotiation

The LNT models of these test purposes are given in [BMMW18, Appendix B1, B2, B3], respectively. Given our formal LNT model of the TLS handshake and our test purposes, we automatically generated with TESTOR the abstract test cases, represented in a simplified form in Figures 4.5, 4.6, and 4.7 (TC I, TC II, and TC III). We simplified the labels by removing the arguments and replacing the channel names with the following initials: `client hello` (`CH`), `server hello` (`SH`), `hello retry request` (`HRR`), `encrypted extensions` (`EE`), `certificate request` (`CR`), `certificate` (`CS` for the server and `CC` for the client), `certificate verify` (`CV`), and `finished` (`FS` for the server and `FC` for the client). Note that we voluntarily left the initials of the certificate names (`CA` and `VOID F`) for the `certificate request` actions, to keep a deterministic representation of the abstract test cases.

During this validation process we found an error in our model, with test purpose III. We couldn't reach the accepting state of this test purpose, because of our initial implementation of this requirement: "The `server hello` message should have the same cryptographic information as the `hello retry request` message." In fact, we were assigning to the `hello retry request` encryption data the value of the `server hello` encryption data, whereas the `hello retry request` message arrives before the `server hello` message. We changed our model to correct this issue.

⋆⋆⋆

We illustrated the formalization and the functional testing of a communication protocol, through a case study of the TLS handshake 1.3. We presented TESTOR, a new tool for on-the-fly conformance test case generation for asynchronous concurrent systems. Like the existing tool TGV, TESTOR was developed on top of the CADP toolbox [GLMS13] and brings several enhancements: online testing by generating (controllable) test cases completely on the fly; a more versatile description of test purposes using the LNT language; and a modular architecture involving generic graph manipulation components from the OPEN/CAESAR environment [Gar98]. The modularity of TESTOR simplifies maintenance and fine-tuning of graph manipulation components, e.g., by adding or removing on-the-fly reductions, or by replacing the synchronous product. Besides the ability to perform online testing, the on-the-fly test selection algorithm sometimes makes possible the

extraction of test cases even when the generation of the complete test graph (CTG) is infeasible. The experiments we carried out on ten-thousands of benchmark examples and three industrial case studies show that TESTOR consumes less memory than TGV, which in turn is sometimes faster, for generating CTGs.

A combination of this conformance test case generation tool and the synchronous testing techniques presented in Chapter 3 is therefore beneficial for testing GALS systems, as it will be illustrated in Chapter 5.

Chapter 5

Validation of GALS Systems

In this chapter we illustrate formal techniques for GALS systems, using a simple, but relevant example, namely an autonomous car, which has to reach a destination, following roads on a geographical map, in the presence of moving obstacles. The car is modeled as a GALS system, comprising synchronous components for perception, decision, and action. Part of this work has been published in [MMPS19].

This chapter is organized as follows. Section 5.1 presents the car example. Section 5.2 introduces the formal GRL model of an autonomous car. Section 5.3 describes the application of model checking for asynchronous systems to the overall GALS system. Section 5.4 experiments manually the functional testing of synchronous components. Section 5.5 presents the integration of asynchronous conformance testing with automated synchronous testing to improve the latter. Finally, we give concluding remarks on our work and synchronous testing.

5.1 GALS Example: An Autonomous Car

To illustrate our approach, we consider the behavioral model of a (simplified) autonomous car interacting with its environment. The environment consists of a geographical map shared by the car and a given set of moving obstacles (pedestrians, cyclists, other cars, etc.). For instance, the autonomous car and a pedestrian could share the streets of Manchester as it is shown in Figure 5.2. To limit the complexity, each obstacle executes a fixed number of random or statically chosen movements.

The autonomous car shown in Figure 5.1, itself consists of four synchronous components:

(i) a *GPS* keeps the car position updated,

(ii) a *radar* detects the presence of the obstacles close to the car and builds a perception grid summarizing information about perceived obstacles,

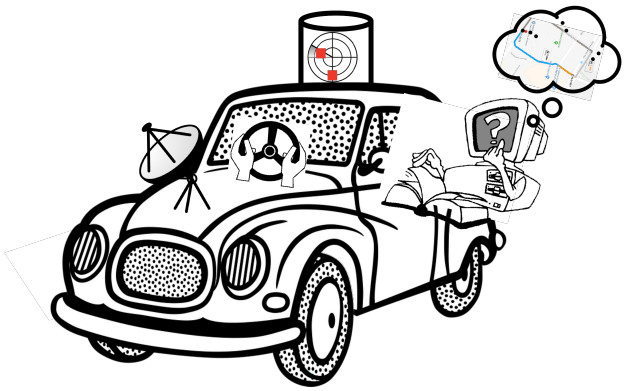


Figure 5.1: Simple autonomous cars

(iii) a *decision* (or trajectory) controller computes an itinerary from the current position to the destination, avoiding streets containing obstacles, and

(iv) an *action* controller commands the engine and direction to follow the itinerary computed by the decision controller, using the perception grid built by the radar to avoid collisions.

These four components communicate in various ways:

- the GPS sends the current position to the decision controller upon request,
- the radar periodically sends the perception grid to the action controller, and
- the action controller requests a new itinerary from the decision controller.

This example is simple, but relevant, because it is representative of a GALS system, with synchronous components (e.g., radar, action controller) interacting with each other, the whole GALS system (e.g., autonomous car) being constrained by a realistic environment (e.g., geographical map and obstacles).

5.2 GRL Model of the GALS System

GRL (GALS Representation Language) [JLM16, Jeb16] is a formal language designed to model GALS systems (more details about GRL are available in Chapter 2 Section 2.4.1). We decided to take advantage of the GRL2LNT [JLM16, Jeb16] translator to LNT [GLS17], which provides a connection to the CADP verification toolbox [GLMS13]. The whole GRL model of the car is presented in the Appendix B. Figure 5.3 shows the architecture of our formal GRL model of the autonomous car. The GRL model reflects the modular specification, connecting the synchronous components by a communication network. Each synchronous component (`ACTION`, `RADAR`, `DECISION`, and `GPS`) is represented in GRL as a

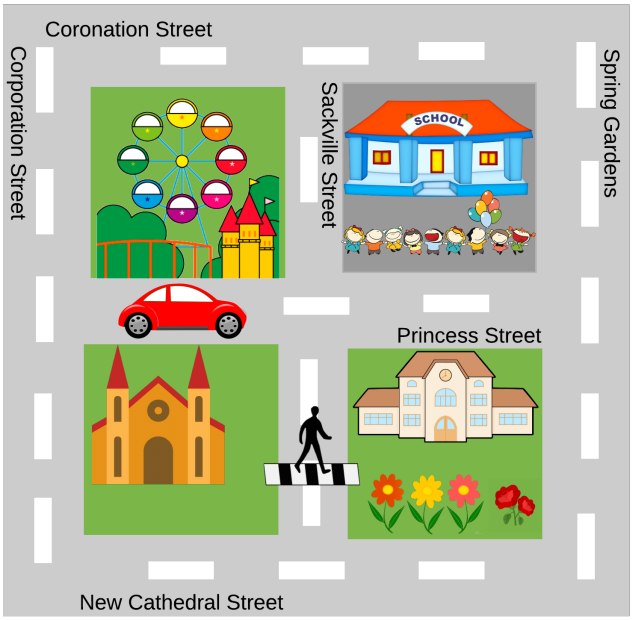


Figure 5.2: Geographical map example with the autonomous car and a pedestrian

block, depicted as a (light blue) rectangle with solid border in Figure 5.3. These blocks exchange data via asynchronous communication media (`POSITION`, `PATH`, `CURRENT_GRID`), each of which is represented in GRL as a **medium**, depicted as a (pink) ellipse with dashed border in Figure 5.3. The interaction between blocks also respects environmental constraints (`MAP_MANAGEMENT`), each one being represented in GRL as an **environment**, depicted as a (light pink) ellipse with thick dashed border in Figure 5.3. The overall model of the GALS is represented in GRL as a **system**, which describes the composition and interactions of blocks, media, and environments. In the sequel, we present excerpts of a block, a medium, an environment, and the system of our GRL model (1189 lines).[1]

The geographical map is represented as a directed graph as illustrated in Figure 5.4, in which edges correspond to streets and nodes correspond to crossroads; for simplicity, we assume that the car or an obstacle occupies a street completely (a longer street can be represented by several edges in the graph). A set of functions is defined to explore this graph, to compute itineraries, etc. For instance, the following LNT constant `initial_map` returns the graph corresponding to the map shown in Figure 5.4.

```
function initial_map : Graph is
  var e: Edges, v: Vertices in
```

[1]The complete GRL model, test purpose, the MCL properties, XTL scripts, and other resources related to the example are available at
http://convecs.inria.fr/software/projex_aQTspX.tgz.

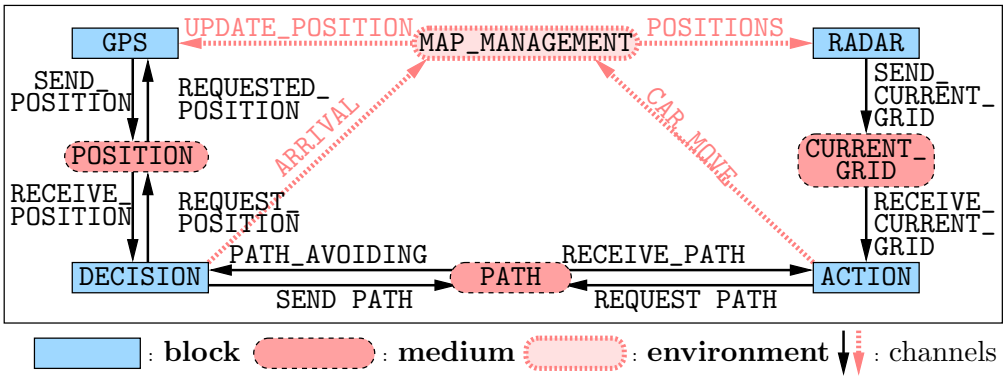


Figure 5.3: Architecture of the GRL model of an autonomous car

```
    v := {0, 1, 2, 3, 4, 5, 6, 7, 8};
    e := {Edge (0, Coronation_Street, 1),
          Edge (0, Corporation_Street, 3),
          Edge (1, Coronation_Street_bis, 0),
          Edge (1, two_Coronation_Street, 2),
          Edge (1, Sackville, 4),
          Edge (2, two_Coronation_Street_bis, 1),
          Edge (2, Spring_Gardens, 5),
          Edge (3, Corporation_Street_bis, 0),
          Edge (3, Princess_Street, 4),
          Edge (3, two_Corporation_Street, 6),
          Edge (4, two_Princess_Street, 5),
          Edge (4, two_Sackville, 7),
          Edge (5, Spring_Gardens_bis, 2),
          Edge (5, two_Princess_Street_bis, 4),
          Edge (5, two_Spring_Gardens, 8),
          Edge (6, two_Corporation_Street_bis, 3),
          Edge (6, New_Cathedral_Street, 7),
          Edge (7, two_Sackville_bis, 4),
          Edge (7, New_Cathedral_Street_bis, 6),
          Edge (7, two_New_Cathedral_Street, 8),
          Edge (8, two_Spring_Gardens_bis, 5),
          Edge (8, two_New_Cathedral_Street_bis, 7)
          };
    return Graph (v, e)
  end var
end function
```

The GRL model is instantiated by providing global constants encoding the map, the initial position and destination of the car, and the set of obstacles with their initial positions and lists of movements.

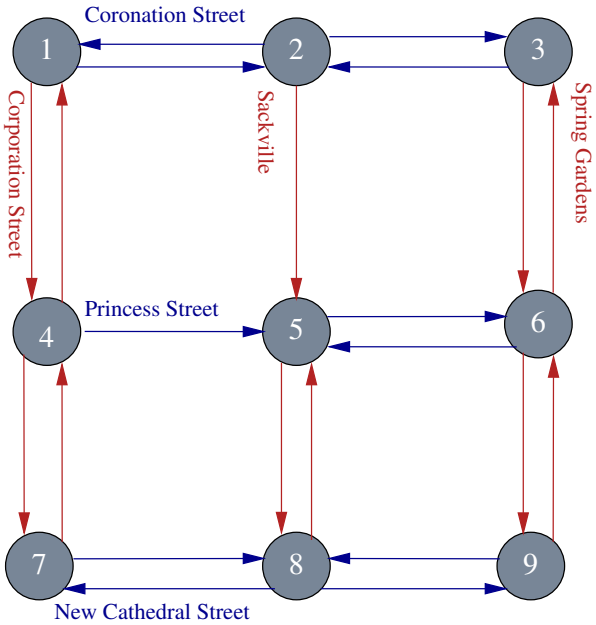


Figure 5.4: GRL geographical map representation

A GRL block defines the deterministic code executed at each *activation* (i.e., clock instant) by the synchronous component. For instance, the radar of our autonomous car is modeled by the GRL block `RADAR`, which has a **static** variable `previous_grid` to keep track of the perception grid computed during the previous activation. This grid is considered to be initially empty (i.e., it has the value `Grid (NIL)`). At each activation, the radar receives the current positions of the car and all the obstacles as input from the environment (**in** parameter `POSITIONS`).[2] It then computes the current perception `grid` indicating, for each possible direction the car might take, whether at least `radar_visibility` steps are free of any obstacle. If there is a change between `previous_grid` and `grid`, both the variable `previous_grid` and the output `CURRENT` are updated; otherwise the output is set to the particular value `already_sent` indicating that the grid did not change. At the end of the activation, the computed grid is sent to the connected medium (**send** parameter `CURRENT`).

```
block RADAR (in POSITIONS: Car_Obstacle_Pos) [send CURRENT: Grid] is
  static var previous_grid: Grid := Grid (NIL)
  var grid: Grid
  grid := perception (POSITIONS, radar_visibility);
  if grid != previous_grid then
    previous_grid := grid;
    CURRENT := grid
  else
    CURRENT := Grid (already_sent)
```

[2]Currently, for the interaction between a block and an environment, GRL requires to wrap several inputs or outputs into a single structured input or output.

```
  end if
end block
```

Synchronous blocks interact with each other via asynchronous communication media. Explicitly representing these media makes it possible to finely model a large panel of behaviors (i.e., message buffering, message loss, nondeterminism, etc.). A medium is connected to each block by at most two channels, called **receive** and **send** channels. Note that a **receive** channel corresponds to the reception of some value in a variable prefixed by "?". Each channel has an associated Boolean condition (tested with a **when** clause), stating whether a message is available. For instance, the following GRL medium `CURRENT_GRID` enables the block `RADAR` to send the current perception grid (via the **receive** channel `SEND_CURRENT_GRID`) to the block `ACTION` (via the **send** channel `RECEIVE_CURRENT_GRID`); the transmission takes place only when the perception grid has not already been sent.

```
medium CURRENT_GRID [receive SEND_CURRENT_GRID: Radar_Grid,
                     send RECEIVE_CURRENT_GRID: Radar_Grid] is
  static var buffer: Radar_Grid := Grid (NIL)
  select
    when ?SEND_CURRENT_GRID ->
      if SEND_CURRENT_GRID != Grid (already_sent) then
         buffer := SEND_CURRENT_GRID end if
  []
    when RECEIVE_CURRENT_GRID -> RECEIVE_CURRENT_GRID := buffer
  end select
end medium
```

Environments provide blocks with inputs and receive their outputs. Block activations are particular inputs, enabling an environment to precisely control the activations of synchronous blocks. For instance, the following fragment of the GRL environment `MAP_MANAGEMENT` ensures that: (1) the geographical map information, such as the position of the car (`map.c`) and obstacles (`grid`), is shared with the block `RADAR` (by sending this information to `RADAR` as input `POSITIONS`); (2) the geographical map information is updated when the car or the obstacles move (by receiving these moves from blocks `ACTION` and `RADAR` as outputs `CAR_MOVE` and `OBSTACLE_MOVE`, respectively); and (3) the blocks are only activated as long as the car did neither arrive at destination, nor crashed. Note that an environment may be nondeterministic, e.g., it may contain nondeterministic choice, modelled in GRL using the **select** statement.

```
environment MAP_MANAGEMENT (block RADAR, ...
                            in OBSTACLE_MOVE: Obstacle,
                            in CAR_MOVE: Control, ...
                            out POSITIONS: Car_Obstacle_Pos, ...) is
  static var grid: Grid := Grid (NIL),
        map: Localization := Localization (initial_street, initial_map),
        crash: Bool := false, car_arrived: Bool := false, ..
```

```
  var collision_detected: Bool, ...
  if not (crash) and not (car_arrived) then
    select
      -- send updated inputs (car and obstacle positions) to the radar
      when POSITIONS -> POSITIONS := pos (map.c, grid)
    []
      -- car movement
      when ?CAR_MOVE ->
        -- update car position in the map
        map := move_car (map, CAR_MOVE);
        -- check for collisions (car and an obstacle on the same street)
        collision_detected := intersection (grid, map);
        if collision_detected then
          crash := true
        end if
    []
      -- potential obstacle movement
      when ?OBSTACLE_MOVE ->
        if OBSTACLE_MOVE != null_obstacle then
          -- update obstacle positions in the grid
          grid := move_obstacle_grid (grid, OBSTACLE_MOVE)
        else
          -- no effective movement
          grid := grid
        end if
    ...
    end select
end environment
```

The following GRL **system** describes the composition of the complete model of the autonomous car, i.e., the blocks `RADAR` and `ACTION`, the medium `CURRENT_GRID`, and the environment `MAP_MANAGEMENT`.

```
system MAIN (SEND_CURRENT_GRID, RECEIVE_CURRENT_GRID: Grid,
             POSITIONS: Car_Obstacle_Pos, REQUEST_PATH: Edges,
             RECEIVE_PATH: Itinerary, CAR_MOVE: Control, ...) is
  block list
    ACTION (?CAR_MOVE) [?REQUEST_PATH,RECEIVE_CURRENT_GRID,RECEIVE_PATH],
    RADAR (POSITIONS) [?SEND_CURRENT_GRID], ...
  medium list
    CURRENT_GRID [SEND_CURRENT_GRID, ?RECEIVE_CURRENT_GRID], ...
  environment list
    MAP_MANAGEMENT (RADAR, OBSTACLE_MOVE, ...
                    ?POSITIONS, ?UPDATE_POSITION, ...)
end system
```

When considering a complete GALS system from the outside, all parts based on the synchronous programming paradigm are hidden. Thus, the overall GALS system is amenable to classic analysis techniques developed for asynchronous systems.

5.3 Model Checking of the GALS Behavior

Using GRL2LNT [JLM16, Jeb16] and CADP, for a geographical map (with 22 streets and 8 crossroads) and two obstacles, each with a first random movement and a second statically chosen movement, we generated (in about 12 minutes on a standard laptop) the LTS corresponding to our GRL autonomous car model (952, 759 states and 1, 518, 227 transitions after strong bisimulation minimization). We first validated our GRL model by checking several safety and liveness properties[3] characterizing the correct behavior of the autonomous car. We expressed the properties in MCL [MT08], which is the data-handling, action-based, branching-time temporal logic of the on-the-fly model checker of CADP (more details about MCL are available in Section 2.3.2). We describe here the properties in natural language and in MCL.

- The position of the car is correctly updated after any movement of the car. This safety property specifies that on all transition sequences, an update of the car position (action "`UPDATE_POSITION ?current_street`", where `current_street` is the street on which the car is) followed by a car movement (action "`CAR_MOVE ?control`", where `control` is a movement command) cannot be followed by an update of the car position inconsistent with `current_street`, `control`, and the map. This can be expressed in MCL using the necessity modality below, which forbids the transition sequences containing inconsistent position updates:

```
[ true* .
  { UPDATE_POSITION ?current_street:String } .
  (not ({ CAR_MOVE ... } or { UPDATE_POSITION ... }))* .
  { CAR_MOVE ?control:String } .
  (not ({ CAR_MOVE ... } or { UPDATE_POSITION ... }))* .
  { UPDATE_POSITION ?new_street:String where
    not (Consistent_Move (current_street, control, new_street)) }
] false
```

  The values of the current position, the movement, and the new position of the car present on the actions `UPDATE_POSITION` and `CAR_MOVE` are captured in the variables `current_street`, `control`, and `new_street` of the corresponding action predicates (surrounded by curly braces) and reused in the **where** clause of the last action predicate. The predicate `Consistent_Move` defines all valid combinations for `current_street`, `control`, and `new_street` allowed by the map.

[3] More details about model checking and temporal properties are available in Section 2.1.3.

- A same message from one of the autonomous car components must be considered only once. For instance, the radar should not send twice the same perception grid, i.e., two successive occurrences of action `SEND_CURRENT_GRID` must carry different values of the grid, reflecting the changes in perception due to obstacle or car movements. This can be expressed by four properties in MCL, one for a `SEND_CURRENT_GRID` with no perceivable obstacles, with one perceivable obstacle, with two perceivable obstacles, and with no updated information (might be repeated). For simplicity, we only give and comment the MCL code of the two perceivable obstacles to illustrate the flavor.[1] This can be expressed in MCL using the necessity modality below, which forbids the transition sequences containing twice the same values for the radar outputs.

```
[ true* .
    (
    { RADAR_OUTPUT ?obstacle1:String ?street1:String
                   ?obstacle2:String ?street2:String } .
    (not { RADAR_OUTPUT ... })* .
    { RADAR_OUTPUT !obstacle1 !street1 !obstacle2 !street2 }
    )
] false
```

- Inevitably (by avoiding the non-progressing iterations of synchronous blocks), the system should reach a state where either the car arrived (`ARRIVED`), or a collision occurred between the car and an obstacle (`COLLISION`), or all obstacles have finished their list of movements (`END_OBSTACLE`). This can be expressed in MCL using the inevitability modality below, which forbids the infinite transition sequences avoiding the non-progressing iterations of synchronous blocks (`NO_UPDATE`) and not leading to one of the three terminal actions (`TERMINATE`).

```
not <(not TERMINATE)*> <(not TERMINATE) and (not NO_UPDATE)>@ ;
```

  The predicate `TERMINATE` defines all valid actions allowed by the map (`ARRIVED`, `COLLISION`, and `END_OBSTACLE`) and the predicate `NO_UPDATE` defines all repeated actions of a block.

We believe that the two last properties are to all GALS systems, because a GALS system contains non-progressing iterations of its synchronous components. Therefore, it is necessary to identify these non-progressing iterations and to verify that the system reaches the terminal states by avoiding these iterations.

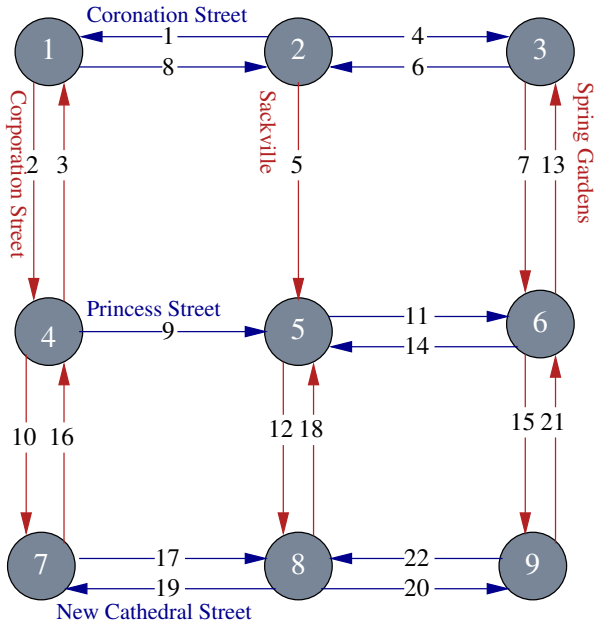


Figure 5.5: Geographical map, with natural numbers identifying the streets fragments.

5.4 Manual Testing of a Synchronous Component

We use the Lurette testing tool[4] [JRB06] to test an implementation of a radar in the C language[5], which might be a part of an implementation of our GALS autonomous car example described in Section 5.2. Usually, a radar builds an occupancy grid with respect to its position (this grid reflects the visibility of the radar); in an autonomous car, the position of the radar is the position of the car. To simplify, in our example the radar takes as input the position of the car and the obstacles (provided by the GRL environment on channel `POSITIONS`) and outputs the perception grid (sent to the medium `CURRENT_GRID`).

In the C radar implementation, the street names of the geographical map presented in Section 5.2 are represented by natural numbers, because the input language of Lutin (the Lurette component for defining test scenarios) supports only Boolean and numerical types. Figure 5.5 gives an overview of this encoding of the geographical map.

As in Section 5.2, we consider a fixed instance, here with two obstacles (called `leo` and `lilly`). Testing the synchronous radar component consists in providing a sequence of inputs and observing the generated sequence of outputs. Each of the radar's inputs (position of the car and the obstacles) can take one out of the 21 streets of the map, yielding 21^3 possible inputs. Fortunately, not all sequences of these inputs are realistic, because the car is not flying and has to respect the constraints of the map and by being fixed on the car, the possible inputs taken by the radar are also constrained by the possible movements

[4]More details about the Lurette tool are available in Section 2.2.3

[5]The radar implementation in C was generated from a Lustre program

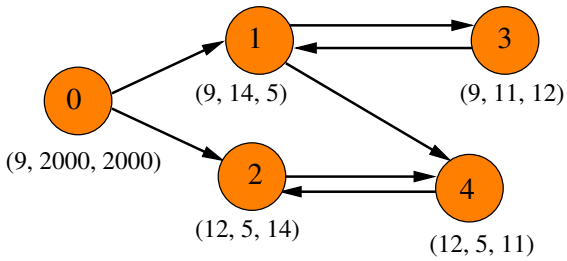


Figure 5.6: Simple scenario automaton

of the car. However, relevant tests of the radar should include situations where the radar detects obstacles.

The input constraints for the radar should enforce that the positions of the car and the obstacles evolve in a realistic manner, i.e., respecting the map. The following Lutin code corresponds to a simple scenario defined by the automaton in Figure 5.6, where the car starts on street 9 and possibly moves to street 12, and lilly (respectively, leo) appears on street 5 (respectively, street 14) and moves back and forth to street 12 (respectively, street 11). The scenario automaton enables to constrain the test inputs generation to those possible in the physical environment (geographical map, and possible movements of the car and obstacles). Each state of the automaton contains a tuple with the three radar inputs values (`car`, `leo`, and `lilly`), the value "2000" indicating that the obstacles are absent. This scenario can be described by an automaton with five states and seven transitions; the corresponding constraints on the inputs of the radar can be encoded in Lutin as a node `input_constraints` with four outputs (the three inputs of the radar plus the state `s` of the automaton). Although simple, the scenario covers the appearance and movement of obstacles, and the case where the perception grid should remain unchanged.

```
node input_constraints () returns (car, leo, lilly, s: int) =
  let not_visible: int = 2000 in
  (* initial state: car on street 9 and no visible obstacles *)
  car = 9 and lilly = not_visible and leo = not_visible and s = 0 fby
  loop {
    | (* s = 0 -> car on street 9, lilly on street 5, leo on street 14, s = 1 *)
      (pre s = 0) and car = 9 and lilly = 5 and leo = 14 and s = 1
    | (* s = 0 -> car on street 12, lilly on street 5, leo on street 14, s = 2 *)
      (pre s = 0) and car = 12 and lilly = 5 and leo = 14 and s = 2
    | (* s = 1 -> car on street 9, leo on street 11, lilly on street 12, s = 3 *)
      (pre s = 1) and car = 9 and lilly = 12 and leo = 11 and s = 3
    | (* s = 1 -> car on street 12, leo on street 11, lilly on street 5, s = 4 *)
      (pre s = 1) and car = 12 and lilly = 5 and leo = 11 and s = 4
    | (* s = 2 -> car on street 12, leo on street 11, lilly on street 5, s = 4 *)
      (pre s = 2) and car = 12 and lilly = 5 and leo = 11 and s = 4
    | (* s = 3 -> car on street 12, leo on street 14, lilly on street 5, s = 1 *)
      (pre s = 3) and car = 9 and lilly = 5 and leo = 14 and s = 1
```

```
  | (* s = 4 -> car on street 12, leo on street 14, lilly on street 5, s = 2 *)
    (pre s = 4) and car = 12 and lilly = 5 and leo = 14 and s = 2
}
```

Although larger and more complex scenarios can be written manually, this task is tedious and error-prone, in particular due to the representation of street names by natural numbers (the input language of Lutin supports only Boolean and numerical types), and may easily introduce redundant definitions or equivalent states.

A small example of an oracle for the previous scenario is given by the following Lustre node `oracle`, describing the expected output (perception grid) for each given input vector (the positions of the car and obstacles). As the Lutin node, the oracle takes, besides the inputs and outputs of the radar, as input also the state `s` to keep track of the evolution (according to the same small automaton shown in Figure 5.6). The oracle outputs the verdict `res`, i.e., whether the observed outputs are those expected for the state and the inputs. For instance, in state 3, lilly (in street 12) and leo (in street 11) should both be detected by the car (in street 9), whereas they should not be detected in the initial state. The oracle also computes two coverage variables `pass` and `blocked`: the former measures the coverage of state 2 (representing the situation where the car arrives at destination and all obstacles appeared), and the latter measures the coverage of the situation where the car is blocked by the obstacles.

```
const invisible = 2000;
const already_sent = 3000;
node oracle (s, car, lilly, leo, perception_leo, perception_lilly: int)
returns (res, pass, blocked: bool);
let res = true ->
  ((* lilly and leo are visible from the street 9 *)
    (s = 0 and car = 9 and lilly = invisible and leo = invisible and
     perception_lilly = invisible and perception_leo = invisible)
   or
   (* the perception did not change, it is already sent *)
    (s = 1 and car = 9 and lilly = 5 and leo = 14 and
     perception_lilly = already_sent and perception_leo = already_sent)
   or
   (* leo and lilly are visible from the street 9 *)
    (s = 3 and car = 9 and lilly = 12 and leo = 11 and
     perception_lilly = 12 and perception_leo = 11)
   or ... );
   (* true if the car reached the destination (state 2) *)
   pass = false -> if s = 2 then true else pre pass;
   (* true, if the car is blocked by the obstacles *)
   blocked = false -> if s = 3 then true else pre pass;
tel
```

Even more than for the Lutin constraints, manually deriving the oracle is complicated,

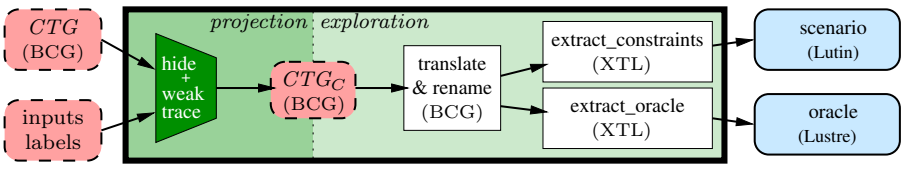


Figure 5.7: Overview of the derivation of synchronous test scenarios by projection and exploration of an asynchronous complete test graph CTG

mainly due to the dependency on the map and the necessity to enumerate all possible movements. For instance, to detect the `already_sent`, one should manually follow the scenario's evolution. Because one can easily forget some cases, an automated generation of these input constraints and the oracle is more convenient, as we illustrate in the next section.

5.5 Test Projection and Exploration

Each synchronous component of a GALS system is constrained by the other synchronous components, communication media, and environments present in the system. In this section we propose an approach that explicitly exploits these constraints to improve the unit testing of a synchronous component taken separately.

The idea is to automatically derive inputs for synchronous testing tools by projecting the complete test graph generated for the entire GALS system on the inputs and outputs of the synchronous component. In this way, the inputs provided to the synchronous component are realistic and relevant, because they are chosen according to possible execution scenarios of the overall GALS system. Furthermore, the synchronous tests generated in this way contain all possible inputs leading to the goals of test purposes (e.g., a possible input leading to collision in the case of $T1$, see Figure 5.8).

Figure 5.7 gives an overview of the approach. In a first step, a complete test graph for the overall GALS system is projected on the synchronous component C to be tested, resulting in a test graph for C. In a second step, this test graph is translated and renamed to be compatible with the synchronous testing tool Lurette. In the last step, the test graph is explored (using XTL [MG98] scripts), generating the input constraints and the oracle for testing C separately. In the remainder of this section, we present the approach in more detail, illustrating how it improves the testing of the radar of our autonomous car example.

5.5.1 Test graph projection

Projecting a complete test graph CTG on a synchronous component C consists in hiding all transitions labeled with an action that is neither an input nor an output of C, and reducing

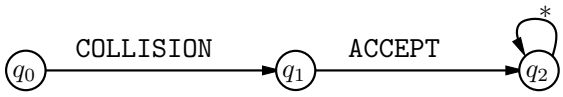


Figure 5.8: Test Purpose $T1$

Table 5.1: Sizes and run-time performance for the complete test graphs for the test purposes

	TP		CTG		time	mem.
	states	trans.	states	trans.	(s)	(MB)
$T1$	2	3	102,985	211,455	1237	231
$T2$	3	4	15,466	29,665	29	200
$T3$	3	4	15,444	29,957	26	200
$T4$	3	4	2,278	4,959	86	193
$T5$	3	5	21,930	42,788	36	201

the resulting graph for weak trace equivalence (see Section 2.1.2), yielding the projected test graph CTG_C. The reduction removes all internal transitions (created by the hiding), so that all actions of CTG_C are either an input or an output of C. A precondition for a successful projection is that all inputs and outputs of C, as well as the verdict transitions, are present and visible in CTG.

For the GRL model of the autonomous car (see Section 5.2), the only controllable inputs are the movements of obstacles; the observable outputs make it possible to study the behavior of the car. An example test purpose ($T1$) is to specify the situation where a collision occurs between a car and an obstacle. The test purpose is expressed as an LTS (in the AUT format) and shown in Figure 5.8, where a transition representing the collision (action `COLLISION`) should lead to a goal state, i.e., having an outgoing transition labelled by an `ACCEPT` action.

We also defined four other test purposes ($T2$, $T3$, $T4$, and $T5$) constraining the car and obstacles interaction on the map. For these test purposes, TESTOR generates the complete test graphs in less than a minute (see Table 5.1).

More particularly, the radar takes as input the positions of the car and of the obstacles. The positions of the obstacles are also a controllable input of the overall GALS system. But the position of the car is computed by the scenario depending on the output of another synchronous component, namely the action controller. The output of the radar is the perception grid, which is, inside the GALS system, sent to the action controller. Hiding, in the complete test graph CTG, all transitions but those corresponding to these inputs and outputs, and reducing the result with respect to weak trace equivalence yields the LTS $\mathtt{CTG}_{\mathtt{RADAR}}$. Columns 2 & 3 of Table 5.2 give the number of states and transitions of $\mathtt{CTG}_{\mathtt{RADAR}}$ for the five test purposes considered.

Table 5.2: Sizes and run-time performance for the tests generated for the test purposes

	$\mathtt{CTG_{RADAR}}$		constraints	oracle	time	mem.
	states	trans.	(Lutin lines)	(Lustre lines)	(s)	(MB)
$T1$	584	3617	3622	1916	1237	231
$T2$	89	281	286	295	29	200
$T3$	83	258	263	282	26	200
$T4$	218	1119	1124	557	86	193
$T5$	105	356	361	344	36	201

5.5.2 Translating and renaming

Because some of the data types used in the GALS model might not be exactly the same as those supported by the synchronous testing tools, a preliminary step is the conversion of the values present on the transition labels of CTG_C, for instance by applying appropriate renaming rules. Null values have been added, in order to transform non scalar data on transitions into variable names and value tuples of constant length.

We defined a generic format for the exploration tools to work properly. Each input transition is renamed into "`INPUT` $!s_1$ $!v_1$ ... $!s_m$ $!v_m$", where s_i is the name of the input variable and v_i its value; each output transition is similarly renamed into "`OUTPUT` $!s_1$ $!v_1$ ... $!s_n$ $!v_n$". For instance, the projected complete test graph $\mathtt{CTG_{RADAR}}$ contains non-scalar data structures, in particular, the perception grid computed by the radar is represented as a list and streets are identified by their names (i.e., character strings). To be usable with the synchronous test generator Lurette [JRB06], these lists need to be transformed into tuples of constant length, and the street names need to be translated into the corresponding (numeric) constants.

5.5.3 Test graph exploration

By construction, the projected CTG_C is an LTS describing the interaction with the SUT, and as such contains both, the sequence of inputs for the SUT and the verdict concerning the test outcome (i.e., the test oracle). Because the test inputs and oracle should be provided to Lurette using two different languages (Lutin for the input constraints and Lustre for the oracle), two explorations of (the renamed) CTG_C are required.

The generation of input constraints

The input constraints are generated by encoding CTG_C as a possibly nondeterministic node in Lutin with the XTL[6] [MG98] script `extract_constraints` (87 lines). The whole XTL

[6]More details about XTL are available in Section 2.3.2

script `extract_constraints` is presented in the Appendix C.1. This node has the same inputs as C and an additional input variable `s` corresponding to the current state of CTG_C, initialized to the initial state. The main loop of the node contains a nondeterministic choice, with a branch for each transition in CTG_C. A branch corresponding to a transition T is executed if `s` is equal to the source state of T, and as result of execution it sets `s` to the target state of T. A branch for an output transition specifies that the inputs are kept unchanged, which corresponds to the behavior expected for the special output transition δ on quiescent states. A branch for an input transition updates the corresponding inputs. A branch for a verdict transition, instead of looping on the same verdict state of the CTG_C (`pass` and `inconclusive`), sets the variable `s` to the initial state of the CTG, thus enabling to not generate the same inputs as a self-loop would do and to cover faster the whole CTG when a verdict is reached. Thus, the Lutin node describes exactly the set of input sequences contained in CTG_C.

An excerpt of the second step of the script `extract_constraints` is given by the following XTL code, illustrating the CTG encoding into the nondeterministic node in Lutin. In the first step of the script, the names of the inputs (stored in the variables `iv1, iv2, iv3`) and their initial values are extracted. Then we explore all transitions of the CTG (<|**fby** `on e:edge` |>`(...)`), and first extract and print the source state (`s`) of the transition, then match, extract and print the new input values (`p2, p4, p6`) through the `print_info` function for the input transition (`INPUTS ?...`), or match and print unchanged input values using the memory Lustre operator (**pre**)[7] for output transitions. Finally, we print the target states (`target (e)`) for all other transitions except the verdict transitions (`INCONCLUSIVE` and `PASS`), where the target state is replaced by the state `-1`, which corresponds to an intermediary state where the values are reset to the initial ones and leading to the initial state.

```
<| fby on e:edge |> (
    printf (" | pre s = ") fby
    print (source (e)) fby
    if (e -> [ INPUTS ?p1:raw ?p2:natural ?p3:raw ?p4:natural
               ?p5:raw ?p6: natural ]) then
       (* car movement; obstacles do not move *)
       print_info (p1, p3, p5, p2, p4, p6)
    else
       (* output or verdict transition: keep inputs unchanged *)
       printf (" and ") fby
       print (iv1) fby printf (" = pre ") fby print (iv1) fby
       printf (" and ") fby
       print (iv2) fby printf (" = pre ") fby print (iv2) fby
       printf (" and ") fby
       print (iv3) fby printf (" = pre ") fby print (iv3)
    end_if fby
```

[7]More details about the Lustre **pre** operator are available in Section 2.2.1

```
    (* change s to the target state *)
    printf (" and s = ") fby
    if e -> [INCONCLUSIVE] or e -> [PASS] then
      printf ("-1")
    else
      print (target (e))
    end_if fby
    printf ("\n")
) fby
```

An input of the radar is a new position of the car and the obstacles (`lilly` and `leo`). Because the exploration takes into account all transitions of $\mathtt{CTG}_{\mathtt{RADAR}}$, the generated Lutin node incorporates all evolution of the car and the obstacles that are relevant for the considered test purpose. The generated Lutin nodes are quite long (see Table 5.2) and complex, because they contain nondeterministic choices, induced by the random movements of the obstacles. Note that the generated Lutin node corresponds to a specific automatically generated automaton scenario (cf. Section 5.4). Writing similar Lutin nodes by hand would probably have been difficult and error-prone.

The generation of oracles

Because a synchronous block C is executed atomically, i.e., not interleaved with other synchronous components, a sequence of input transitions for C is immediately followed by the expected sequence of output transitions. Because the target state s of the last transition in such a sequence of inputs is also the source state of the first transition in the sequence of outputs, we call such a state s a *corner state*. For the radar, the sequence of input transitions corresponding to new positions of the car or the obstacles is followed by an output transition corresponding to the expected perception grid. The oracle is generated with the XTL script `extract_oracle` (263 lines), by encoding CTG_C as a deterministic node in Lustre, which, to each corner state and its set of inputs/outputs, associates a Boolean verdict, indicating whether the outputs are the expected ones. The whole XTL script `extract_oracle` is presented in the Appendix C.2. An excerpt of the script `extract_oracle` is given by the following XTL code describing the rules of extraction for corner states. In this code for each corner state `s`, we iterate over pairs of an input and an output transition, assign the input values (`i1, i2, i3`), to the accumulator (`a_i1, a_i2, a_i3`) and print them with the source state (`s`) and the output values (`o1, o2`) through the `print_corner` function.

```
<| fby on s:state where corner_state (s) |>
  let (a: action, i1, i2, i3: natural) =
    for in_e: edge among in (s), out_e: edge among out (s)
    in (a: action, a_i1, a_i2, a_i3: natural)
    where in_e -> [INPUTS ...] and out_e -> [OUTPUTS ...]
      apply (fby, replace, replace, replace)
```

```
      from (nop, 0, 0, 0)
      to
      if in_e -> [INPUTS _ ?i1:natural _ ?i2:natural _ ?i3:natural] then
        if (i1 = a_i1) and (i2 = a_i2) and (i3 = a_i3) then
        (* already handled combination of inputs : skip *)
          (nop, i1, i2, i3)
        else_if out_e -> [OUTPUTS _ ?o1:natural _ ?o2:natural] then
          (print_corner (s, iv1, iv2, iv3, ov1, ov2, i1, i2, i3, o1, o2),
          i1, i2, i3)
        else (* never reached *)
          (nop, a_i1, a_i2, a_i3)
        end_if
      else (* never reached *)
        (nop, a_i1, a_i2, a_i3)
      end_if
    end_for
  in
    use i1, i2, i3 in a end_use
  end_let fby
```

The oracle node is also defined for the verdict states of CTG_C. For a *pass* (respectively, *fail*) verdict it always returns true (respectively, false). For an *inconclusive* verdict it returns true iff the outputs are unchanged; this matches the generation of the input constraints. In order to observe the coverage of the generated tests, for each verdict state (`pass` and `inconclusive`) and each other state of the CTG_C, a coverage variable is introduced in the Lustre node generated for the oracle. These coverage variables are true if at least one of the executions passed by the corresponding state of CTG_C (see Section 5.4).

```
node oracle (s, car, lilly, leo, perception_leo, perception_lilly: int)
returns (res, pass, inconclusive, s0, ... s86: bool);
let
  res = true -> (
    if s = 6 then
    (car = 11 and lilly = 13 and leo = 6 and
     perception_lilly = 13 and perception_leo = 2000)
      ... );
  pass = false -> if s = 83 then true else pre pass;
  inconclusive = false -> if s = 7 then true else pre inconclusive;
  s0 = false -> if s = 0 then true else pre s0; ...
  s86 = false -> if s = 86 then true else pre s86; ...
```

The generated oracles are quite long (see Table 5.2) and complex, because they contain all oracle verdicts and coverage criteria. For each of the five test purposes, we executed the generated scenarios on the radar SUT using Lurette (the last columns of Table 5.2 indicate the overall time and memory consumed by the overall testing process). This enabled us to

discover (and fix) a mistake in the SUT related to the incorrect management of the special value `already_sent`.

⋆⋆⋆

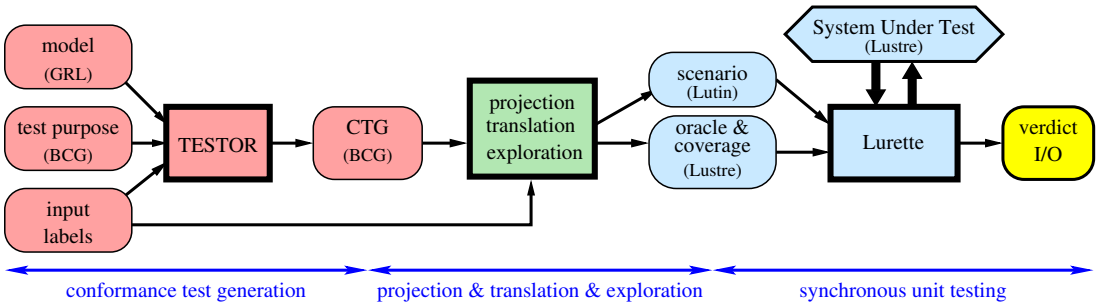


Figure 5.9: Overview of the testing methodology for GALS systems

We presented an automatic approach (see Figure 5.9) integrating both asynchronous and synchronous testing tools to derive complex, but relevant unit test cases for the synchronous components of a GALS system. From a formal model of the system in GRL [JLM16] and a test purpose, the conformance testing tool TESTOR [MMS18] automatically generates a complete test graph [JJ05] capturing the asynchronous behavior of the system relevant to the test purpose. Such a complete test graph is then projected on a synchronous component C and explored using XTL [MG98] scripts to provide a synchronous test scenario (input constraints in Lutin [RRJ08] and an oracle in Lustre [HCRP91]) required to test C with the Lurette tool [JRB06]. We automated all these steps using an SVL [GL01] script. The approach substantially relieves the burden of handcrafting these test scenarios, because, by construction, the derived scenarios constrain the inputs provided to C to relevant values, covering a test purpose, which might arise during the execution of the GALS system. We illustrated the approach on an autonomous car example. This work also enabled us to observe two common properties of GALS systems, more precisely that it is necessary to identify the non-progressing iterations of GALS's synchronous components and verify that the system reaches the terminal states by avoiding these iterations.

Chapter 6

Conclusion

GALS (Globally Asynchronous, Locally Synchronous) [Cha84] systems consist of multiple synchronous components that execute independently and interact with each other. They are intrinsically complex, often critical, and nowadays become increasingly large and distributed. The simultaneous presence of synchronous and asynchronous aspects makes their development and debugging difficult. Hence, to ensure their functional correctness and reliability, it is required to use a rigorous testing process based on formal methods, such as model-based testing [BJK+05], a validation technique taking advantage of a model of a system (both, requirements and behavior) to automate the generation of relevant test cases. In this dissertation, we have explored different ways to carry out the model-based testing of GALS systems. In particular, we have presented testing techniques for synchronous components, for communication protocols between components, and how to combine these two techniques for testing complete GALS systems.

Synchronous formal techniques for the functional testing of components

In the first part of this dissertation, we explored formal techniques for the functional testing of synchronous components. As a case-study, we reconsidered the Message Authenticator Algorithm (MAA), a pioneering cryptographic function designed in the mid-80s, and we formalized it as a synchronous dataflow algorithm in the synchronous language Lustre [HCRP91]. A large part of our Lustre model contains general definitions, half of which are largely independent of the MAA. To validate our Lustre model, we defined sets of test vectors derived from the specification in [DC88]. We automated the test execution process, by using the test generation tool Lurette [JRB06] of the Lustre V6 toolbox[1] [HCRP91, RRJ08]. The modeling and validation of the MAA enabled us to discover various mistakes in prior (informal and formal) specifications of the MAA, the test vectors and code of the ISO 1990 [ISO90, Annex E], and the ISO 1992 [ISO92, Annex A] standards,

[1] http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/

in the main program (implementation of the MAA in C) provided by [DC88], and in some of the compilers and verification tools that we used. Testing our Lustre MAA model was a tedious task: in order to automate the tests validation with a Lutin environment and an oracle, we had to define more than 400 constants. An automation of this testing process is therefore highly beneficial.

Asynchronous conformance testing techniques for a communication protocol

Then, we explored the formalization and the functional testing of a communication protocol between two (synchronous) components. For this purpose, we introduced TESTOR [MMS18], a new tool for on-the-fly conformance test case generation for asynchronous concurrent systems. TESTOR explores the model and generates automatically a set of test cases or a CTG (Complete Test Graph) to be executed on a physical implementation of the system. Like the existing tool TGV [JJ05], TESTOR was developed on top of the CADP[2] toolbox [GLMS13] and brings several enhancements: online testing by generating (controllable) test cases completely on the fly; a more versatile description of test purposes using the LNT language; and a modular architecture involving generic graph manipulation components from the OPEN/CAESAR environment [Gar98]. The modularity of TESTOR simplifies maintenance and fine-tuning of graph manipulation components, e.g., by adding or removing on-the-fly reductions, or by replacing the synchronous product between the model and the test purpose. Besides the ability to perform online testing, the on-the-fly test selection algorithm sometimes makes possible the extraction of test cases even when the generation of the CTG is infeasible. The experiments we carried out on ten-thousands of benchmark examples and three industrial case studies show that TESTOR consumes less memory than TGV for generating CTGs. As a case-study, we reconsider the formalization of the Transport Layer Security (TLS) 1.3 handshake, a protocol responsible for the authentication and exchange of keys necessary to establish or resume a secure communication. Taking the draft specification of the TLS [IET18] protocol Version 1.3 as starting point, we formalized the TLS handshake protocol in the LNT language [GLS17, CCG+19]. Our LNT model of TLS 1.3 handshake has been validated against a concrete implementation of TLS by an approach using TESTOR [BMMW18]. The validation process enabled us to find an ambiguity in the specification of the draft TLS 1.3 handshake [IET18].

Combined synchronous and asynchronous testing techniques for GALS systems

Finally, we proposed a combination of conformance test generation and synchronous testing techniques that cumulates their benefits for testing a complete GALS system. We presented an automatic approach combining both techniques, to minimize the handwriting test effort while maximizing the coverage. Our methodology integrates: (1) synchronous

[2] `https://cadp.inria.fr/`

and asynchronous concurrent models, (2) functional unit testing and behavioral conformance testing, and (3) various formal methods and their supporting tools. First, the GALS system is modelled in GRL (GALS Representation Language) [JLM16, Jeb16], a formal language designed for describing GALS systems. The asynchronous aspects of the GRL model can be validated using CADP, e.g., by checking temporal logic formulas expressing desired (global) correctness properties. Next, from the formal model of the system in GRL and a test purpose, the TESTOR tool can be used to automatically generate CTGs, capturing the asynchronous behavior of the system relevant to the test purpose, which can be used to assess whether an actual implementation of the GALS system conforms to the GRL model. Then, the CTG is projected on a synchronous component C and subsequently translated automatically, using XTL [MG98] scripts, into a scenario (i.e., input constraints in Lutin [RRJ08] and an oracle in Lustre), required to automate the testing of C using the synchronous test generation tool Lurette. Because these scenarios are automatically generated from the GRL model of the GALS system, they correspond by construction to relevant (and often complex) executions of the synchronous component. The whole methodology has been automated using SVL [GL01] scripts. The approach substantially relieves the burden of handcrafting these test scenarios, because, by construction, the derived scenarios constrain the inputs provided to C to relevant values, covering a high-level test purpose, which might arise during the execution of the GALS system. We illustrated the approach on an autonomous car example.

Perspectives

As future work, we plan to consider the behavioral coverage of GALS systems, which can be achieved by identifying a set of test purposes (ideally as small as possible) whose corresponding CTGs cover the whole state space of a GALS system. The set of test purposes could be built by deriving them from the action-based, branching-time temporal properties of the model (following the results of [FFJ+12] in the state-based, linear-time setting) or by synthesizing them according to behavioral coverage criteria.

Furthermore, we also plan to consider the transition coverage of a CTG, which can be achieved by identifying a set of test cases (ideally as small as possible) covering all states and transitions of a CTG. The difficulty is to find a simple solution, which extracts automatically controllable test cases (TCs) from an acyclic CTG, and which can cover the CTG using as few TCs as possible. This could be done by invoking the TESTOR algorithm repeatedly on the CTG itself, and by ensuring that the generated TCs have covered all CTG transitions, by taking into account also the transitions looping in the cycles possibly requiring unfolding of some cycles. The extraction strategy has to be studied and finely-tuned experimentally.

Finally, to execute a generated TC against a system under test (SUT), it is necessary to refine the TC to take into account the asynchronous communication between the SUT

and the tester. Actually, the SUT accepts every input at any time, whereas the TC is deterministic, i.e., there is no choice between an input and an output. An approach for connecting a TC (randomly selected) and an asynchronous SUT was defined in [WW09]. A similar approach using test purposes (TPs) to guide the test generation was proposed in [Bha14] and subsequently extended to timed automata [Bha17]. Recently, this kind of connection was automated by the MOTEST tool [GH17]. It would be interesting to investigate how these solutions could be integrated into our methodology to automate the execution of TCs on asynchronous SUTs, and to experiment it on a real GALS system.

Appendix A

Formal Model of the MAA in Lustre

This annex presents the specification of the MAA in the Lustre language. This specification is fully self-contained, meaning that it does not depend on any externally-defined library – with the minor disadvantage of somewhat lengthy definitions for byte and blocks constants. For readability, the specification has been split into 16 parts, each part being devoted to a particular type, a group of functions sharing a common purpose, or a collection of test vectors. The first parts contain general definitions that are largely independent from the MAA; starting from Section A.10, the definitions become increasingly more MAA-specific. All machine words (bytes, blocks, etc.) are represented according to the "big endian" convention, i.e., the first argument of each corresponding constructor denote the most significant bit.

A.1 Definitions for type Bit

We define bits using an enumeration `Bit` (`X0` and `X1`), together with functions implementing logical operations on bits.

```
type Bit = enum {X0, X1};

-----------------------------------------
function notBit (x1: Bit) returns (x: Bit);
let
  x = if x1 = X0 then X1 else X0;
tel;
-----------------------------------------
function andBit (x1, x2: Bit) returns (x: Bit);
let
  x = if x2 = X0 then X0 else x1;
tel;
```

```
-----------------------------------------
function orBit (x1, x2: Bit) returns (x: Bit);
let
  x = if x2 = X0 then x1 else X1;
tel;
-----------------------------------------
function xorBit (x1, x2: Bit) returns (x: Bit);
let
  x = if x2 = X0 then x1 else notBit (x1);
tel;
-----------------------------------------
```

A.2 Definitions for Type Byte

We define bytes (octets) using a structure `Octet` that contains the eight bits of a byte, together with functions implementing bitwise logical operations, left-shift and right-shift operations on bytes, as well as all byte constants needed to formally describe the MAA and its test vectors.

```
type Octet = struct {x1: Bit; x2: Bit; x3: Bit; x4: Bit;
                    x5: Bit; x6: Bit; x7: Bit; x8: Bit};
-----------------------------------------
const x00 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x01 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x02 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const x03 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x04 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x05 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x06 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const x07 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const x08 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const x09 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const x0A = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X1; x6 = X0; x7 = X1; x8 = X0};
```

```
const x0B = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const x0C = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const x0D = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const x0E = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x0F = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const x10 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x11 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x12 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const x13 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x14 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x15 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x16 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const x17 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const x18 = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const x1A = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const x1B = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const x1C = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const x1D = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const x1E = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x1F = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const x20 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x21 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x22 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
```

```
                 x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const x23 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x24 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x25 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x26 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const x27 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const x28 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const x29 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const x2A = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const x2B = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const x2D = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const x2E = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x2F = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const x30 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x31 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x32 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const x33 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x34 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x35 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x36 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const x37 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const x38 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const x39 = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X1};
```

```
const x3A = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const x3B = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const x3C = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const x3D = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const x3E = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x3F = Octet {x1 = X0; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const x40 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x41 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x42 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const x43 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x44 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x45 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x46 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const x47 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const x48 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const x49 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const x4A = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const x4B = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const x4C = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const x4D = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const x4E = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x4F = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const x50 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
```

```
                 x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x51 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x53 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x54 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x55 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x58 = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const x5A = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const x5B = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const x5C = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const x5D = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const x5E = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x5F = Octet {x1 = X0; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const x60 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x61 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x62 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const x63 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x64 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x65 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x66 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const x67 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const x69 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const x6A = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const x6B = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X1};
```

```
const x6C = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const x6D = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const x6E = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x6F = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const x70 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x71 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x72 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const x73 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x74 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x75 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x76 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const x77 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const x78 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const x79 = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const x7A = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const x7B = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const x7C = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const x7D = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const x7E = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x7F = Octet {x1 = X0; x2 = X1; x3 = X1; x4 = X1;
                  x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const x80 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x81 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x83 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
```

```
                 x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x84 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x85 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x86 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const x89 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const x8C = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const x8D = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const x8E = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x8F = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const x90 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const x91 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const x92 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const x93 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const x94 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const x95 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const x96 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const x97 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const x98 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const x99 = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const x9A = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const x9B = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const x9C = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const x9D = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X1};
```

```
const x9E = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const x9F = Octet {x1 = X1; x2 = X0; x3 = X0; x4 = X1;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const xA1 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const xA0 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const xA3 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const xA4 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const xA5 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const xA6 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const xA7 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const xA8 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const xA9 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const xAA = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const xAB = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const xAC = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const xAE = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const xAF = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X0;
                   x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const xB0 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const xB1 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const xB2 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const xB3 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const xB5 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const xB6 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                   x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const xB8 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
```

```
                 x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const xB9 = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const xBA = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const xBB = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const xBC = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const xBE = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const xBF = Octet {x1 = X1; x2 = X0; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const xC0 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const xC1 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const xC2 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const xC4 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const xC5 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const xC6 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const xC7 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const xC8 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const xC9 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const xCA = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const xCB = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const xCC = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const xCD = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const xCE = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const xD0 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const xD1 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X1};
```

```
const xD2 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const xD3 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const xD4 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const xD5 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const xD6 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const xD7 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const xD9 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const xD8 = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const xDB = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const xDC = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const xDD = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const xDE = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const xDF = Octet {x1 = X1; x2 = X1; x3 = X0; x4 = X1;
                  x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const xE0 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const xE1 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const xE2 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const xE3 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const xE6 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const xE8 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const xE9 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const xEA = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const xEB = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                  x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const xEC = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
```

```
                 x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const xED = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const xEE = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const xEF = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X0;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X1};
const xF0 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X0};
const xF1 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X0; x8 = X1};
const xF2 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X0};
const xF3 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X0; x7 = X1; x8 = X1};
const xF4 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X0};
const xF5 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X0; x8 = X1};
const xF6 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X0};
const xF7 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X0; x6 = X1; x7 = X1; x8 = X1};
const xF8 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X0};
const xF9 = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X0; x8 = X1};
const xFA = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X0};
const xFB = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X0; x7 = X1; x8 = X1};
const xFC = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X0};
const xFD = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X0; x8 = X1};
const xFE = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X0};
const xFF = Octet {x1 = X1; x2 = X1; x3 = X1; x4 = X1;
                 x5 = X1; x6 = X1; x7 = X1; x8 = X1};
-----------------------------------------
function andOctet (o1, o2: Octet) returns (o: Octet);
let
  o = Octet {x1 = andBit (o1.x1, o2.x1); x2 = andBit (o1.x2, o2.x2);
            x3 = andBit (o1.x3, o2.x3); x4 = andBit (o1.x4, o2.x4);
            x5 = andBit (o1.x5, o2.x5); x6 = andBit (o1.x6, o2.x6);
```

```
            x7 = andBit (o1.x7, o2.x7); x8 = andBit (o1.x8, o2.x8)};
tel;
-----------------------------------------
function orOctet (o1, o2: Octet) returns (o: Octet);
let
  o = Octet {x1 = orBit (o1.x1, o2.x1); x2 = orBit (o1.x2, o2.x2);
            x3 = orBit (o1.x3, o2.x3); x4 = orBit (o1.x4, o2.x4);
            x5 = orBit (o1.x5, o2.x5); x6 = orBit (o1.x6, o2.x6);
            x7 = orBit (o1.x7, o2.x7); x8 = orBit (o1.x8, o2.x8)};
tel;
-----------------------------------------
function xorOctet (o1, o2: Octet) returns (o: Octet);
let
  o = Octet {x1 = xorBit (o1.x1, o2.x1); x2 = xorBit (o1.x2, o2.x2);
            x3 = xorBit (o1.x3, o2.x3); x4 = xorBit (o1.x4, o2.x4);
            x5 = xorBit (o1.x5, o2.x5); x6 = xorBit (o1.x6, o2.x6);
            x7 = xorBit (o1.x7, o2.x7); x8 = xorBit (o1.x8, o2.x8)};
tel;
-----------------------------------------
function leftOctet1 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = o1.x2; x2 = o1.x3; x3 = o1.x4; x4 = o1.x5;
            x5 = o1.x6; x6 = o1.x7; x7 = o1.x8; x8 = X0};
tel;
-----------------------------------------
function leftOctet2 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = o1.x3; x2 = o1.x4; x3 = o1.x5; x4 = o1.x6;
            x5 = o1.x7; x6 = o1.x8; x7 = X0; x8 = X0};
tel;
-----------------------------------------
function leftOctet3 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = o1.x4; x2 = o1.x5; x3 = o1.x6; x4 = o1.x7;
            x5 = o1.x8; x6 = X0; x7 = X0; x8 = X0};
tel;
-----------------------------------------
function leftOctet4 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = o1.x5; x2 = o1.x6; x3 = o1.x7; x4 = o1.x8;
            x5 = X0; x6 = X0; x7 = X0; x8 = X0};
tel;
-----------------------------------------
function leftOctet5 (o1: Octet) returns (o: Octet);
let
```

```
  o = Octet {x1 = o1.x6; x2 = o1.x7; x3 = o1.x8; x4 = X0;
             x5 = X0; x6 = X0; x7 = X0; x8 = X0};
tel;
-----------------------------------------
function leftOctet6 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = o1.x7; x2 = o1.x8; x3 = X0; x4 = X0;
             x5 = X0; x6 = X0; x7 = X0; x8 = X0};
tel;
-----------------------------------------
function leftOctet7 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = o1.x8; x2 = X0; x3 = X0; x4 = X0;
             x5 = X0; x6 = X0; x7 = X0; x8 = X0};
tel;
-----------------------------------------
function rightOctet1 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = X0; x2 = o1.x1; x3 = o1.x2; x4 = o1.x3;
             x5 = o1.x4; x6 = o1.x5; x7 = o1.x6; x8 = o1.x7};
tel;
-----------------------------------------
function rightOctet2 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = X0; x2 = X0; x3 = o1.x1; x4 = o1.x2;
             x5 = o1.x3; x6 = o1.x4; x7 = o1.x5; x8 = o1.x6};
tel;
-----------------------------------------
function rightOctet3 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = o1.x1;
             x5 = o1.x2; x6 = o1.x3; x7 = o1.x4; x8 = o1.x5};
tel;
-----------------------------------------
function rightOctet4 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
             x5 = o1.x1; x6 = o1.x2; x7 = o1.x3; x8 = o1.x4};
tel;
-----------------------------------------
function rightOctet5 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
             x5 = X0; x6 = o1.x1; x7 = o1.x2; x8 = o1.x3};
tel;
```

```
-----------------------------------------
function rightOctet6 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
             x5 = X0; x6 = X0; x7 = o1.x1; x8 = o1.x2};
tel;
-----------------------------------------
function rightOctet7 (o1: Octet) returns (o: Octet);
let
  o = Octet {x1 = X0; x2 = X0; x3 = X0; x4 = X0;
             x5 = X0; x6 = X0; x7 = X0; x8 = o1.x1};
tel;
```

A.3 Definitions for Type OctetSum

We define type OctetSum that stores the result of the addition of two octets. Values of this type are 9-bit words, made up using the structure `OctetSum` that gathers one bit for the carry and an octet for the sum. The two principal functions for this type are addOctetSum (which adds two octets and an input carry bit, and returns both an output carry bit and an 8-bit sum), and addOctet (which is derived from the former one by dropping the input and output carry bits); the other functions are auxiliary functions implementing an 8-bit adder.

```
type OctetSum = struct {x: Bit; o: Octet};
-----------------------------------------
function addBit (x1, x2, x3: Bit) returns (x: Bit);
let
  x = xorBit (xorBit (x1, x2), x3);
tel;
-----------------------------------------
function carBit (x1, x2, x3: Bit) returns (x: Bit);
let
  x = orBit (andBit (andBit (x1, x2), notBit (x3)),
             andBit (orBit (x1, x2), x3));
tel;
-----------------------------------------
function addOctetSum (o1, o2: Octet; x: Bit) returns (os: OctetSum);
var x1, x11, x2, x22, x3, x33, x4, x44, x5, x55: Bit;
    x6, x66, x7, x77, x8, x88: Bit;
let
  x1 = carBit (o1.x8, o2.x8, x);
  x11 = addBit (o1.x8, o2.x8, x);
  x2 = carBit (o1.x7, o2.x7, x1);
```

```
  x22 = addBit (o1.x7, o2.x7, x1);
  x3 = carBit (o1.x6, o2.x6, x2);
  x33 = addBit (o1.x6, o2.x6, x2);
  x4 = carBit (o1.x5, o2.x5, x3);
  x44 = addBit (o1.x5, o2.x5, x3);
  x5 = carBit (o1.x4, o2.x4, x4);
  x55 = addBit (o1.x4, o2.x4, x4);
  x6 = carBit (o1.x3, o2.x3, x5);
  x66 = addBit (o1.x3, o2.x3, x5);
  x7 = carBit (o1.x2, o2.x2, x6);
  x77 = addBit (o1.x2, o2.x2, x6);
  x8 = carBit (o1.x1, o2.x1, x7);
  x88 = addBit (o1.x1, o2.x1, x7);
  os = OctetSum {x = x8;
               o = Octet {x1 = x88; x2 = x77; x3 = x66; x4 = x55;
                          x5 = x44; x6 = x33; x7 = x22; x8 = x11}};
tel;
-------------------------------------------
function dropCarryOctetSum (os: OctetSum) returns (o: Octet);
let
  o = os.o;
tel;
-------------------------------------------
function addOctet (o1, o2: Octet) returns (o: Octet);
let
  o = dropCarryOctetSum (addOctetSum (o1, o2, X0));
tel;
-------------------------------------------
```

A.4 Definitions for Type Half

We define 16-bit words ("named half words") using a structure `Half` that contains two bytes corresponding to a half word, together with two usual constants, and a function implementing operation mulOctet that takes two octets and computes their 16-bit product; the other functions are auxiliary functions implementing an 8-bit multiplier.

```
type Half = struct {o1: Octet; o2: Octet};
-------------------------------------------
const x0000 = Half {o1 = x00; o2 = x00};
const x0001 = Half {o1 = x00; o2 = x01};
-------------------------------------------
function mulOctetA (h1: Half; o1, o2: Octet) returns (h: Half);
var o3: Octet; os: OctetSum;
let
```

```
  o3 = addOctet (h1.o1, o1);
  os = addOctetSum (h1.o2, o2, X0);
  h = if os.x = X0 then
        Half {o1 = o3; o2 = os.o}
      else Half {o1 = addOctet (o3, x01); o2 = os.o};
tel;
-----------------------------------------
function mulOctet (o1, o2: Octet) returns (h: Half);
var h1, h2, h3, h4, h5, h6, h7: Half;
let
  h1 = if o1.x1 = X0 then x0000
       else mulOctetA (x0000, rightOctet1 (o2), leftOctet7 (o2));
  h2 = if o1.x2 = X0 then h1
       else mulOctetA (h1, rightOctet2 (o2), leftOctet6 (o2));
  h3 = if o1.x3 = X0 then h2
       else mulOctetA (h2, rightOctet3 (o2), leftOctet5 (o2));
  h4 = if o1.x4 = X0 then h3
       else mulOctetA (h3, rightOctet4 (o2), leftOctet4 (o2));
  h5 = if o1.x5 = X0 then h4
       else mulOctetA (h4, rightOctet5 (o2), leftOctet3 (o2));
  h6 = if o1.x6 = X0 then h5
       else mulOctetA (h5, rightOctet6 (o2), leftOctet2 (o2));
  h7 = if o1.x7 = X0 then h6
       else mulOctetA (h6, rightOctet7 (o2), leftOctet1 (o2));
  h = if o1.x8 = X0 then h7
       else mulOctetA (h7, x00, o2);
tel;
-----------------------------------------
```

A.5 Definitions for Type HalfSum

We define type HalfSum that stores the result of the addition of two half words. Values of this type are 17-bit words, made up using the constructor buildHalfSum that gathers one bit for the carry and a half word for the sum. The five principal non-constructors for this type are eqHalfSum (which tests equality), addHalfSum (which adds two half words and returns both a carry bit and a 16-bit sum), addHalf (which is derived from the former one by dropping the carry bit), addHalfOctet and addHalfOctets (which are similar to the former one but take byte arguments that are converted to half words before summation); the other non-constructors are auxiliary functions implementing a 16-bit adder built using two 8-bit adders.

```
type HalfSum = struct {x: Bit; h: Half};
-----------------------------------------
```

```
function addHalfSum (h1, h2: Half) returns (hs: HalfSum)
var os, os1: OctetSum;
let
  os = addOctetSum (h1.o2, h2.o2, X0);
  os1 = addOctetSum (h1.o1, h2.o1, os.x);
  hs = HalfSum {x = os1.x; h = Half {o1 = os1.o; o2 = os.o}};
tel;
-----------------------------------------
function dropCarryHalfSum (hs: HalfSum) returns (h: Half);
let
  h = hs.h;
tel;
-----------------------------------------
function addHalf (h1, h2: Half) returns (h: Half);
let
  h = dropCarryHalfSum (addHalfSum (h1, h2));
tel;
-----------------------------------------
function addHalfOctet (o1: Octet; h1: Half) returns (h: Half);
let
  h = addHalf (Half {o1 = x00; o2 = o1}, h1);
tel;
-----------------------------------------
function addHalfOctets (o1, o2: Octet) returns (h: Half);
let
  h = addHalf (Half {o1 = x00; o2 = o1}, Half {o1 = x00; o2 = o2});
tel;
-----------------------------------------
```

A.6 Definitions for Type Block

We define 32-bit words (named "blocks" according to the MAA terminology) using a constructor buildBlock that takes four bytes and returns a block. The seven principal non-constructors for this type are eqBlock (which tests equality), andBlock, orBlock, and xorBlock (which implement bitwise logical operations on blocks), HalfU and HalfL (which decompose a block into two half words), and mulHalf (which takes two half words and computes their 32-bit product); the other non-constructors are auxiliary functions implementing a 16-bit multiplier built using four 8-bit multipliers, as well as all block constants needed to formally describe the MAA and its test vectors.

```
type Block = struct {o1: Octet; o2: Octet; o3: Octet; o4: Octet};

-----------------------------------------
```

```
const x00000000 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x00};
const x00000001 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x01};
const x00000002 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x02};
const x00000003 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x03};
const x00000004 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x04};
const x00000005 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x05};
const x00000006 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x06};
const x00000007 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x07};
const x00000008 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x08};
const x00000009 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x09};
const x0000000A = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x0A};
const x0000000B = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x0B};
const x0000000C = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x0C};
const x0000000D = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x0D};
const x0000000E = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x0E};
const x0000000F = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x0F};
const x00000010 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x10};
const x00000012 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x12};
const x00000014 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x14};
const x00000016 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x16};
const x00000018 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x18};
const x0000001B = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x1B};
const x0000001D = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x1D};
const x0000001E = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x1E};
const x0000001F = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x1F};
const x00000031 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x31};
const x00000036 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x36};
const x00000060 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x60};
const x00000080 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = x80};
const x000000A5 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = xA5};
const x000000B6 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = xB6};
const x000000C4 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = xC4};
const x000000D2 = Block {o1 = x00; o2 = x00; o3 = x00; o4 = xD2};
const x00000100 = Block {o1 = x00; o2 = x00; o3 = x01; o4 = x00};
const x00000129 = Block {o1 = x00; o2 = x00; o3 = x01; o4 = x29};
const x0000018C = Block {o1 = x00; o2 = x00; o3 = x01; o4 = x8C};
const x00004000 = Block {o1 = x00; o2 = x00; o3 = x40; o4 = x00};
const x00010000 = Block {o1 = x00; o2 = x01; o3 = x00; o4 = x00};
const x00020000 = Block {o1 = x00; o2 = x02; o3 = x00; o4 = x00};
const x00030000 = Block {o1 = x00; o2 = x03; o3 = x00; o4 = x00};
const x00040000 = Block {o1 = x00; o2 = x04; o3 = x00; o4 = x00};
const x00060000 = Block {o1 = x00; o2 = x06; o3 = x00; o4 = x00};
const x00804021 = Block {o1 = x00; o2 = x80; o3 = x40; o4 = x21};
const x00FF00FF = Block {o1 = x00; o2 = xFF; o3 = x00; o4 = xFF};
const x0103050B = Block {o1 = x01; o2 = x03; o3 = x05; o4 = x0B};
```

```
const x01030703 = Block {o1 = x01; o2 = x03; o3 = x07; o4 = x03};
const x01030705 = Block {o1 = x01; o2 = x03; o3 = x07; o4 = x05};
const x0103070F = Block {o1 = x01; o2 = x03; o3 = x07; o4 = x0F};
const x02040801 = Block {o1 = x02; o2 = x04; o3 = x08; o4 = x01};
const x0297AF6F = Block {o1 = x02; o2 = x97; o3 = xAF; o4 = x6F};
const x07050301 = Block {o1 = x07; o2 = x05; o3 = x03; o4 = x01};
const x07C72EAA = Block {o1 = x07; o2 = xC7; o3 = x2E; o4 = xAA};
const x0A202020 = Block {o1 = x0A; o2 = x20; o3 = x20; o4 = x20};
const x0AD67E20 = Block {o1 = x0A; o2 = xD6; o3 = x7E; o4 = x20};
const x10000000 = Block {o1 = x10; o2 = x00; o3 = x00; o4 = x00};
const x11A9D254 = Block {o1 = x11; o2 = xA9; o3 = xD2; o4 = x54};
const x11AC46B8 = Block {o1 = x11; o2 = xAC; o3 = x46; o4 = xB8};
const x1277A6D4 = Block {o1 = x12; o2 = x77; o3 = xA6; o4 = xD4};
const x13647149 = Block {o1 = x13; o2 = x64; o3 = x71; o4 = x49};
const x160EE9B5 = Block {o1 = x16; o2 = x0E; o3 = xE9; o4 = xB5};
const x17065DBB = Block {o1 = x17; o2 = x06; o3 = x5D; o4 = xBB};
const x1D10D8D3 = Block {o1 = x1D; o2 = x10; o3 = xD8; o4 = xD3};
const x1D3B7760 = Block {o1 = x1D; o2 = x3B; o3 = x77; o4 = x60};
const x1D9C9655 = Block {o1 = x1D; o2 = x9C; o3 = x96; o4 = x55};
const x1F3F7FFF = Block {o1 = x1F; o2 = x3F; o3 = x7F; o4 = xFF};
const x21D869BA = Block {o1 = x21; o2 = xD8; o3 = x69; o4 = xBA};
const x24B66FB5 = Block {o1 = x24; o2 = xB6; o3 = x6F; o4 = xB5};
const x270EEDAF = Block {o1 = x27; o2 = x0E; o3 = xED; o4 = xAF};
const x277B4B25 = Block {o1 = x27; o2 = x7B; o3 = x4B; o4 = x25};
const x2829040B = Block {o1 = x28; o2 = x29; o3 = x04; o4 = x0B};
const x288FC786 = Block {o1 = x28; o2 = x8F; o3 = xC7; o4 = x86};
const x28EAD8B3 = Block {o1 = x28; o2 = xEA; o3 = xD8; o4 = xB3};
const x29907CD8 = Block {o1 = x29; o2 = x90; o3 = x7C; o4 = xD8};
const x29C1485F = Block {o1 = x29; o2 = xC1; o3 = x48; o4 = x5F};
const x29EEE96B = Block {o1 = x29; o2 = xEE; o3 = xE9; o4 = x6B};
const x2A6091AE = Block {o1 = x2A; o2 = x60; o3 = x91; o4 = xAE};
const x2BF8499A = Block {o1 = x2B; o2 = xF8; o3 = x49; o4 = x9A};
const x2E80AC30 = Block {o1 = x2E; o2 = x80; o3 = xAC; o4 = x30};
const x2FD76FFB = Block {o1 = x2F; o2 = xD7; o3 = x6F; o4 = xFB};
const x30261492 = Block {o1 = x30; o2 = x26; o3 = x14; o4 = x92};
const x303FF4AA = Block {o1 = x30; o2 = x3F; o3 = xF4; o4 = xAA};
const x33D5A466 = Block {o1 = x33; o2 = xD5; o3 = xA4; o4 = x66};
const x344925FC = Block {o1 = x34; o2 = x49; o3 = x25; o4 = xFC};
const x34ACF886 = Block {o1 = x34; o2 = xAC; o3 = xF8; o4 = x86};
const x3CD54DEB = Block {o1 = x3C; o2 = xD5; o3 = x4D; o4 = xEB};
const x3CF3A7D2 = Block {o1 = x3C; o2 = xF3; o3 = xA7; o4 = xD2};
const x3DD81AC6 = Block {o1 = x3D; o2 = xD8; o3 = x1A; o4 = xC6};
const x3F6F7248 = Block {o1 = x3F; o2 = x6F; o3 = x72; o4 = x48};
const x48B204D6 = Block {o1 = x48; o2 = xB2; o3 = x04; o4 = xD6};
const x4A645A01 = Block {o1 = x4A; o2 = x64; o3 = x5A; o4 = x01};
```

```
const x4C49AAE0 = Block {o1 = x4C; o2 = x49; o3 = xAA; o4 = xE0};
const x4CE933E1 = Block {o1 = x4C; o2 = xE9; o3 = x33; o4 = xE1};
const x4D53901A = Block {o1 = x4D; o2 = x53; o3 = x90; o4 = x1A};
const x4DA124A1 = Block {o1 = x4D; o2 = xA1; o3 = x24; o4 = xA1};
const x4F998E01 = Block {o1 = x4F; o2 = x99; o3 = x8E; o4 = x01};
const x50DEC930 = Block {o1 = x50; o2 = xDE; o3 = xC9; o4 = x30};
const x51AF3C1D = Block {o1 = x51; o2 = xAF; o3 = x3C; o4 = x1D};
const x51EDE9C7 = Block {o1 = x51; o2 = xED; o3 = xE9; o4 = xC7};
const x550D91CE = Block {o1 = x55; o2 = x0D; o3 = x91; o4 = xCE};
const x55555555 = Block {o1 = x55; o2 = x55; o3 = x55; o4 = x55};
const x55DD063F = Block {o1 = x55; o2 = xDD; o3 = x06; o4 = x3F};
const x5834A585 = Block {o1 = x58; o2 = x34; o3 = xA5; o4 = x85};
const x5A35D667 = Block {o1 = x5A; o2 = x35; o3 = xD6; o4 = x67};
const x5BC02502 = Block {o1 = x5B; o2 = xC0; o3 = x25; o4 = x02};
const x5CCA3239 = Block {o1 = x5C; o2 = xCA; o3 = x32; o4 = x39};
const x5EBA06C2 = Block {o1 = x5E; o2 = xBA; o3 = x06; o4 = xC2};
const xF0239DD5 = Block {o1 = xF0; o2 = x23; o3 = x9D; o4 = xD5};
const x5F38EEF1 = Block {o1 = x5F; o2 = x38; o3 = xEE; o4 = xF1};
const x613F8E2A = Block {o1 = x61; o2 = x3F; o3 = x8E; o4 = x2A};
const x63C70DBA = Block {o1 = x63; o2 = xC7; o3 = x0D; o4 = xBA};
const x6AD6E8A4 = Block {o1 = x6A; o2 = xD6; o3 = xE8; o4 = xA4};
const x6AEBACF8 = Block {o1 = x6A; o2 = xEB; o3 = xAC; o4 = xF8};
const x6D67E884 = Block {o1 = x6D; o2 = x67; o3 = xE8; o4 = x84};
const x7050EC5E = Block {o1 = x70; o2 = x50; o3 = xEC; o4 = x5E};
const x717153D5 = Block {o1 = x71; o2 = x71; o3 = x53; o4 = xD5};
const x7201F4DC = Block {o1 = x72; o2 = x01; o3 = xF4; o4 = xDC};
const x7397C9AE = Block {o1 = x73; o2 = x97; o3 = xC9; o4 = xAE};
const x74B39176 = Block {o1 = x74; o2 = xB3; o3 = x91; o4 = x76};
const x7783C51D = Block {o1 = x77; o2 = x83; o3 = xC5; o4 = x1D};
const x7792F9D4 = Block {o1 = x77; o2 = x92; o3 = xF9; o4 = xD4};
const x7BC180AB = Block {o1 = x7B; o2 = xC1; o3 = x80; o4 = xAB};
const x7DB2D9F4 = Block {o1 = x7D; o2 = xB2; o3 = xD9; o4 = xF4};
const x7DFEFBFF = Block {o1 = x7D; o2 = xFE; o3 = xFB; o4 = xFF};
const x7F76A3B0 = Block {o1 = x7F; o2 = x76; o3 = xA3; o4 = xB0};
const x7F839576 = Block {o1 = x7F; o2 = x83; o3 = x95; o4 = x76};
const x7FFFFFF0 = Block {o1 = x7F; o2 = xFF; o3 = xFF; o4 = xF0};
const x7FFFFFF1 = Block {o1 = x7F; o2 = xFF; o3 = xFF; o4 = xF1};
const x7FFFFFFC = Block {o1 = x7F; o2 = xFF; o3 = xFF; o4 = xFC};
const x7FFFFFFD = Block {o1 = x7F; o2 = xFF; o3 = xFF; o4 = xFD};
const x80000000 = Block {o1 = x80; o2 = x00; o3 = x00; o4 = x00};
const x80000002 = Block {o1 = x80; o2 = x00; o3 = x00; o4 = x02};
const x800000C2 = Block {o1 = x80; o2 = x00; o3 = x00; o4 = xC2};
const x80018000 = Block {o1 = x80; o2 = x01; o3 = x80; o4 = x00};
const x80018001 = Block {o1 = x80; o2 = x01; o3 = x80; o4 = x01};
const x80397302 = Block {o1 = x80; o2 = x39; o3 = x73; o4 = x02};
```

```
const x81D10CA3 = Block {o1 = x81; o2 = xD1; o3 = x0C; o4 = xA3};
const x89D635D7 = Block {o1 = x89; o2 = xD6; o3 = x35; o4 = xD7};
const x8CE37709 = Block {o1 = x8C; o2 = xE3; o3 = x77; o4 = x09};
const x8DC8BBDE = Block {o1 = x8D; o2 = xC8; o3 = xBB; o4 = xDE};
const x9115A558 = Block {o1 = x91; o2 = x15; o3 = xA5; o4 = x58};
const x91896CFA = Block {o1 = x91; o2 = x89; o3 = x6C; o4 = xFA};
const x9372CDC6 = Block {o1 = x93; o2 = x72; o3 = xCD; o4 = xC6};
const x98D1CC75 = Block {o1 = x98; o2 = xD1; o3 = xCC; o4 = x75};
const x9D15C437 = Block {o1 = x9D; o2 = x15; o3 = xC4; o4 = x37};
const x9DB15CF6 = Block {o1 = x9D; o2 = xB1; o3 = x5C; o4 = xF6};
const x9E2E7B36 = Block {o1 = x9E; o2 = x2E; o3 = x7B; o4 = x36};
const xA018C83B = Block {o1 = xA0; o2 = x18; o3 = xC8; o4 = x3B};
const xA0B87B77 = Block {o1 = xA0; o2 = xB8; o3 = x7B; o4 = x77};
const xA44AAAC0 = Block {o1 = xA4; o2 = x4A; o3 = xAA; o4 = xC0};
const xA511987A = Block {o1 = xA5; o2 = x11; o3 = x98; o4 = x7A};
const xA70FC148 = Block {o1 = xA7; o2 = x0F; o3 = xC1; o4 = x48};
const xA93BD410 = Block {o1 = xA9; o2 = x3B; o3 = xD4; o4 = x10};
const xAAAAAAAA = Block {o1 = xAA; o2 = xAA; o3 = xAA; o4 = xAA};
const xAB00FFCD = Block {o1 = xAB; o2 = x00; o3 = xFF; o4 = xCD};
const xAB01FCCD = Block {o1 = xAB; o2 = x01; o3 = xFC; o4 = xCD};
const xAB6EED4A = Block {o1 = xAB; o2 = x6E; o3 = xED; o4 = x4A};
const xABEEED6B = Block {o1 = xAB; o2 = xEE; o3 = xED; o4 = x6B};
const xACBC13DD = Block {o1 = xAC; o2 = xBC; o3 = x13; o4 = xDD};
const xB1CC1CC5 = Block {o1 = xB1; o2 = xCC; o3 = x1C; o4 = xC5};
const xB8142629 = Block {o1 = xB8; o2 = x14; o3 = x26; o4 = x29};
const xB99A62DE = Block {o1 = xB9; o2 = x9A; o3 = x62; o4 = xDE};
const xBA92DB12 = Block {o1 = xBA; o2 = x92; o3 = xDB; o4 = x12};
const xBBA57835 = Block {o1 = xBB; o2 = xA5; o3 = x78; o4 = x35};
const xBE9F0917 = Block {o1 = xBE; o2 = x9F; o3 = x09; o4 = x17};
const xBF2D7D85 = Block {o1 = xBF; o2 = x2D; o3 = x7D; o4 = x85};
const xBFEF7FDF = Block {o1 = xBF; o2 = xEF; o3 = x7F; o4 = xDF};
const xC1ED90DD = Block {o1 = xC1; o2 = xED; o3 = x90; o4 = xDD};
const xC21A1846 = Block {o1 = xC2; o2 = x1A; o3 = x18; o4 = x46};
const xC4EB1AEB = Block {o1 = xC4; o2 = xEB; o3 = x1A; o4 = xEB};
const xC6B1317E = Block {o1 = xC6; o2 = xB1; o3 = x31; o4 = x7E};
const xCBC865BA = Block {o1 = xCB; o2 = xC8; o3 = x65; o4 = xBA};
const xCD959B46 = Block {o1 = xCD; o2 = x95; o3 = x9B; o4 = x46};
const xD0482465 = Block {o1 = xD0; o2 = x48; o3 = x24; o4 = x65};
const xD636250D = Block {o1 = xD6; o2 = x36; o3 = x25; o4 = x0D};
const xD7843FDC = Block {o1 = xD7; o2 = x84; o3 = x3F; o4 = xDC};
const xD78634BC = Block {o1 = xD7; o2 = x86; o3 = x34; o4 = xBC};
const xD8804CA5 = Block {o1 = xD8; o2 = x80; o3 = x4C; o4 = xA5};
const xDB79FBDC = Block {o1 = xDB; o2 = x79; o3 = xFB; o4 = xDC};
const xDB9102B0 = Block {o1 = xDB; o2 = x91; o3 = x02; o4 = xB0};
const xE0C08000 = Block {o1 = xE0; o2 = xC0; o3 = x80; o4 = x00};
```

```
const xE6A12F07 = Block {o1 = xE6; o2 = xA1; o3 = x2F; o4 = x07};
const xEB35B97F = Block {o1 = xEB; o2 = x35; o3 = xB9; o4 = x7F};
const xF14D6E28 = Block {o1 = xF1; o2 = x4D; o3 = x6E; o4 = x28};
const xF2EF3501 = Block {o1 = xF2; o2 = xEF; o3 = x35; o4 = x01};
const xF6A09667 = Block {o1 = xF6; o2 = xA0; o3 = x96; o4 = x67};
const xFD297DA4 = Block {o1 = xFD; o2 = x29; o3 = x7D; o4 = xA4};
const xFDC1A8BA = Block {o1 = xFD; o2 = xC1; o3 = xA8; o4 = xBA};
const xFE4E5BDD = Block {o1 = xFE; o2 = x4E; o3 = x5B; o4 = xDD};
const xFECCAA6E = Block {o1 = xFE; o2 = xCC; o3 = xAA; o4 = x6E};
const xFEFC07F0 = Block {o1 = xFE; o2 = xFC; o3 = x07; o4 = xF0};
const xFF2D7DA5 = Block {o1 = xFF; o2 = x2D; o3 = x7D; o4 = xA5};
const xFFEF0001 = Block {o1 = xFF; o2 = xEF; o3 = x00; o4 = x01};
const xFFFF00FF = Block {o1 = xFF; o2 = xFF; o3 = x00; o4 = xFF};
const xFFFFFF2D = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = x2D};
const xFFFFFF3A = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = x3A};
const xFFFFFFF0 = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xF0};
const xFFFFFFF1 = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xF1};
const xFFFFFFF4 = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xF4};
const xFFFFFFF5 = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xF5};
const xFFFFFFF7 = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xF7};
const xFFFFFFF9 = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xF9};
const xFFFFFFFA = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xFA};
const xFFFFFFFB = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xFB};
const xFFFFFFFC = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xFC};
const xFFFFFFFD = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xFD};
const xFFFFFFFE = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xFE};
const xFFFFFFFF = Block {o1 = xFF; o2 = xFF; o3 = xFF; o4 = xFF};
-------------------------------------------
function andBlock (w1, w2: Block) returns (w: Block);
let
  w = Block {o1 = andOctet (w1.o1, w2.o1); o2 = andOctet (w1.o2, w2.o2);
             o3 = andOctet (w1.o3, w2.o3); o4 = andOctet (w1.o4, w2.o4)};
tel;
-------------------------------------------
function orBlock (w1, w2: Block) returns (w: Block);
let
  w = Block {o1 = orOctet (w1.o1, w2.o1); o2 = orOctet (w1.o2, w2.o2);
             o3 = orOctet (w1.o3, w2.o3); o4 = orOctet (w1.o4, w2.o4)};
tel;
-------------------------------------------
function xorBlock (w1, w2: Block) returns (w: Block);
let
  w = Block {o1 = xorOctet (w1.o1, w2.o1); o2 = xorOctet (w1.o2, w2.o2);
             o3 = xorOctet (w1.o3, w2.o3); o4 = xorOctet (w1.o4, w2.o4)};
tel;
```

```
-----------------------------------------
function HalfU (w: Block) returns (o1o2: Half);
let
  o1o2 = Half {o1 = w.o1; o2 = w.o2};
tel;
-----------------------------------------
function HalfL (w: Block) returns (o3o4: Half);
let
  o3o4 = Half {o1 = w.o3; o2 = w.o4};
tel;
-----------------------------------------
function mulHalf (h1, h2: Half) returns (w: Block);
var h3, h4, h5, h6, h7, h8, h9: Half;
let
  h3 = mulOctet (h1.o1, h2.o1);
  h4 = mulOctet (h1.o1, h2.o2);
  h5 = mulOctet (h1.o2, h2.o1);
  h6 = mulOctet (h1.o2, h2.o2);
  h7 = addHalfOctet (h4.o2, addHalfOctets (h5.o2, h6.o1));
  h8 = addHalfOctet (h7.o1, addHalfOctet (h3.o2,
                                    addHalfOctets (h4.o1, h5.o1)));
  h9 = addHalfOctets (h8.o1, h3.o1);
  w = Block {o1 = h9.o2; o2 = h8.o2; o3 = h7.o2; o4 = h6.o2};
tel;
-----------------------------------------
```

A.7 Definitions for Type BlockSum

We define type BlockSum that stores the result of the addition of two blocks. Values of this type are 33-bit words, made up using the constructor buildBlockSum that gathers one bit for the carry and a block for the sum. The five principal non-constructors for this type are eqBlockSum (which tests equality), addBlockSum (which adds two blocks and returns both a carry bit and a 32-bit sum), addBlock (which is derived from the former one by dropping the carry bit), addBlockHalf and addBlockHalves (which are similar to the former one but take half-word arguments that are converted to blocks before summation); the other non-constructors are auxiliary functions implementing a 32-bit adder built using four 8-bit adders.

```
type OctetSum = struct {x: Bit; o: Octet};
type BlockSum = struct {x: Bit; b: Block};
-----------------------------------------
function addBlockSum (w1, w2 : Block) returns (ws: BlockSum);
var os, os1, os2, os3: OctetSum;
```

```
let
  os = addOctetSum (w1.o4, w2.o4, X0);
  os1 = addOctetSum (w1.o3, w2.o3, os.x);
  os2 = addOctetSum (w1.o2, w2.o2, os1.x);
  os3 = addOctetSum (w1.o1, w2.o1, os2.x);
  ws = BlockSum {x = os3.x;
               w = Block {o1 = os3.o; o2 = os2.o;
                          o3 = os1.o; o4 = os.o}};

tel;
-----------------------------------------
function dropCarryBlockSum (ws: BlockSum) returns (w: Block);
let
  w = ws.w;
tel;
-----------------------------------------
function addBlock (w1, w2 : Block) returns (w: Block);
let
  w = dropCarryBlockSum (addBlockSum (w1, w2));
tel;
-----------------------------------------
function addBlockHalf (h1: Half; w1: Block) returns (w: Block);
let
  w = addBlock (Block {o1 = x00; o2 = x00; o3 = h1.o1; o4 = h1.o2}, w1);
tel;
-----------------------------------------
function addBlockHalves (h1, h2: Half) returns (w: Block);
let
  w = addBlock (Block {o1 = x00; o2 = x00; o3 = h1.o1; o4 = h1.o2},
              Block {o1 = x00; o2 = x00; o3 = h2.o1; o4 = h2.o2});
tel;
-----------------------------------------
```

A.8 Definitions for Type Pair

We define 64-bit words (named "pair" according to the MAA terminology) using a constructor buildPair that takes two blocks and returns a pair. The main function for this type is mulBlock (which takes two blocks and computes their 64-bit product); using auxiliary functions presented in Section A.6 implementing a 32-bit multiplier built using four 16-bit multipliers defined in Section A.4.

```
type Pair = struct {w1: Block; w2: Block};
-----------------------------------------
```

```
function mulBlock (w1, w2: Block) returns (ww: Pair);
var w11, w12, w21, w22, w3, w4, w5: Block;
let
  w11 = mulHalf (HalfU (w1), HalfU (w2));
  w12 = mulHalf (HalfU (w1), HalfL (w2));
  w21 = mulHalf (HalfL (w1), HalfU (w2));
  w22 = mulHalf (HalfL (w1), HalfL (w2));
  w3 = addBlockHalf (HalfL (w12),
                   addBlockHalves (HalfL (w21), HalfU (w22)));
  w4 = addBlockHalf (HalfU (w3), addBlockHalf (HalfL (w11),
                          addBlockHalves (HalfU (w12), HalfU(w21))));
  w5 = addBlockHalves (HalfU (w4), HalfU (w11));
  ww = Pair {w1 = Block {o1 = w5.o3; o2 = w5.o4;
                       o3 = w4.o3; o4 = w4.o4};
            w2 = Block {o1 = w3.o3; o2 = w3.o4;
                       o3 = w22.o3; o4 = w22.o4}};
tel;
-----------------------------------------
```

A.9 Definitions for Type Key

We define a type Key that is intended to represent the 64-bit keys (J, K) used by the MAA. This type has a constructor buildKey that takes two blocks and returns a key. In [MvOV96], keys are represented using the type Pair, but we prefer introducing a dedicated type to clearly distinguish between keys and, e.g., results of the multiplication of two blocks.

```
type Key = struct {K: Block; J: Block};
```

A.10 Definitions (1) of MAA-specific Cryptographic Functions

We define a first set of functions to be used for MAA computations, most of which were present in [DC88] or have been later introduced in [MvOV96].

```
function CYC (w1: Block) returns (w: Block);
let
  w = Block {o1 = Octet {x1 = w1.o1.x2; x2 = w1.o1.x3; x3 = w1.o1.x4;
                       x4 = w1.o1.x5; x5 = w1.o1.x6; x6 = w1.o1.x7;
                       x7 = w1.o1.x8; x8 = w1.o2.x1};
            o2 = Octet {x1 = w1.o2.x2; x2 = w1.o2.x3; x3 = w1.o2.x4;
```

```
                        x4 = w1.o2.x5; x5 = w1.o2.x6; x6 = w1.o2.x7;
                        x7 = w1.o2.x8; x8 = w1.o3.x1};
            o3 = Octet {x1 = w1.o3.x2; x2 = w1.o3.x3; x3 = w1.o3.x4;
                        x4 = w1.o3.x5; x5 = w1.o3.x6; x6 = w1.o3.x7;
                        x7 = w1.o3.x8; x8 = w1.o4.x1};
            o4 = Octet {x1 = w1.o4.x2; x2 = w1.o4.x3; x3 = w1.o4.x4;
                        x4 = w1.o4.x5; x5 = w1.o4.x6; x6 = w1.o4.x7;
                        x7 = w1.o4.x8; x8 = w1.o1.x1};};
tel;
-----------------------------------------
function FIX1 (w1: Block) returns (w: Block);
let
  w = andBlock (orBlock (w1, x02040801), xBFEF7FDF);
tel;
-----------------------------------------
function FIX2 (w1: Block) returns (w: Block);
let
  w = andBlock (orBlock (w1, x00804021), x7DFEFBFF);
tel;
-----------------------------------------
function needAjust (o: Octet) returns (b: bool);
let
  b = ((o = x00) or (o =xFF));
tel;
-----------------------------------------
function adjustCode (o: Octet) returns (x: Bit);
let
  x = if needAjust (o) = true then X1 else X0;
tel;
-----------------------------------------
function adjust (o1, o2: Octet) returns (o: Octet);
let
  o = if needAjust (o1) = true then xorOctet (o1, o2) else o1;
tel;
-----------------------------------------
function PAT(w1, w2: Block) returns (o: Octet);
let
  o = Octet {x1 = adjustCode (w1.o1); x2 = adjustCode (w1.o2);
            x3 = adjustCode (w1.o3); x4 = adjustCode (w1.o4);
            x5 = adjustCode (w2.o1); x6 = adjustCode (w2.o2);
            x7 = adjustCode (w2.o3); x8 = adjustCode (w2.o4)};
tel;
-----------------------------------------
function BYT (w1, w2: Block) returns (w, wp: Block);
var opat: Octet;
```

```
let
  opat = PAT (w1, w2);
  w = Block {o1 = adjust (w1.o1, rightOctet7 (opat));
             o2 = adjust (w1.o2, rightOctet6 (opat));
             o3 = adjust (w1.o3, rightOctet5 (opat));
             o4 = adjust (w1.o4, rightOctet4 (opat))};
  wp = Block {o1 = adjust (w2.o1, rightOctet3 (opat));
              o2 = adjust (w2.o2, rightOctet2 (opat));
              o3 = adjust (w2.o3, rightOctet1 (opat));
              o4 = adjust (w2.o4, opat)};
tel;
-------------------------------------------
function ADDC (w1, w2: Block) returns (ww: Pair);
var ws: BlockSum;
let
  ws = addBlockSum (w1, w2);
  ww = if ws.x = X0 then
         Pair {w1 = x00000000; w2 = ws.w}
       else Pair {w1 = x00000001; w2 = ws.w};
tel;
-------------------------------------------
```

A.11 Definitions (2) of MAA-specific Cryptographic Functions

We define a second set of functions, namely the "multiplicative" functions used for MAA computations. The three principal operations are MUL1, MUL2, and MUL2A.

```
function MUL1 (w1, w2 : Block) returns (w: Block);
var w1w2, w3w4: Pair;
let
  w1w2 = mulBlock (w1, w2);
  w3w4 = ADDC (w1w2.w1, w1w2.w2);
  w = addBlock (w3w4.w2, w3w4.w1);
tel;
-------------------------------------------
function MUL2 (w1, w2 : Block) returns (w: Block);
var w1w2, w3w4, w5w6: Pair; w3: Block;
let
  w1w2 = mulBlock (w1, w2);
  w3w4 = ADDC (w1w2.w1, w1w2.w1);
  w3 = addBlock (w3w4.w2, addBlock (w3w4.w1, w3w4.w1));
  w5w6 = ADDC (w3, w1w2.w2);
```

```
  w = addBlock (w5w6.w2, addBlock (w5w6.w1, w5w6.w1));
tel;
-----------------------------------------
function MUL2A (w1, w2 : Block) returns (w: Block);
var w1w2, w3w4: Pair; w3: Block;
let
  w1w2 = mulBlock (w1, w2);
  w3 = addBlock (w1w2.w1, w1w2.w1);
  w3w4 = ADDC (w3, w1w2.w2);
  w = addBlock (w3w4.w2, addBlock (w3w4.w1, w3w4.w1));
tel;
-----------------------------------------
```

A.12 Definitions (3) of MAA-specific Cryptographic Functions

We define a third set of auxilary functions used for MAA computations, and the higher-level functions that implement the MAA algorithm, namely the prelude, the inner loop, and the coda; the two principal functions are MAA (which computes the signature of a nonsegmented message) and MAC (which splits a message into 1024-byte segments and computes the overall signature of this message by iterating on each segment, the 4-byte signature of each segment being prepended to the bytes of the next segment).

```
function squareHalf (h: Half) returns (w: Block);
let
  w = mulHalf (h, h);
tel;
-----------------------------------------
function Q (o: Octet) returns (w: Block);
let
  w = squareHalf (addHalf (Half {o1 = x00; o2 = o}, x0001));
tel;
-----------------------------------------
function preludeJ (J1 : Block) returns (J12, J14, J16, J18: Block;
                                        J22, J24, J26, J28: Block);
let
   J12 = MUL1 (J1, J1);
   J14 = MUL1 (J12, J12);
   J16 = MUL1 (J12, J14);
   J18 = MUL1 (J12, J16);
   J22 = MUL2 (J1, J1);
   J24 = MUL2 (J22, J22);
```

```
   J26 = MUL2 (J22, J24);
   J28 = MUL2 (J22, J26);
tel;
-------------------------------------------
function preludeK (K1: Block) returns (K12, K14, K15, K17, K19: Block;
                                       K22, K24, K25, K27, K29: Block);
let
   K12 = MUL1 (K1, K1);
   K14 = MUL1 (K12, K12);
   K15 = MUL1 (K1, K14);
   K17 = MUL1 (K12, K15);
   K19 = MUL1 (K12, K17);
   K22 = MUL2 (K1, K1);
   K24 = MUL2 (K22, K22);
   K25 = MUL2 (K1, K24);
   K27 = MUL2 (K22, K25);
   K29 = MUL2 (K22, K27);
tel;
-------------------------------------------
function preludeHJ (J14, J16, J18, J24, J26, J28: Block)
                  returns (H4, H6, H8: Block);
let
   H4 = xorBlock (J14, J24);
   H6 = xorBlock (J16, J26);
   H8 = xorBlock (J18, J28);
tel;
-------------------------------------------
function preludeHK (K15, K17, K19, K25, K27, K29: Block; P : Octet)
                   returns (H0, H5, H7, H9: Block);
let
   H0 = xorBlock (K15, K25);
   H5 = MUL2 (H0, Q (P));
   H7 = xorBlock (K17, K27);
   H9 = xorBlock (K19, K29);
tel;
-------------------------------------------
function prelude (J, K: Block) returns (X, Y, V, W, S, T: Block);
var P: Octet; J1, J12, J14, J16, J18, J22, J24, J26, J28: Block;
    K1, K12, K14, K15, K17, K22, K24, K25, K27, K19, K29: Block;
    H4, H0, H5, H6, H7, H8, H9: Block;
let
  J1, K1 = BYT (J, K);
  P = PAT (J, K);
  J12, J14, J16, J18, J22, J24, J26, J28 = preludeJ (J1);
  K12, K14, K15, K17, K19, K22, K24, K25, K27, K29 = preludeK (K1);
```

```
  H4, H6, H8 = preludeHJ (J14, J16, J18, J24, J26, J28);
  H0, H5, H7, H9 = preludeHK (K15, K17, K19, K25, K27, K29, P);
  X, Y = BYT (H4, H5);
  V, W = BYT (H6, H7);
  S, T = BYT (H8, H9);
tel;
-------------------------------------------
function mainLoop (X, Y, V, W, B: Block) returns (Xp, Yp, Vp: Block);
var E: Block;
let
  Vp = CYC (V);
  E = xorBlock (Vp,W);
  Xp = MUL1 (xorBlock (X, B), FIX1 (addBlock (xorBlock (Y, B), E)));
  Yp = MUL2A (xorBlock (Y, B), FIX2 (addBlock (xorBlock (X, B), E)));
tel;
-------------------------------------------
function mainLoop2 (X0, Y0, V0, W, Z, B: Block) returns (Xp, Yp, Vp: Block);
var X, V, Y: Block;
let
  X, Y, V = mainLoop (X0, Y0, V0, W, Z);
  Xp, Yp, Vp = mainLoop (X, Y, V, W, B);
tel;
-------------------------------------------
function coda (X, Y, V, W, S, T: Block) returns (Z: Block);
var X1, X2, Y1, Y2, V1, V2: Block;
let
  X1, Y1, V1 = mainLoop (X, Y, V, W, S);
  X2, Y2, V2 = mainLoop (X1, Y1, V1, W, T);
  Z = xorBlock (X2,Y2);
tel;
-------------------------------------------
node MAC (KJ: Key; Mn: Block)
        returns (X, Y, V, W, S, T, Z: Block);
var X0, Y0, V0: Block;
    init: bool; n: int;
let
  init = true -> false;
  n = if init then 1
      else if pre n = 256 then 0
      else pre n + 1;
  -- initialisations
  X0, Y0, V0, W, S, T = prelude (KJ.J, KJ.K);
  -- mainloops
  X, Y, V = if init then
            mainLoop (X0, Y0, V0, W, Mn)
```

```
          else if n = 0 then
            -- mode of operations
            mainLoop2 (X0, Y0, V0, W, pre Z, Mn)
          else mainLoop (pre X, pre Y, pre V, W, Mn);
  -- coda
  Z = coda (X, Y, V, W, S, T);
tel;
```

A.13 Test Vectors (1) for Checking MAA Computations

We define a first set of test vectors for the MAA. The following expressions implement the checks listed in Tables 1, 2, and 3 of [DC88] and should all evaluate to true if the MAA functions are correctly implemented.

```
function check_Table_1_2 () returns (res: bool);
var U, L, U', L', U'', L'': Block;
let

  -- test vectors for function mul1 - cf. Table 1 of [ISO 8731-2:1992]
  assert (mul1 (x0000000F, x0000000E) = x000000D2);
  assert (mul1 (xFFFFFFF0, x0000000E) = xFFFFFF2D);
  assert (mul1 (xFFFFFFF0, xFFFFFFF1) = x000000D2);

  -- test vectors for function mul2 - cf. Table 1 of [ISO 8731-2:1992]
  assert (mul2 (x0000000F, x0000000E) = x000000D2);
  assert (mul2 (xFFFFFFF0, x0000000E) = xFFFFFF3A);
  assert (mul2 (xFFFFFFF0, xFFFFFFF1) = x000000B6);

  -- test vectors for function mul2A - cf. Table 1 of [ISO 8731-2:1992]
  assert (mul2A (x0000000F, x0000000E) = x000000D2);
  assert (mul2A (xFFFFFFF0, x0000000E) = xFFFFFF3A);
  assert (mul2A (x7FFFFFF0, xFFFFFFF1) = x800000C2);
  assert (mul2A (xFFFFFFF0, x7FFFFFF1) = x000000C4);

  -- test vectors for function BYT - cf. Table 2 of [ISO 8731-2:1992]
  U, L = BYT (x00000000, x00000000);
  assert (U = x0103070F);
  assert (L = x1F3F7FFF);
  U', L' = BYT (xFFFF00FF, xFFFFFFFF);
  assert (U' = xFEFC07F0);
  assert (L' = xE0C08000);
  U'', L'' = BYT (xAB00FFCD, xFFEF0001);
```

```
  assert (U'' = xAB01FCCD);
  assert (L'' = xF2EF3501);

  -- test vectors for function PAT - cf. Table 2 of [ISO 8731-2:1992]
  assert (PAT(x00000000, x00000000) = xFF);
  assert (PAT(xFFFF00FF, xFFFFFFFF) = xFF);
  assert (PAT(xAB00FFCD, xFFEF0001) = x6A);

  res = true;
tel;
-----------------------------------------
function check_Table_3 () returns (res: bool);
var U, U', U'', L, L', L'': Block;
    J1, J12, J14, J16, J18, J22, J24, J26, J28: Block;
    K1, K12, K14, K15, K17, K19, K22, K24, K25, K27, K29: Block;
    H0, H4, H5, H6, H7, H8, H9: Block; P: Octet;
let
  J1 = x00000100;
  K1 = x00000080;
  P = x01;
  J12, J14, J16, J18, J22, J24, J26, J28 = preludeJ (J1);
  K12, K14, K15, K17, K19, K22, K24, K25, K27, K29 = preludeK (K1);
  H4, H6, H8 = preludeHJ (J14, J16, J18, J24, J26, J28);
  H0, H5, H7, H9 = preludeHK (K15, K17, K19, K25, K27, K29, P);

  -- test vectors for J1i values - cf. Table 3 of [ISO 8731-2:1992]
  assert (J12 = x00010000);
  assert (J14 = x00000001);
  assert (J16 = x00010000);
  assert (J18 = x00000001);

  -- test vectors for J2i values - cf. Table 3 of [ISO 8731-2:1992]
  assert (J22 = x00010000);
  assert (J24 = x00000002);
  assert (J26 = x00020000);
  assert (J28 = x00000004);

  -- test vectors for Hi values - cf. Table 3 of [ISO 8731-2:1992]
  assert (H4 = x00000003);
  assert (H6 = x00030000);
  assert (H8 = x00000005);

  -- test vectors for K1i values - cf. Table 3 of [ISO 8731-2:1992]
  assert (K12 = x00004000);
  assert (K14 = x10000000);
```

```
  assert (K15 = x00000008);
  assert (K17 = x00020000);
  assert (K19 = x80000000);

  -- test vectors for K2i values - cf. Table 3 of [ISO 8731-2:1992]
  assert (K22 = x00004000);
  assert (K24 = x10000000);
  assert (K25 = x00000010);
  assert (K27 = x00040000);
  assert (K29 = x00000002);

  -- test vectors for Hi values - cf. Table 3 of [ISO 8731-2:1992]
  assert (H0 = x00000018);
  assert (Q (P) = x00000004);
  assert (H5 = x00000060);
  assert (H7 = x00060000);
  assert (H9 = x80000002);

  -- test vectors for function PAT - cf. Table 3 of [ISO 8731-2:1992]
  assert (PAT (H4, H5) = xEE);
  assert (PAT (H6, H7) = xBB);
  assert (PAT (H8, H9) = xE6);

  -- test vectors for function BYT - logically inferred from Table 3
  U, L = BYT (H4, H5);
  assert (U = x01030703);
  assert (L = x1D3B7760);
  U', L' = BYT (H6, H7);
  assert (U' = x0103050B);
  assert (L' = x17065DBB);
  U'', L'' = BYT (H8, H9);
  assert (U'' = x01030705);
  assert (L'' = x80397302);

  res = true;
tel;
```

A.14 Test Vectors (2) for Checking MAA Computations

We define a second set of test vectors for the MAA, based upon Table 4 of [DC88]. The following expressions implement six groups of checks (three single-block messages and one

three-block message). They should all evaluate to true if the main loop of MAA (as described page 10 of [DC88]) is correctly implemented.

```
function check_Table_4_m1 () returns (res: bool);
var A, B, C, D, E, F, F', F'', G, G', G'', M, V, V': Block;
    W, X0, X, X', Y0, Y, Y', Z: Block;
let
-- test vectors for the first single-Block message
  A = x00000004; -- fake "A" constant
  B = x00000001; -- fake "B" constant
  C = xFFFFFFF7; -- fake "C" constant
  D = xFFFFFFFB; -- fake "D" constant
  V = x00000003;
  W = x00000003;
  X0 = x00000002;
  Y0 = x00000003;
  M = x00000005;
  V' = CYC (V); assert (V' = x00000006);
  E = xorBlock (V', W); assert (E = x00000005);
  X = xorBlock (X0, M); assert (X = x00000007);
  Y = xorBlock (Y0, M); assert (Y = x00000006);
  F = addBlock (E, Y); assert (F = x0000000B);
  G = addBlock (E, X); assert (G = x0000000C);
  F' = orBlock (F, A); assert (F' = x0000000F);
  G' = orBlock (G, B); assert (G' = x0000000D);
  F'' = andBlock (F', C); assert (F'' = x00000007);
  G'' = andBlock (G', D); assert (G'' = x00000009);
  X' = mul1 (X, F''); assert (X' = x00000031);
  Y' = mul2A (Y, G''); assert (Y' = x00000036);
  Z = xorBlock (X', Y'); assert (Z = x00000007);

  res = true;

tel;
-----------------------------------------
function check_Table_4_m2 () returns (res: bool);
var A, B, C, D, E, F, F', F'', G, G', G'', M, V, V': Block;
    W, X0, X, X', Y0, Y, Y', Z: Block;
let
-- test vectors for the second single-Block message
  A = x00000001; -- fake "A" constant
  B = x00000004; -- fake "B" constant
  C = xFFFFFFF9; -- fake "C" constant
  D = xFFFFFFFC; -- fake "D" constant
  V = x00000003;
  W = x00000003;
```

```
  X0 = xFFFFFFFD;
  Y0 = xFFFFFFFC;
  M = x00000001;
  V' = CYC (V); assert (V' = x00000006);
  E = xorBlock (V', W); assert (E = x00000005);
  X = xorBlock (X0, M); assert (X = xFFFFFFFC);
  Y = xorBlock (Y0, M); assert (Y = xFFFFFFFD);
  F = addBlock (E, Y); assert (F = x00000002);
  G = addBlock (E, X); assert (G = x00000001);
  F' = orBlock (F, A); assert (F' = x00000003);
  G' = orBlock (G, B); assert (G' = x00000005);
  F'' = andBlock (F', C); assert (F'' = x00000001);
  G'' = andBlock (G', D); assert (G'' = x00000004);
  X' = mul1 (X, F''); assert (X' = xFFFFFFFC);
  Y' = mul2A (Y, G''); assert (Y' = xFFFFFFFA);
  Z = xorBlock (X', Y'); assert (Z = x00000006);

  res = true;

tel;
-------------------------------------------
function check_Table_4_m3 () returns (res: bool);
var A, B, C, D, E, F, F', F'', G, G', G'', M, V, V': Block;
    W, X0, X, X', Y0, Y, Y', Z: Block;
let
-- test vectors for the third single-Block message
  A = x00000001; -- fake "A" constant
  B = x00000002; -- fake "B" constant
  C = xFFFFFFFE; -- fake "C" constant
  D = x7FFFFFFD; -- fake "D" constant
  V = x00000007;
  W = x00000007;
  X0 = xFFFFFFFD;
  Y0 = xFFFFFFFC;
  M = x00000008;
  V' = CYC (V); assert (V' = x0000000E);
  E = xorBlock (V', W); assert (E = x00000009);
  X = xorBlock (X0, M); assert (X = xFFFFFFF5);
  Y = xorBlock (Y0, M); assert (Y = xFFFFFFF4);
  F = addBlock (E, Y); assert (F = xFFFFFFFD);
  G = addBlock (E, X); assert (G = xFFFFFFFE);
  F' = orBlock (F, A); assert (F' = xFFFFFFFD);
  G' = orBlock (G, B); assert (G' = xFFFFFFFE);
  F'' = andBlock (F', C); assert (F'' = xFFFFFFFC);
  G'' = andBlock (G', D); assert (G'' = x7FFFFFFC);
```

```
  X' = mul1 (X, F''); assert (X' = x0000001E);
  Y' = mul2A (Y, G''); assert (Y' = x0000001E);
  Z = xorBlock (X', Y'); assert (Z = x00000000);

  res = true;

tel;
-----------------------------------------
function check_3_messages_m1 () returns (res: bool);
var A, B, C, D, E, F, F', F'', G, G', G'', M, V, V': Block;
    W, X0, X, X', Y0, Y, Y', Z: Block;
let
-- test vectors for the three-Block message: first Block
  A = x00000002; -- fake "A" constant
  B = x00000001; -- fake "B" constant
  C = xFFFFFFFB; -- fake "C" constant
  D = xFFFFFFFB; -- fake "D" constant
  V = x00000001;
  W = x00000001;
  X0 = x00000001;
  Y0 = x00000002;
  M = x00000000;
  V' = CYC (V); assert (V' = x00000002);
  E = xorBlock (V', W); assert (E = x00000003);
  X = xorBlock (X0, M); assert (X = x00000001);
  Y = xorBlock (Y0, M); assert (Y = x00000002);
  F = addBlock (E, Y); assert (F = x00000005);
  G = addBlock (E, X); assert (G = x00000004);
  F' = orBlock (F, A); assert (F' = x00000007);
  G' = orBlock (G, B); assert (G' = x00000005);
  F'' = andBlock (F', C); assert (F'' = x00000003);
  G'' = andBlock (G', D); assert (G'' = x00000001);
  X' = mul1 (X, F''); assert (X' = x00000003);
  Y' = mul2A (Y, G''); assert (Y' = x00000002);
  Z = xorBlock (X', Y'); assert (Z = x00000001);

  res = true;

tel;
-----------------------------------------
function check_3_messages_m2 () returns (res: bool);
var A, B, C, D, E, F, F', F'', G, G', G'', M, V, V': Block;
    W, X0, X, X', Y0, Y, Y', Z: Block;
let
-- test vectors for the three-Block message: second Block
```

```
  A = x00000002; -- fake "A" constant
  B = x00000001; -- fake "B" constant
  C = xFFFFFFFB; -- fake "C" constant
  D = xFFFFFFFB; -- fake "D" constant
  V = x00000002;
  W = x00000001;
  X0 = x00000003;
  Y0 = x00000002;
  M = x00000001;
  V' = CYC (V); assert (V' = x00000004);
  E = xorBlock (V', W); assert (E = x00000005);
  X = xorBlock (X0, M); assert (X = x00000002);
  Y = xorBlock (Y0, M); assert (Y = x00000003);
  F = addBlock (E, Y); assert (F = x00000008);
  G = addBlock (E, X); assert (G = x00000007);
  F' = orBlock (F, A); assert (F' = x0000000A);
  G' = orBlock (G, B); assert (G' = x00000007);
  F'' = andBlock (F', C); assert (F'' = x0000000A);
  G'' = andBlock (G', D); assert (G'' = x00000003);
  X' = mul1 (X, F''); assert (X' = x00000014);
  Y' = mul2A (Y, G''); assert (Y' = x00000009);
  Z = xorBlock (X', Y'); assert (Z = x0000001D);

  res = true;

tel;
-----------------------------------------
function check_3_messages_m3 () returns (res: bool);
var A, B, C, D, E, F, F', F'', G, G', G'', M, V, V': Block;
    W, X0, X, X', Y0, Y, Y', Z: Block;
let
-- test vectors for the three-Block message: third Block
  A = x00000002; -- fake "A" constant
  B = x00000001; -- fake "B" constant
  C = xFFFFFFFB; -- fake "C" constant
  D = xFFFFFFFB; -- fake "D" constant
  V = x00000004;
  W = x00000001;
  X0 = x00000014;
  Y0 = x00000009;
  M = x00000002;
  V' = CYC (V'); assert (V' = x00000008);
  E = xorBlock (V', W); assert (E = x00000009);
  X = xorBlock (X0, M); assert (X = x00000016);
  Y = xorBlock (Y0, M); assert (Y = x0000000B);
```

```
  F = addBlock (E, Y); assert (F = x00000014);
  G = addBlock (E, X); assert (G = x0000001F);
  F' = orBlock (F, A); assert (F' = x00000016);
  G' = orBlock (G, B); assert (G' = x0000001F);
  F'' = andBlock (F', C); assert (F'' = x00000012);
  G'' = andBlock (G', D); assert (G'' = x0000001B);
  X' = mul1 (X, F''); assert (X' = x0000018C);
  Y' = mul2A (Y, G''); assert (Y' = x00000129);
  Z = xorBlock (X', Y'); assert (Z = x000000A5);

  res = true;

tel;
-----------------------------------------
```

We complete the above tests with additional test vectors taken from [ISO90, Annex E.3.3], which only gives detailed values for the first block of the 84-block test message.

```
function check_Annex_E () returns (res: bool);
var A, B, C, D, E, F, F', F'', G, G', G'', M, V0, V: Block;
    W, X0, X, X', Y0, Y, Y': Block;
let
-- test vectors for block x0A202020 with key (J = xE6A12F07, K = x9D15C437)
  A = x02040801; -- true "A" constant
  B = x00804021; -- true "B" constant
  C = xBFEF7FDF; -- true "C" constant
  D = x7DFEFBFF; -- true "D" constant
  X0 = x21D869BA;
  Y0 = x7792F9D4;
  V0 = xC4EB1AEB;
  W = xF6A09667;
  M = x0A202020;
  V = CYC (V0); assert (V = x89D635D7);
  E = xorBlock (V, W); assert (E = x7F76A3B0);
  X = xorBlock (X0, M); assert (X = x2BF8499A);
  Y = xorBlock (Y0, M); assert (Y = x7DB2D9F4);
  F = addBlock (E, Y); assert (F = xFD297DA4);
  G = addBlock (E, X); assert (G = xAB6EED4A);
  F' = orBlock (F, A); assert (F' = xFF2D7DA5);
  G' = orBlock (G, B); assert (G' = xABEEED6B);
  F'' = andBlock (F', C); assert (F'' = xBF2D7D85);
  G'' = andBlock (G', D); assert (G'' = x29EEE96B);
  X' = mul1 (X, F''); assert (X' = x0AD67E20);
  Y' = mul2A (Y, G''); assert (Y' = x30261492);

  res = true;
```

```
tel;
-----------------------------------------
```

A.15 Test vectors (3) for Checking MAA Computations

We define a third set of test vectors for the MAA, based upon Table 5 of [DC88]. The following expressions implement four groups of checks, with two different keys and two different messages. They should all evaluate to true if the MAA signature is correctly computed.

```
function check_Table_5_v1 () returns (res: bool);
var J, K, X0, X, X', X'', X''', Y0, Y, Y', Y'', Y''': Block;
   V0, V, V', V'', V''', W, S, T, Z, M1, M2: Block;
let
-- test vectors of the first column of Table 5
  J = x00FF00FF;
  K = x00000000;
  M1 = x55555555;
  M2 = xAAAAAAAA;
  assert (PAT (J, K) = xFF);
  X0, Y0, V0, W, S, T = prelude (J, K);
  assert (X0 = x4A645A01);
  assert (Y0 = x50DEC930);
  assert (V0 = x5CCA3239);
  assert (W = xFECCAA6E);
  assert (S = x51EDE9C7);
  assert (T = x24B66FB5);
  -- 1st MainLoop iteration
  X, Y, V = mainLoop (X0, Y0, V0, W, M1);
  assert (X = x48B204D6);
  assert (Y = x5834A585);
  -- 2nd MainLoop iteration
  X', Y', V' = mainLoop (X, Y, V, W, M2);
  assert (X' = x4F998E01);
  assert (Y' = xBE9F0917);
  -- Coda: MainLoop iteration with S
  X'', Y'', V'' = mainLoop (X', Y', V', W, S);
  assert (X'' = x344925FC);
  assert (Y'' = xDB9102B0);
  -- Coda: MainLoop iteration with T
  X''', Y''', V''' = mainLoop (X'', Y'', V'', W, T);
```

```
  assert (X''' = x277B4B25);
  assert (Y''' = xD636250D);
  Z = xorBlock (X''',Y''');
  assert (Z = xF14D6E28);

  res = true;

tel;
-----------------------------------------
function check_Table_5_v2 () returns (res: bool);
var J, K, X0, X, X', X'', X''', Y0, Y, Y', Y'', Y''': Block;
    V0, V, V', V'', V''', W, S, T, Z, M1, M2: Block;
let
-- test vectors of the second column of Table 5
  J = x00FF00FF;
  K = x00000000;
  M1 = xAAAAAAAA;
  M2 = x55555555;
  assert (PAT (J, K) = xFF);
  X0, Y0, V0, W, S, T = prelude (J, K);
  assert (X0 = x4A645A01);
  assert (Y0 = x50DEC930);
  assert (V0 = x5CCA3239);
  assert (W = xFECCAA6E);
  assert (S = x51EDE9C7);
  assert (T = x24B66FB5);
  -- 1st MainLoop iteration
  X, Y, V = mainLoop (X0, Y0, V0, W, M1);
  assert (X = x6AEBACF8);
  assert (Y = x9DB15CF6);
  -- 2nd MainLoop iteration
  X', Y', V' = mainLoop (X, Y, V, W, M2);
  assert (X' = x270EEDAF);
  assert (Y' = xB8142629);
  -- Coda: MainLoop iteration with S
  X'', Y'', V'' = mainLoop (X', Y', V', W, S);
  assert (X'' = x29907CD8);
  assert (Y'' = xBA92DB12);
  -- Coda: MainLoop iteration with T
  X''', Y''', V''' = mainLoop (X'', Y'', V'', W, T);
  assert (X''' = x28EAD8B3);
  assert (Y''' = x81D10CA3);
  Z = xorBlock (X''',Y''');
  assert (Z = xA93BD410);
```

```
  res = true;

tel;
-----------------------------------------
function check_Table_5_v3 () returns (res: bool);
var J, K, X0, X, X', X'', X''', Y0, Y, Y', Y'', Y''': Block;
    V0, V, V', V'', V''', W, S, T, Z, M1, M2: Block;
let
-- test vectors of the third column of Table 5
  J = x55555555;
  K = x5A35D667;
  M1 = x00000000;
  M2 = xFFFFFFFF;
  assert (PAT (J, K) = x00);
  X0, Y0, V0, W, S, T = prelude (J, K);
  assert (X0 = x34ACF886);
  assert (Y0 = x7397C9AE);
  assert (V0 = x7201F4DC);
  assert (W = x2829040B);
  assert (S = x9E2E7B36);
  assert (T = x13647149);
  -- 1st MainLoop iteration
  X, Y, V = mainLoop (X0, Y0, V0, W, M1);
  assert (X = x2FD76FFB);
  assert (Y = x550D91CE);
  -- 2nd MainLoop iteration
  X', Y', V' = mainLoop (X, Y, V, W, M2);
  assert (X' = xA70FC148);
  assert (Y' = x1D10D8D3);
  -- Coda: MainLoop iteration with S
  X'', Y'', V'' = mainLoop (X', Y', V', W, S);
  assert (X'' = xB1CC1CC5);
  assert (Y'' = x29C1485F);
  -- Coda: MainLoop iteration with T
  X''', Y''', V''' = mainLoop (X'', Y'', V'', W, T);
  assert (X''' = x288FC786);
  assert (Y''' = x9115A558);
  Z = xorBlock (X''',Y''');
  assert (Z = xB99A62DE);

  res = true;

tel;
-----------------------------------------
function check_Table_5_v4 () returns (res: bool);
```

```
var J, K, X0, X, X', X'', X''', Y0, Y, Y', Y'', Y''': Block;
    V0, V, V', V'', V''', W, S, T, Z, M1, M2: Block;
let
-- test vectors of the fourth column of Table 5
  J = x55555555;
  K = x5A35D667;
  M1 = xFFFFFFFF;
  M2 = x00000000;
  assert (PAT (J, K) = x00);
  X0, Y0, V0, W, S, T = prelude (J, K);
  assert (X0 = x34ACF886);
  assert (Y0 = x7397C9AE);
  assert (V0 = x7201F4DC);
  assert (W = x2829040B);
  assert (S = x9E2E7B36);
  assert (T = x13647149);
  -- 1st MainLoop iteration
  X, Y, V = mainLoop (X0, Y0, V0, W, M1);
  assert (X = x8DC8BBDE);
  assert (Y = xFE4E5BDD);
  -- 2nd MainLoop iteration
  X', Y', V' = mainLoop (X, Y, V, W, M2);
  assert (X' = xCBC865BA);
  assert (Y' = x0297AF6F);
  -- Coda: MainLoop iteration with S
  X'', Y'', V'' = mainLoop (X', Y', V', W, S);
  assert (X'' = x3CF3A7D2);
  assert (Y'' = x160EE9B5);
  X''', Y''', V''' = mainLoop (X'', Y'', V'', W, T);
  assert (X''' = xD0482465);
  assert (Y''' = x7050EC5E);
  Z = xorBlock (X''',Y''');
  assert (Z = xA018C83B);

  res = true;

tel;
-----------------------------------------
```

We complete the above tests with additional test vectors taken from from [ISO90, Annex E.3.3], which gives prelude results computed for another key.

```
function check_Prelude_Annex_E33 () returns (res: bool);
var J, K, X, Y, V, W, S, T: Block;
let
-- test vectors of Annex E.3.3 of [ISO 8730:1990]
```

```
  J = xE6A12F07;
  K = x9D15C437;
  X, Y, V, W, S, T = prelude (J, K);
  assert (X = x21D869BA);
  assert (Y = x7792F9D4);
  assert (V = xC4EB1AEB);
  assert (W = xF6A09667);
  assert (S = x6D67E884);
  assert (T = xA511987A);

  res = true;

tel;
-----------------------------------------
```

A.16 Test Vectors (4) for Checking MAA Computations

We define a last set of test vectors for the MAA. The first one (a message of 20 blocks containing only zeros) was directly taken from Table 6 of [DC88].

```
function check_message_20 () returns (res: bool);
var B, J, K, X0, Y0, V0, W, S, T: Block;
    X, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11: Block;
    X12, X13, X14, X15, X16, X17, X18, X19, X20, X21: Block;
    Y, Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8, Y9, Y10, Y11: Block;
    Y12, Y13, Y14, Y15, Y16, Y17, Y18, Y19, Y20, Y21: Block;
    V, V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11: Block;
    V12, V13, V14, V15, V16, V17, V18, V19, V20, V21: Block;
let
  -- test vectors for the whole algorithm
  J = x80018001;
  K = x80018000;

  -- test mentioned in Table 6 of [ISO 8731-2:1992]
  -- iterations on a message containg 20 null Blocks
  X0, Y0, V0, W, S, T = prelude (J, K);
  B = x00000000;
  -- 1st MainLoop iteration
  X, Y, V = mainLoop (X0, Y0, V0, W, B);
  assert (X = x303FF4AA);
  assert (Y = x1277A6D4);
  -- 2nd MainLoop iteration
```

```
X1, Y1, V1 = mainLoop (X, Y, V, W, B);
assert (X1 = x55DD063F);
assert (Y1 = x4C49AAE0);
-- 3rd MainLoop iteration
X2, Y2, V2 = mainLoop (X1, Y1, V1, W, B);
assert (X2 = x51AF3C1D);
assert (Y2 = x5BC02502);
-- 4th MainLoop iteration
X3, Y3, V3 = mainLoop (X2, Y2, V2, W, B);
assert (X3 = xA44AAAC0);
assert (Y3 = x63C70DBA);
-- 5th MainLoop iteration
X4, Y4, V4 = mainLoop (X3, Y3, V3, W, B);
assert (X4 = x4D53901A);
assert (Y4 = x2E80AC30);
-- 6th MainLoop iteration
X5, Y5, V5 = mainLoop (X4, Y4, V4, W, B);
assert (X5 = x5F38EEF1);
assert (Y5 = x2A6091AE);
-- 7th MainLoop iteration
X6, Y6, V6 = mainLoop (X5, Y5, V5, W, B);
assert (X6 = xF0239DD5);
assert (Y6 = x3DD81AC6);
-- 8th MainLoop iteration
X7, Y7, V7 = mainLoop (X6, Y6, V6, W, B);
assert (X7 = xEB35B97F);
assert (Y7 = x9372CDC6);
-- 9th MainLoop iteration
X8, Y8, V8 = mainLoop (X7, Y7, V7, W, B);
assert (X8 = x4DA124A1);
assert (Y8 = xC6B1317E);
-- 10th MainLoop iteration
X9, Y9, V9 = mainLoop (X8, Y8, V8, W, B);
assert (X9 = x7F839576);
assert (Y9 = x74B39176);
-- 11th MainLoop iteration
X10, Y10, V10 = mainLoop (X9, Y9, V9, W, B);
assert (X10 = x11A9D254);
assert (Y10 = xD78634BC);
-- 12th MainLoop iteration
X11, Y11, V11 = mainLoop (X10, Y10, V10, W, B);
assert (X11 = xD8804CA5);
assert (Y11 = xFDC1A8BA);
-- 13th MainLoop iteration
X12, Y12, V12 = mainLoop (X11, Y11, V11, W, B);
```

```
  assert (X12 = x3F6F7248);
  assert (Y12 = x11AC46B8);
  -- 14th MainLoop iteration
  X13, Y13, V13 = mainLoop (X12, Y12, V12, W, B);
  assert (X13 = xACBC13DD);
  assert (Y13 = x33D5A466);
  -- 15th MainLoop iteration
  X14, Y14, V14 = mainLoop (X13, Y13, V13, W, B);
  assert (X14 = x4CE933E1);
  assert (Y14 = xC21A1846);
  -- 16th MainLoop iteration
  X15, Y15, V15 = mainLoop (X14, Y14, V14, W, B);
  assert (X15 = xC1ED90DD);
  assert (Y15 = xCD959B46);
  -- 17th MainLoop iteration
  X16, Y16, V16 = mainLoop (X15, Y15, V15, W, B);
  assert (X16 = x3CD54DEB);
  assert (Y16 = x613F8E2A);
  -- 18th MainLoop iteration
  X17, Y17, V17 = mainLoop (X16, Y16, V16, W, B);
  assert (X17 = xBBA57835);
  assert (Y17 = x07C72EAA);
  -- 19th MainLoop iteration
  X18, Y18, V18 = mainLoop (X17, Y17, V17, W, B);
  assert (X18 = xD7843FDC);
  assert (Y18 = x6AD6E8A4);
  -- 20th MainLoop iteration
  X19, Y19, V19 = mainLoop (X18, Y18, V18, W, B);
  assert (X19 = x5EBA06C2);
  assert (Y19 = x91896CFA);
  -- Coda: MainLoop iteration with S
  X20, Y20, V20 = mainLoop (X19, Y19, V19, W, S);
  assert (X20 = x1D9C9655);
  assert (Y20 = x98D1CC75);
  -- Coda: MainLoop iteration with T
  X21, Y21, V21 = mainLoop (X20, Y20, V20, W, T);
  assert (X21 = x7BC180AB);
  assert (Y21 = xA0B87B77);

  assert (coda (X19, Y19, V19, W, S, T) = xDB79FBDC);

  res = true;

tel;
```

A.17 Functional Testing of MAA with Lurette

We automate the test execution process, by using the testing tool Lurette [JRB06], for this purpose we added a function selecting a message (`get_mess_key`), and adapted the main node `MAC`.

```
function get_mess_key (id: int) returns (message: Block; JK: Key);
let
  message = if (id = 1 or id = 22) then x55555555
            else if (id = 2 or id = 12) then xAAAAAAAA
            else if (id = 3 or id = 42) then x00000000
            else xFFFFFFFF;
  JK = if (id = 1 or id = 12 or id = 2 or id = 22) then
         Key {K = x00000000; J = x00FF00FF}
       else Key {K = x5A35D667; J = x55555555};
tel;
-----------------------------------------
node MAC (id: int; init: bool)
        returns (X, Y, V, W, S, T, Z: Block; n: int);
var X0, Y0, V0: Block;
    KJ: Key; Mn: Block;
let
  Mn, KJ = get_mess_key (id);
  n = if init then 1
      else if pre n = 256 then 0
      else pre n + 1;
  -- initialisations
  X0, Y0, V0, W, S, T = prelude (KJ.J, KJ.K);
  -- mainloops
  X, Y, V = if init then
              mainLoop (X0, Y0, V0, W, Mn)
            else if n = 0 then
              -- mode of operations
              mainLoop2 (X0, Y0, V0, W, pre Z, Mn)
            else mainLoop (pre X, pre Y, pre V, W, Mn);
  -- coda
  Z = coda (X, Y, V, W, S, T);
tel;
-----------------------------------------
node oracle (id: int; init: bool; X, Y, Z: Block; n: int)
            returns (res: bool);
let
  res = true ->
   ((id = 1 and init and X = x48B204D6 and Y = x5834A585) or
    (id = 12 and not init and X = x4F998E01 and Y = xBE9F0917) or
```

```
    (id = 12 and Z = xF14D6E28) or
    (id = 2 and init and X = x6AEBACF8 and Y = x9DB15CF6) or
    (id = 22 and not init and X = x270EEDAF and Y = xB8142629) or
    (id = 22 and Z = xA93BD410) or
    (id = 3 and init and X = x2FD76FFB and Y = x550D91CE) or
    (id = 32 and not init and X = xA70FC148 and Y = x1D10D8D3) or
    (id = 32 and Z = xB99A62DE) or
    (id = 4 and init and X = x8DC8BBDE and Y = xFE4E5BDD) or
    (id = 42 and not init and X = xCBC865BA and Y = x0297AF6F) or
    (id = 42 and Z = xA018C83B)
    and CHECK () = true);
tel;
```

Appendix B

Formal Model of a Simple Autonomous Car in GRL

This appendix contains the sources for the running example of the Chapter 5. Precisely, this directory contains: the GRL model the autonomous car with required LNT libraries and the the XTL scripts to extract the synchronous test scenarios.

B.1 Definitions for the Synchronous Blocks

We define here each synchronous component of the autonomous car; `ACTION`, `RADAR`, `DECISION`, and `GPS` as a block. In order to model the environmnet, we also define three others blocks:

- `OBSTACLE_EXECUTION` block manages an obstacle action.
- `INIT_BLOCKS` block initialise and print the component information; i.e., the names of their inputs and their initialisation values.
- `FIN_CAR` block is activated when the car arrived, or a collision occurred between the car and an obstacle, or all obstacles have finished their list of movements.

```
block RADAR (in POSITIONS: Car_Obstacle_Pos) [send CURRENT_GRID: Grid]
is
  static var PREVIOUS_GRID: Grid := Grid (NIL)
  var grid: Grid
  grid := perception (POSITIONS, radar_visibility);
  if grid != PREVIOUS_GRID then
    PREVIOUS_GRID := grid;
    CURRENT_GRID := grid
  else
```

```
    CURRENT_GRID := Grid (already_sent)
  end if
end block
-----------------------------------------------
block GPS (in UPDATE_POSITION: Edge)
          [receive REQUESTED_POS: bool,
           send CURRENT_POSITION: Edge]
is
  static var CUR_POS: Edge := initial_street
  if (REQUESTED_POS == true) and
      (UPDATE_POSITION != CUR_POS) then
    CUR_POS := UPDATE_POSITION
  else
    CUR_POS := already
  end if;
  CURRENT_POSITION := CUR_POS
end block
-----------------------------------------------
block DECISION (out ARRIVAL: bool)
               [receive PATH_AVOIDING: Edges,
                send REQUEST_POS: bool,
                send CURRENT_PATH: Itinerary,
                receive CURRENT_POSITION: Edge]
is
  var map: Localization := Localization (initial_street, initial_map),
      dest: Edge
  dest := destination;
  REQUEST_POS := true;
  if CURRENT_POSITION == dest then
    -- the car arrived to the final destination
    CURRENT_PATH := NIL;
    ARRIVAL := true
  else
    ARRIVAL := false;
    -- compute an itineray and send it to the driving controller
    CURRENT_PATH := compute_itinary (map.G,
                                     CURRENT_POSITION, dest,
                                     PATH_AVOIDING) -- streets to avoid
  end if
end block
-----------------------------------------------
block ACTION (out CAR_MOVE: Control)
            [send REQUEST_PATH: Edges,
             receive CURRENT_GRID: Grid,
             receive CURRENT_PATH: Itinerary]
```

```
is
  -- check feasibility of the itinerary
  REQUEST_PATH := get_grid (CURRENT_GRID);
  if CURRENT_PATH != NIL then
    CAR_MOVE := car_controls (head (CURRENT_PATH))
  else
    CAR_MOVE := brakes
  end if
end block
-----------------------------------------------
block OBSTACLE_EXECUTION (in OBSTACLE_i: Obstacle_Info,
                          out OBSTACLE_MOVE: Obstacle_Wait)
is
  var args: Obstacle_Wait,
      no_action: Bool,
      op: Obstacle
    op := get_obstacle_inf (OBSTACLE_i);
    if (op == null_obstacle) then
      no_action := True;
      args := obst_wait (OBSTACLE_i.o, true)
    else
      args := obstacle_movement (OBSTACLE_i.car,
                                 OBSTACLE_i.obstacles,
                                 OBSTACLE_i.o);
      no_action := get_action (args)
    end if;
    if (no_action == False) then
      OBSTACLE_MOVE := args
    else
      OBSTACLE_MOVE := obst_wait (OBSTACLE_i.o, true)
    end if
end block
-----------------------------------------------
block INIT_BLOCKS (in INIT_V: Block_Init, out INIT_E: Block_Sent)
is
  var grid_obstacle: Grid := Grid (already_sent),
      v1: String,
      v2: String
  v1 := ret_b1(INIT_V);
  v2 := ret_b2(INIT_V);
  -- INIT_E := INIT_V
  INIT_E := already_values (v1, grid_obstacle, v2, already)
end block
-----------------------------------------------
block FIN_CAR (in LOG_CAR: LOG_END_CAR, out FIN_CAR: LOG_END_CAR)
```

```
is
  FIN_CAR := LOG_CAR
end block
```

B.2 Definitions for the Asynchronous Mediums

The blocks exchange data via asynchronous communication media, which is represented in GRL as a **medium**. We define here the mediums `POSITION`, `PATH`, and `CURRENT_GRID`.

```
-----------------------------------------------
medium Medium_CURRENT_GRID [receive Input : Grid,
                            send Output : Grid] is
  static var Buffer: Grid := Grid (NIL)
  select
    when ?Input -> if Input != Grid (already_sent) then
                     Buffer := Input
                   end if
  []
    -- Value emission
    when Output -> Output := Buffer
  end select
end medium
-----------------------------------------------
medium Medium_POSITION [receive Input: Edge,
                        receive REQUEST: bool,
                        send POS_REQUESTED: bool,
                        send Output: Edge] is
  static var Buffer: Edge := initial_street,
             REQUEST_POS: bool := false
  select
    when ?REQUEST -> REQUEST_POS := REQUEST
  []
    when POS_REQUESTED -> POS_REQUESTED := REQUEST_POS
  []
    when ?Input -> if Input != already then
                     Buffer := Input
                   end if
  []
    -- Value emission
    when Output -> Output := Buffer
  end select
end medium
-----------------------------------------------
medium Medium_PATH [receive REQUEST_PATH: Edges,
```

```
                send PATH_AVOIDING: Edges,
                receive SEND_PATH: Itinerary,
                send RECEIVE_PATH: Itinerary] is
  static var Buffer: Itinerary := init_itinary,
             l_avoid: Edges := NIL,
             itinerary_requested: bool := false
  select
    when ?REQUEST_PATH -> l_avoid := REQUEST_PATH;
                          itinerary_requested := true
  []
    when PATH_AVOIDING -> PATH_AVOIDING := l_avoid;
                          l_avoid := NIL
  []
    when ?SEND_PATH -> if itinerary_requested == true then
                         Buffer := SEND_PATH;
                         itinerary_requested := false
                       end if
  []
    -- Value emission
    when RECEIVE_PATH -> RECEIVE_PATH := Buffer;
                         Buffer := NIL
  end select
end medium
```

B.3 Definitions for the Asynchronous Environement

The interaction between blocks also respects environmental constraints, each one being represented in GRL as an **environment**. We present here the environment `MAP_MANAGEMENT` managing the positions of the car and obstacles on the map.

```
-------------------------------------------
environment Env_MAP_MANAGEMENT (block ACTION,
                                block DECISION,
                                block RADAR,
                                block GPS,
                                block OBSTACLE_EXECUTION,
                                block FIN_CAR,
                                block INIT_BLOCKS,
                                in CAR_MOVE: Control,
                                in OBSTACLE_MOVE: Obstacle_Wait,
                                in ARRIVAL: bool,
                                in FC: LOG_END_CAR,
                                in E: Block_Sent,
                                out UPDATE_POSITION: Edge,
```

```
                                out POS_GRID: Car_Obstacle_Pos,
                                out OBSTACLE_i: Obstacle_Info,
                                out FIN: LOG_END_CAR,
                                out V: Block_Init)
is
  static var grid: Grid := Grid (NIL),
             map: Localization := Localization (initial_street, initial_map),
             crash: bool := false,
             car_arrived: bool := false,
             init: Bool := True,
             obstacles: The_Obstacles := init_obstacles,
             nb_obst : Nat := lgth_obst_grid (init_obstacles),
             log_end : LOG_END_CAR := END_OBSTACLE
  var collision_detected: Bool, l_obst: The_Obstacles,
      oi: Obstacle, waiting: Bool, b: Obstacle_behaviour,
      opi: Operation, n1: Nat, l_break: bool := true
  if init == True then
   select
     enable INIT_BLOCKS
   []
     when V -> V := init_block_values ("RADAR", pos (map.c, grid), "GPS", map.c)
   []
     when ?E -> init := get_false (E)
   end select
  else
   if (crash == false) and (car_arrived == false) and (nb_obst > 0) then
     select
       enable ACTION
     []
       enable DECISION
     []
       enable RADAR
     []
       enable GPS
     []
       enable OBSTACLE_EXECUTION
     []
       -- initialize the position of the car: the map in the car's GPS does not
       -- change, thus send only the current street
       when UPDATE_POSITION -> UPDATE_POSITION := get_localizalitation(map)
     []
       -- handle car movement
       when ?CAR_MOVE ->
         -- compute car position
         map := move_car (map, CAR_MOVE);
```

```
    collision_detected := intersection (grid, map);
    -- check if the car is in the same edge as an obstacle
    if collision_detected == True then
      crash := true;
      log_end := COLLISION
    end if
[]
  -- handle car movement
  when ?ARRIVAL -> car_arrived:= ARRIVAL;
                   log_end := ARRIVED
[]
  -- after a car movement, update the radar grid
  when POS_GRID -> POS_GRID := pos (map.c, grid)
[]
  -- select an obstacle movement (the car and obstacles positions)
  when OBSTACLE_i ->
                   oi := null_obstacle;
                   l_obst := obstacles;
                   while (l_break) and (l_obst != NIL) loop
                    b := get_obstacle_behaviour (head (l_obst));
                    if b != NIL then
                      oi := head (l_obst);
                      select
                       l_break := false
                      []
                        l_obst := tail (l_obst)
                      end select
                    else
                      l_obst := tail (l_obst)
                    end if
                   end loop;
                   b := get_obstacle_behaviour (oi);
                   if (oi != null_obstacle) and (b != NIL) then
                     opi := head (b);
                     if opi == random then
                       l_break := true;
                       n1 := next_street (map, oi);
                       while (l_break == true) loop
                         if (n1 > 0) then
                           select
                             n1 := n1 - 1
                           []
                             opi := turned_n (n1); l_break := false
                           []
                             opi := leave; l_break := false
```

```
                        end select
                      else
                        opi := turned_n (n1); l_break := false
                      end if
                    end loop;
                    oi := Obstacle (oi.name, oi.position,
                                    cons (opi, tail(b)))
                  end if
                end if;
                OBSTACLE_i := obst_inf (oi, map.c, grid)
  []
        -- handle obstacle movement
    when ?OBSTACLE_MOVE ->
        -- compute new grid for the radar and
        -- inform the radar and the obstacles about the change
        waiting := get_action(OBSTACLE_MOVE);
        if (waiting == False) then
          grid := move_obstacle_grid (grid, OBSTACLE_MOVE.o);
          obstacles := update_obstacle_in_L (obstacles, OBSTACLE_MOVE.o)
        else
          grid := grid
        end if;
        b := get_obstacle_behaviour (OBSTACLE_MOVE.o);
        if b == NIL then
          nb_obst := nb_obst - 1
        end if
    end select
  else
    select
      enable FIN_CAR
    []
      when FIN -> FIN := log_end
    []
      when ?FC -> log_end := FC
    end select
  end if
 end if
end environment
-----------------------------------------------------
```

B.4 Definitions of the global GALS system

The overall model of the GALS is represented in GRL as a **system**, which describes the composition and interactions of blocks, media, and environments. We present here the system `Main` of our GRL model.

```
-----------------------------------------------
system Main (SEND_CURRENT_GRID, RECEIVE_CURRENT_GRID: Grid,
            UPDATE_POSITION: Edge,
            SEND_CURRENT_POSITION, RECEIVE_CURRENT_POSITION: Edge,
            POSITIONS: Car_Obstacle_Pos, REQUEST_PATH: Edges,
            PATH_AVOIDING: Edges, SEND_PATH, RECEIVE_PATH: Itinerary,
            CAR_MOVE: Control, ARRIVAL: bool,
            REQUEST_POS: bool, REQUESTED_POS: bool,
            OBSTACLE_MOVE: Obstacle_Wait, OBSTACLE_i: Obstacle_Info,
            LOG_CAR, FIN_CAR: LOG_END_CAR, V: Block_Init, E: Block_Sent)
is
   alias
     ACTION as ACT, OBSTACLE_EXECUTION as OBST_EXEC, DECISION as DECI,
     RADAR as PER_RADAR, GPS as PER_GPS, FIN_CAR as FN, INIT_BLOCKS as INIT
   block list
     PER_RADAR (POSITIONS) [?SEND_CURRENT_GRID],
     PER_GPS (UPDATE_POSITION) [REQUESTED_POS, ?SEND_CURRENT_POSITION],
     DECI (?ARRIVAL)
          [PATH_AVOIDING, ?REQUEST_POS, ?SEND_PATH, RECEIVE_CURRENT_POSITION],
     ACT (?CAR_MOVE) [?REQUEST_PATH, RECEIVE_CURRENT_GRID, RECEIVE_PATH],
     OBST_EXEC (OBSTACLE_i, ?OBSTACLE_MOVE),
     FN (LOG_CAR, ?FIN_CAR),
     INIT (V, ?E)
   environment list
     Env_MAP_MANAGEMENT (ACT, DECI, PER_RADAR, PER_GPS, OBST_EXEC, FN, INIT,
                         CAR_MOVE, OBSTACLE_MOVE, ARRIVAL, FIN_CAR, E,
                         ?UPDATE_POSITION, ?POSITIONS, ?OBSTACLE_i,
                         ?LOG_CAR, ?V)
   medium list
     Medium_CURRENT_GRID [SEND_CURRENT_GRID, ?RECEIVE_CURRENT_GRID],
     Medium_POSITION [SEND_CURRENT_POSITION, REQUEST_POS,
                      ?REQUESTED_POS, ?RECEIVE_CURRENT_POSITION],
     Medium_PATH [REQUEST_PATH, ?PATH_AVOIDING, SEND_PATH, ?RECEIVE_PATH]
end system
```

B.5 Definitions of LNT library

We define here the LNT types together with functions related to this case study, and the LNT functions defining the scenario used in this case study.

B.5.1 Definitions of LNT Graph Functions and Types

We define here the types related to graph definitions together with the classical graph functions.

```
type Corner is
  0,
  1,
  2,
  3,
  4,
  5,
  6,
  7,
  8
end type
-----------------------------------------------
type Edge is
  Edge (q1: Corner, l: Street, q2: Corner)
end type
-----------------------------------------------
type Edges is
-- hash-table used as cache
!card 100000
  list of Edge
with "head", "tail", "reverse", "length"
end type
-----------------------------------------------
type Vertices is
  list of Corner
with "head", "tail"
end type
-----------------------------------------------
type Graph is
-- Digraph (directed graph)
  Graph (V: Vertices, E: Edges)
end type
-----------------------------------------------
-- Functions
```

```
-----------------------------------------------
function add_Edge (q1, q2: Corner, l: Street, e: Edges): Edges is
  return cons (Edge (q1, l, q2), e)
end function
-----------------------------------------------
function succ_s (in var E: Edges, s: Corner) : Vertices is
  var list_succ: Vertices in
    list_succ := {};
    loop
      case E in
      var q1, q2: Corner in
        NIL -> return list_succ
      | CONS (Edge (q1, any Street, q2), E) ->
          if q1 == s then
            list_succ := cons (q2, list_succ)
          end if
      end case
    end loop
  end var
end function
-----------------------------------------------
function succ_l (in var L: Edges, e: Edge) : Edges is
  var list_succ: Edges in
    list_succ := {};
    loop
      case L in
      var q1, q2: Corner, s: Street in
        NIL -> return reverse (list_succ)
      | CONS (Edge (q1, s, q2), L) ->
          if q1 == e.q2 then
            list_succ := cons (Edge (q1, s, q2), list_succ)
          end if
      end case
    end loop
  end var
end function
-----------------------------------------------
function succ_l_s (in var E: Edges, l: Street) : Corner is
  -- return succesor of a label
  loop
    assert E != {};
    if head(E).l == l then
      return head(E).q2
    end if;
    E := tail (E)
```

```
  end loop
end function
-----------------------------------------------
function edge_l (in var E: Edges, l: Street) : Edge is
  -- return succesor of a label
  loop
    assert E != {};
    if head(E).l == l then
      return head(E)
    end if;
    E := tail (E)
  end loop
end function
-----------------------------------------------
function is_l_in_E (in var E: Edges, l: Street) : Bool is
  loop
    if E == {} then
      return false
    else
      if head(E).l == l then
        return true
      end if;
      E := tail(E)
    end if
  end loop
end function
-----------------------------------------------
function is_e_l (e: Edge, in var L: Edges) : Bool is
  loop L1 in
    if L == {} then
      break L1
    end if;
    if head(L) == e then
      return true
    end if;
    L := tail (L)
  end loop;
  return false
end function

-----------------------------------------------

function is_s_l (S: Corner, in var V: Vertices) : Bool is
  loop L1 in
    if V == {} then
```

```
      break L1
    end if;
    if head(V) == S then
      return true
    end if;
    V := tail (V)
  end loop;
  return false
end function
-------------------------------------------------
function pred_edge (E: Edges, in var D: Edge)
: Edges is
  -- retrace predecessors leading to D
  var pred, temp: Edges in
    pred := {D};
    temp := E;
    loop L in
      if temp == {} then
        return pred
      end if;
      if (head (temp).q2 == D.q1) then
        D := head(temp);
        pred := cons (D, pred)
      else
        temp := tail (temp)
      end if
    end loop
  end var
end function

-------------------------------------------------

function delete_l_in_E (in var E: Edges, l: Street) : Edges is
  var new: Edges in
    new := {};
    loop
      if E == {} then
        return new
      else
        if head(E).l != l then
          new := cons (head(E), new)
        end if;
        E := tail(E)
      end if
    end loop
```

```
  end var
end function
-----------------------------------------------
function is_q1q2_in_E (in var E: Edges, A: Edge) : Bool is
  loop
    if E == {} then
      return false
    else
      if (head(E).q1) == A.q1 and (head(E).q2 == A.q2) then
        return true
      end if;
      E := tail(E)
    end if
  end loop
end function
-----------------------------------------------
```

B.5.2 Definitions of LNT Datatypes

The definition of the main types related to this case study together with heir functions.

```
-----------------------------------------------
type Localization is !printedby "PRINT_LOCALIZATION"
  -- global localization (GPS)
  -- localisation (graph with anotate state for the current position)
  Localization (c: Edge,
                G: Graph)
end type
-----------------------------------------------
type Grid is
  -- local localization
  Grid (E: Edges)
end type
-----------------------------------------------
type Direction is
  turn_n (N: Nat) -- 5th if crossroads
end type
-----------------------------------------------
type Itinerary is
  list of Direction
with "head", "tail", "reverse"
end type
-----------------------------------------------
type Control is
```

```
  -- direction
  turned_n (N: Nat),
  brakes
end type
-----------------------------------------------
type Operation is
  -- obstacle operations
  turned_n (N: Nat), -- 5th if crossroads,
  leave,
  random
end type
-----------------------------------------------
type Obstacle_behaviour is
  list of Operation
with "head", "tail"
end type
-----------------------------------------------
type Obstacle is
  Obstacle (name: Street, position: Edge, behaviour: Obstacle_behaviour)
end type
-----------------------------------------------
type The_Obstacles is
  list of Obstacle
with "head", "tail", "length"
end type
-----------------------------------------------
type LOG_END_CAR is
  COLLISION,
  ARRIVED,
  END_OBSTACLE
end type
-----------------------------------------------
type Obstacle_Wait is
  obst_wait (o: Obstacle, waiting: bool)
end type
-----------------------------------------------
type Obstacle_Info is
  obst_inf (o: Obstacle, car: Edge, obstacles: Grid)
end type
-----------------------------------------------
type Car_Obstacle_Pos is
  pos (car: Edge, obstacles: Grid)
end type
-----------------------------------------------
type Block_Init is
```

```
   init_block_values (b1: String, v1: Car_Obstacle_Pos, b2: String, v2: Edge)
end type
-----------------------------------------------
type Block_Sent is
   already_values (b1: String, v1: Grid, b2: String, v2: Edge)
end type
-----------------------------------------------
-- Functions
-----------------------------------------------
function succ_avoid_reach_D (in E: Edges, P: Edge, D: Edge, avoid: Edges)
: Edges is
  -- compute successors leading to D avoiding the edges in a given list
  var a: Edge, pred, succ, succ_t: Edges, dejavu: Vertices in
    if is_e_l (P, E) == false then
      return {}
    end if;
    a := P;
    pred := {a};
    succ := succ_l (E, a);
    succ_t := succ;
    dejavu := {a.q2};
    loop
      if is_e_l (D, succ_t) then
        return pred_edge (pred, D)
      elsif succ == {} then
        return {}
      else
        a := head (succ);
        succ := tail(succ);
        if (is_s_l (a.q2, dejavu) == false) and
                  (is_e_l (a, avoid) == false) then
          succ_t := succ_l (E, a);
          pred := cons (a, pred);
          succ := union (succ_t, succ);
          dejavu := cons (a.q2, dejavu)
        end if
      end if
    end loop
  end var
end function
-----------------------------------------------
function edges_to_itinerary (map: Localization, in var succ: Edges)
: Itinerary is
  var eg: Edges, it: Itinerary, I: Nat in
    it := {};
```

```
    eg := map.G.E;
    loop
      if (succ == {}) or (eg == {}) then
        return reverse (it)
      end if;
      if head (eg) == head (succ) then
        I:= 0;
        succ := tail (succ);
        loop L in
          if (succ != {}) then
            eg := tail (eg);
            if eg == {} then
              eg := map.G.E
            end if;
            if head(eg).q1 == head (succ).q1 then
              if head(eg).q2 == head (succ).q2 then
                it := cons (turn_n (I), it);
                break L
              end if;
              I := I + 1
            end if
          else
            return it
          end if
        end loop;
        eg := map.G.E
      else
        eg := tail (eg)
      end if
    end loop
  end var
end function
-----------------------------------------------
function compute_itinary (map: Graph, position, destination: Edge,
                          avoid: Edges)
: Itinerary is
  -- return if possible an itinerary leading from S to D in map
  var succ: Edges in
    succ := succ_avoid_reach_D (map.E, position, destination, avoid);
    return edges_to_itinerary (Localization (position, map), succ)
  end var
end function
-----------------------------------------------
function car_controls (dir: Direction) : Control is
  case dir in
```

```
  var N: Nat in
    turn_n (N) -> return turned_n (N)
  end case
end function
-----------------------------------------------
function move_n (G: Graph, P: Edge, N: Nat) : Edge is
  -- return the car advancing in the n-th neighbourd
  var eg: Edges, i: Nat in
    eg := G.E;
    i := 0;
    loop L in
      if eg == {} then
        -- no successor found: do not move
        return P
      end if;
      if head (eg).q1 == P.q2 then
        if i == N then
          return head(eg)
        else
          i := i + 1
        end if
      end if;
      eg := tail (eg)
    end loop
  end var
end function
-----------------------------------------------
function move_car (map: Localization, action: Control) : Localization is
  case action in
  var N: Nat in
    turned_n (N) -> return Localization (move_n (map.G, map.c, N), map.G)
  | any -> return map
  end case
end function
-----------------------------------------------
function move_obstacle (g: Graph, movement: Operation, o: Obstacle)
: Edge is
  case movement in
  var N: Nat in
    turned_n (N) -> return move_n (g, o.position, N)
  | leave -> return Edge (0, HIDDEN, 1)
  | random -> return Edge (0, HIDDEN,1)
  end case
end function
-----------------------------------------------
```

```
function move_obstacle_grid (in var maj_radar: Grid,
                             obst: Obstacle)
: Grid is
  -- return the same obstacle position exists
  if is_q1q2_in_E (maj_radar.E, obst.position) then
    return maj_radar
  end if;
  -- delete the previous obstacle position
  maj_radar := Grid (delete_l_in_E (maj_radar.E, obst.name));
  -- add the new obstacle position
  return Grid (add_Edge (obst.position.q1, obst.position.q2,
                         obst.name, maj_radar.E))
end function
-----------------------------------------------
function succ_edge (in var R: Grid, D: Edge)
: Grid is
  -- sucesseur of D
  var succ: Edges, temp: Edge in
    succ := {};
    loop L in
      if R.E == {} then
        return Grid (succ)
      end if;
      temp := head (R.E);
      if (temp.q1 == D.q2) then
        succ := cons (temp, succ)
      end if;
      R := R.{E => tail (R.E)}
    end loop
  end var
end function
-----------------------------------------------
function succ_n_depth (in R: Grid, in var D: Edge, in var n: Nat)
: Grid is
  var succ, succ_n, succ_t: Edges in
    succ := (succ_edge (R, D)).E;
    succ_n := succ;
    succ_t := {};
    loop
      if n == 0 then
        return Grid (succ)
      end if;
      n := n - 1;
      loop L in
        if (succ_n == {}) or (n == 0) then
```

```
          break L
        end if;
        D := head (succ_n);
        succ_n := tail (succ_n);
        succ_t := union (succ_t, (succ_edge (R, D)).E)
      end loop;
        succ := union (succ, succ_t);
        succ_n := succ_t;
        succ_t := {}
    end loop
  end var
end function
-----------------------------------------------
function perception (POSITIONS: Car_Obstacle_Pos, VISIBILITY: Nat)
: Grid is
  return succ_n_depth (POSITIONS.obstacles, POSITIONS.car, VISIBILITY)
end function
-----------------------------------------------
function obstacle_in_L (in var l_obs: The_Obstacles, l: Street)
: Obstacle is
  -- return Obstacle corresponding to a name
  loop
    assert l_obs != {};
    if head(l_obs).name == l then
      return head(l_obs)
    end if;
    l_obs := tail (l_obs)
  end loop
end function
-----------------------------------------------
function update_obstacle_in_L (in var l_obs: The_Obstacles,
                               o: Obstacle)
: The_Obstacles is
  -- return the obstacles list updated
  var up_l: The_Obstacles in
    up_l := {};
    loop L in
      if l_obs == {} then
        break L
      end if;
      if head(l_obs).name == o.name then
        up_l := cons (o, up_l)
      else
        up_l := cons (head(l_obs), up_l)
      end if;
```

```
      l_obs := tail (l_obs)
    end loop;
    return up_l
  end var
end function
-----------------------------------------------
function is_q1q2_in_R (R: Grid, A: Localization) : bool is
  var E : Edges, P: Edge in
    E := R.E;
    P := A.C;
    loop
      if E == {} then
        return false
      else
        if (head(E).q1) == P.q1 and (head(E).q2 == P.q2) then
          return true
        end if;
        E := tail(E)
      end if
    end loop
  end var
end function
-----------------------------------------------
function intersection (R: Grid, A: Localization) : bool is
  return is_q1q2_in_R (R, A)
end function
-----------------------------------------------
function get_localizalitation (L: Localization) : Edge is
  return L.C
end function
-----------------------------------------------
function get_graph (L: Localization) : Graph is
  return L.G
end function
-----------------------------------------------
function get_grid (R: Grid) : Edges is
  return R.E
end function
-----------------------------------------------
function get_obstacle (OW: Obstacle_Wait) : Obstacle is
  return OW.o
end function
-----------------------------------------------
function get_action (OW: Obstacle_Wait) : bool is
  return OW.waiting
```

```
end function
-----------------------------------------------
function get_obstacle_behaviour (O: Obstacle) : Obstacle_behaviour is
  return O.behaviour
end function
-----------------------------------------------
function null_obstacle : Obstacle is
  return Obstacle (NO_OBSTACLE, Edge (0, DIED ,0), {})
end function
-----------------------------------------------
function lgth_obst_grid (l: The_Obstacles) : Nat is
  return length (l)
end function
-----------------------------------------------
function is_obst_in_L (in var L: The_Obstacles, o: Obstacle) : bool is
  loop
    if L == {} then
      return false
    else
      if head(L).name == o.name then
        return true
      end if;
      L := tail(L)
    end if
  end loop
end function
-----------------------------------------------
function get_map_streets (map: Localization) : Edges is
    return map.G.E
end function
-----------------------------------------------
function already: Edge is
    return Edge (0, NO_UPDATE,0)
end function
-----------------------------------------------
function already_sent: Edges is
    return {already}
end function
-----------------------------------------------
function next_street (map: Localization, o: Obstacle) : Nat is
  return length (succ_l (get_map_streets(map), o.position))
end function
-----------------------------------------------
function get_obstacle_inf (obi: Obstacle_Info) : Obstacle is
  return obi.o
```

```
end function
-----------------------------------------------
function get_false (b: Block_Sent) : bool is
  use b;
  return false
end function
-----------------------------------------------
function ret_b1 (b: Block_Init) : String is
  return b.b1
end function
-----------------------------------------------
function ret_b2 (b: Block_Init) : String is
  return b.b2
end function
-----------------------------------------------
```

B.5.3 Definitions of the Scenario

We present here all functions corresponding to the scenario information of our model, which is modifiable such as the geographical map, the obstacles behaviors, the initial and destination streets.

```
-----------------------------------------------
-- Geographical map
-----------------------------------------------
-- ATTENTION : the edges must be ordered by increasing source state, and
-- the by increasing target state

function initial_map : Graph is
  var e: Edges, v: Vertices in
    v := {0, 1, 2, 3, 4, 5, 6, 7, 8};
    e := {Edge (0, Coronation_Street, 1),
          Edge (0, Corporation_Street, 3),
          Edge (1, Coronation_Street_bis, 0),
          Edge (1, two_Coronation_Street, 2),
          Edge (1, Sackville, 4),
          Edge (2, two_Coronation_Street_bis, 1),
          Edge (2, Spring_Gardens, 5),
          Edge (3, Corporation_Street_bis, 0),
          Edge (3, Princess_Street, 4),
          Edge (3, two_Corporation_Street, 6),
          Edge (4, two_Princess_Street, 5),
          Edge (4, two_Sackville, 7),
          Edge (5, Spring_Gardens_bis, 2),
```

```
        Edge (5, two_Princess_Street_bis, 4),
        Edge (5, two_Spring_Gardens, 8),
        Edge (6, two_Corporation_Street_bis, 3),
        Edge (6, New_Cathedral_Street, 7),
        Edge (7, two_Sackville_bis, 4),
        Edge (7, New_Cathedral_Street_bis, 6),
        Edge (7, two_New_Cathedral_Street, 8),
        Edge (8, two_Spring_Gardens_bis, 5),
        Edge (8, two_New_Cathedral_Street_bis, 7)
        };
    return Graph (v, e)
  end var
end function
-----------------------------------------------
-- Radar visibility
-----------------------------------------------
function radar_visibility: Nat is
  return 1
end function
-----------------------------------------------
-- Position initial
-----------------------------------------------
function initial_street: Edge is
  return Edge (3, Princess_Street, 4)
end function
-----------------------------------------------
-- Destination
-----------------------------------------------
function destination: Edge is
  -- return Edge (5, Spring_Gardens_bis, 2)
  -- return Edge (2, Spring_Gardens, 5)
  -- return Edge (8, two_Spring_Gardens_bis, 5)
  return Edge (0, Coronation_Street, 1)
end function
-----------------------------------------------
-- Init itinary
-----------------------------------------------
function init_itinary: Itinerary is
  var it: Itinerary in
    it := compute_itinary (initial_map, initial_street, destination, {});
    assert it != {};
    return it
  end var
end function
-----------------------------------------------
```

```
-- Obstacles
-------------------------------------------------
function init_obstacles: The_Obstacles is
  var b1, b2: Obstacle_behaviour, baby, cyclist: Obstacle in
    -- initialization of obstacle behaviours
    b1 := {random, turned_n (0)};
    b2 := {random, turned_n (0)};
    -- initialization of obstacle positions
    -- baby := Obstacle (Lilly, Edge (1, Sackville, 4), b1);
    -- cyclist := Obstacle (Leo, Edge (5, two_Princess_Street_bis, 4), b2);
    -- baby := Obstacle (Lilly, Edge (1, Coronation_Street_bis, 0), b1);
    -- cyclist := Obstacle (Leo, Edge (4, two_Princess_Street, 5), b2);
    baby := Obstacle (Lilly, Edge (5, Spring_Gardens_bis, 2), b1);
    cyclist := Obstacle (Leo, Edge (2, two_Coronation_Street_bis, 1), b2);
    return {baby, cyclist}
  end var
end function
-------------------------------------------------
-- Functions
-------------------------------------------------
function obstacle_movement (in car_position: Edge,
                           in obstacles_grid: Grid,
                           in var o: Obstacle) : Obstacle_Wait is

  var map: Graph, present: Bool, pos: Edge, outputs: Obstacle_Wait in
    map := initial_map;
    present := is_l_in_E (obstacles_grid.E, o.name);
    -- apparition of the obstacle at its initial position
    if not (present or is_q1q2_in_E (cons (car_position, obstacles_grid.E),
                                     o.position))
    then
      outputs := obst_wait (o, false)
    -- movement of the obstacle
    elsif present and (o.behaviour != NIL) then
      -- if o.position == Edge (0, HIDDEN,1) then
        -- pos := move_n (map, car_position, 1)
      -- else
        pos := move_obstacle (map, head (o.behaviour), o);
      -- end if;
      if is_q1q2_in_E (cons (car_position, obstacles_grid.E),
                       pos) == false
      then
        o := Obstacle (o.name, pos, tail (o.behaviour));
        outputs := obst_wait (o, false)
      else
```

```
      outputs := obst_wait (o, true)
    end if
  else
    outputs := obst_wait (o, true)
  end if;
  return outputs
 end var
end function
-----------------------------------------------
```

Appendix C

XTL Scripts Extracting Synchronous Test Scenarios

This appendix contains the XTL [MG98] scripts to extract the synchronous test scenarios. More particularly we present two XTL scripts exploring the test graph (CTG_C) in order to generate the input constraints and the oracle for testing a component C separately.

C.1 The Generation of Input Constraints (extract_input)

We define here the XTL script `extract_constraints` generating the input constraints by encoding a CTG_C as a possibly nondeterministic node in Lutin. Note that this node has the same inputs as C and an additional input variable `s` corresponding to the current state of CTG_C, initialized to the initial state.

```
(*
 * Lutin node "environment" expressing a scenario constraining the three
 * inputs of the synchronous block under test
 * in addition to the inputs of the radar, the node also handles the updates
 * of the state of the complete test graph or test case
 *)
-----------------------------------------
def print_info (p1, p3, p5:raw, p2, p4, p6: natural)
: action =
  printf (" and ") fby
  print (p1) fby printf (" = ") fby print (p2) fby
  printf (" and ") fby
  print (p3) fby printf (" = ") fby print (p4) fby
  printf (" and ") fby
```

```
  print (p5) fby printf (" = ") fby print (p6)
end_def
-----------------------------------------
(* extraction of the names of the three inputs *)
let (iv1, iv2, iv3: raw) =
  for in_e: edge where in_e -> [INPUTS ?iv1:raw _ ?iv2:raw _ ?iv3:raw _]
  apply (replace, replace, replace)
  from (null, null, null)
  to (iv1, iv2, iv3)
  end_for
in
(* extraction of the initial values *)
let (in1, in2, in3: natural) =
  for init_e: edge
    where init_e -> [INITIAL_VALUES _ ?in1:natural _ ?in2:natural _ ?in3:natural]
  apply (replace, replace, replace)
  from (0, 0, 0)
  to (in1, in2, in3)
  end_for
in
-----------------------------------------
printf ("node input_constraints () returns (s, ") fby
print (iv1) fby printf (", ") fby
print (iv2) fby printf (", ") fby
print (iv3) fby printf (": int) =\n") fby

(* initialisation *)
print (iv1) fby printf (" = ") fby print (in1) fby printf (" and ") fby
print (iv2) fby printf (" = ") fby print (in2) fby printf (" and ") fby
print (iv3) fby printf (" = ") fby print (in3) fby
printf (" and s = 0 fby\n") fby

(* main reactive loop *)
printf ("loop {\n") fby
printf (" | pre s = -1 and ") fby
print (iv1) fby printf (" = ") fby print (in1) fby printf (" and ") fby
print (iv2) fby printf (" = ") fby print (in2) fby printf (" and ") fby
print (iv3) fby printf (" = ") fby print (in3) fby
printf (" and s = 0\n") fby
<| fby on e:edge |> (
    printf (" | pre s = ") fby
    print (source (e)) fby
    if (e -> [ INPUTS ?p1:raw ?p2:natural ?p3:raw
                      ?p4:natural ?p5:raw ?p6: natural ]) then
        (* car movement; obstacles do not move *)
```

```
      print_info (p1, p3, p5, p2, p4, p6)
    else
      (* output or verdict transition: keep inputs unchanged *)
      printf (" and ") fby
      print (iv1) fby printf (" = pre ") fby print (iv1) fby
      printf (" and ") fby
      print (iv2) fby printf (" = pre ") fby print (iv2) fby
      printf (" and ") fby
      print (iv3) fby printf (" = pre ") fby print (iv3)
    end_if fby
    (* change s to the target state *)
    printf (" and s = ") fby
    if e -> [INCONCLUSIVE] or e -> [PASS] then
      printf ("-1")
    else
      print (target (e))
    end_if fby
    printf ("\n")
) fby
printf ("}\n")
end_let
end_let
```

C.2 The Generation of Oracles (extract_oracle)

We define here the XTL script extract_oracle generating the oracles, by encoding CTG_C as a deterministic node in Lustre, which, to each corner state and its set of inputs/outputs, associates a Boolean verdict, indicating whether the outputs are the expected ones and also defined for the verdict states coverage of CTG_C.

```
-----------------------------------------
(* a corner state is the target of an input transition and the source of an
 * output transition *)

def corner_state (s: state): boolean =
  exists e:edge among in (s) in e -> [INPUTS ...] end_exists and
  exists e:edge among out (s) in e -> [OUTPUTS ...] end_exists
end_def
-----------------------------------------
def print_corner (soutput: state,
                  iv1, iv2, iv3, ov1, ov2: raw,
                  p1, p2, p3, o1, o2: natural): action =
  printf (" (s = ") fby
  print (soutput) fby printf (" and\n ") fby
```

```
  print (iv1) fby printf (" = ") fby print (p1) fby printf (" and ") fby
  print (iv2) fby printf (" = ") fby print (p2) fby printf (" and ") fby
  print (iv3) fby printf (" = ") fby print (p3) fby printf (" and ") fby
  print (ov1) fby printf (" = ") fby print (o1) fby printf (" and ") fby
  print (ov2) fby printf (" = ") fby print (o2) fby printf (")\n or")
end_def
-----------------------------------------
macro verdict_edge (e) =
  (((e) -> [INCONCLUSIVE]) or ((e) -> [PASS]))
end_macro

def verdict_state (s: state): boolean =
  exists e:edge among out (s) in verdict_edge (e) end_exists and
  forall e:edge among out (s) in verdict_edge (e) end_forall
end_def

def nb_verdicts (s: state): natural =
  <| + on e: edge among out (s) where verdict_edge (e) |> 1
end_def
-----------------------------------------
def print_inconclusive (n: natural, iv1, iv2, iv3: raw): action =
  if n > 1 then
    printf (" or\n")
  else
    nop
  end_if fby
  printf (" ") fby
  print (iv1) fby printf (" = pre ") fby print (iv1) fby printf (" and ") fby
  print (iv2) fby printf (" = pre ") fby print (iv2) fby printf (" and ") fby
  print (iv3) fby printf (" = pre ") fby print (iv3)
end_def

def print_pass (n: natural): action =
  if n > 1 then
    printf (" or\n")
  else
    nop
  end_if fby
  printf (" true")
end_def
-----------------------------------------
(* extraction of the names of the three inputs *)
let (iv1, iv2, iv3: raw) =
  for in_e: edge where in_e -> [INPUTS ?iv1:raw _ ?iv2:raw _ ?iv3:raw _]
  apply (replace, replace, replace)
```

```
  from (null, null, null)
  to (iv1, iv2, iv3)
  end_for
in
(* extraction of the names of the two outputs *)
let (ov1, ov2: raw) =
  for out_e: edge where out_e -> [OUTPUTS ?ov1:raw _ ?ov2:raw _]
  apply (replace, replace)
  from (null, null)
  to (ov1, ov2)
  end_for
in
(* extraction of the initial values *)
let (in1, in2, in3: natural) =
  for init_e: edge
    where init_e -> [INITIAL_VALUES _ ?in1:natural _ ?in2:natural _ ?in3:natural]
  apply (replace, replace, replace)
  from (0, 0, 0)
  to (in1, in2, in3)
  end_for
in
(* extraction of the already_sent values *)
let (already1, already2: natural) =
  for init_e: edge
    where init_e -> [ALREADY_SENT _ ?already1:natural _ ?already2:natural]
  apply (replace, replace)
  from (0, 0)
  to (already1, already2)
  end_for
in
-----------------------------------------
printf ("node oracle (s, ") fby
print (iv1) fby printf (", ") fby
print (iv2) fby printf (", ") fby
print (iv3) fby printf (", ") fby
print (ov1) fby printf (", ") fby
print (ov2) fby printf (": int)\n") fby
printf ("returns (res, pass, inconclusive") fby
(* output profile containing all states *)
<| fby on s:state |> (
  if ((mod (number (s), 9) = 0) and (number (s) > 0)) then
    printf (",\n s")
  else printf (", s")
  end_if fby
  print (s)
```

```
) fby
printf (", states_covered: bool);\n") fby

(* comment containing the list of corner states *)
printf (" (* corner states:") fby
<| fby on s:state where corner_state (s) |> (
  printf (" ") fby
  print (s)
) fby
printf (" *)\n") fby

(* comment containing the list of verdict states *)
printf (" (* verdict states:") fby
<| fby on s:state where verdict_state (s) |> (
  printf (" ") fby
  print (s)
) fby
printf (" *)\n") fby
printf ("let\n res = true -> (\n ") fby

(* rules for corner states *)
<| fby on s:state where corner_state (s) |>
  (* for each corner state s, iterate over pairs of an input and the output *)
  (* transition *)
  let (a: action, i1, i2, i3: natural) =
    for in_e: edge among in (s), out_e: edge among out (s)
    in (a: action, a_i1, a_i2, a_i3: natural)
    where in_e -> [INPUTS ...] and out_e -> [OUTPUTS ...]
    apply (fby, replace, replace, replace)
    from (nop, 0, 0, 0)
    to
      if in_e -> [INPUTS _ ?i1:natural _ ?i2:natural _ ?i3:natural] then
        if (i1 = a_i1) and (i2 = a_i2) and (i3 = a_i3) then
          (* already handled combination of inputs : skip *)
          (nop, i1, i2, i3)
        else_if out_e -> [OUTPUTS _ ?o1:natural _ ?o2:natural] then
          (print_corner (s, iv1, iv2, iv3, ov1, ov2, i1, i2, i3, o1, o2),
           i1, i2, i3)
        else (* never reached *)
          (nop, a_i1, a_i2, a_i3)
        end_if
      else (* never reached *)
        (nop, a_i1, a_i2, a_i3)
      end_if
    end_for
```

```
  in
    use i1, i2, i3 in
      a
    end_use
  end_let fby

(* rules for verdict states *)
<| fby on s:state where verdict_state (s) |> (
  printf (" (s = ") fby
  print (s) fby
  printf (" and \n") fby
  let (a: action, n: natural) =
    for e: edge among out (s) in (a: action, n: natural) where verdict_edge (e)
    apply (fby, +) from (nop, 0) to
    (
      if e -> [INCONCLUSIVE] then
        print_inconclusive (n, iv1, iv2, iv3)
      else (* e -> [PASS] *)
        print_pass (n)
      end_if,
      nb_verdicts (s)
    )
    end_for
  in
    use n in
      a
    end_use
  end_let fby
  printf (")\n or")
) fby

(* initial state *)
printf (" (s = 0 and") fby printf ("\n ") fby
print (iv1) fby printf (" = ") fby print (in1) fby printf (" and ") fby
print (iv2) fby printf (" = ") fby print (in2) fby printf (" and ") fby
print (iv3) fby printf (" = ") fby print (in3) fby printf (")\n or") fby

(* default case: no change *)
printf ("\n (") fby
print (iv1) fby printf (" = pre ") fby print (iv1) fby printf (" and ") fby
print (iv2) fby printf (" = pre ") fby print (iv2) fby printf (" and ") fby
print (iv3) fby printf (" = pre ") fby print (iv3) fby printf (" and ") fby
print (ov1) fby printf (" = ") fby print (already1) fby printf (" and ") fby
print (ov2) fby printf (" = ") fby print (already2) fby printf (")\n") fby
printf (" );\n") fby
```

```
(* pass verdict coverage *)
printf (" pass = false ->\n") fby
<| fby on s:state where
     exists e:edge among out (s) in
       (e -> [PASS]) and (target (e) = s)
     end_exists
|> (
  printf (" if s = ") fby
  print (s) fby
  printf (" then\n") fby
  printf (" true\n") fby
  printf (" else\n") fby
  printf (" pre pass;\n")
) fby

(* inconclusive verdict coverage *)
printf (" inconclusive = false ->\n") fby
<| fby on s:state where
     exists e:edge among out (s) in
       (e -> [INCONCLUSIVE]) and (target (e) = s)
     end_exists
|> (
  printf (" if s = ") fby
  print (s) fby
  printf (" then\n") fby
  printf (" true\n") fby
  printf (" else\n") fby
  printf (" pre inconclusive;\n")
) fby

(* all states coverage *)
<| fby on s:state |> (
  printf (" s") fby
  print (s) fby
  printf (" = false -> if s = ") fby
  print (s) fby
  printf (" then true else pre s") fby
  print (s) fby
  printf (";\n")
) fby
printf (" states_covered = false -> ") fby
<| fby on s:state |> (
  if ((mod (number (s), 6) = 0) and (number (s) > 0)) then
    printf ("\n s")
```

```
  else printf ("s")
  end_if fby
  print (s) fby
  printf (" and ")
) fby
printf ("true;\n") fby

printf ("tel\n")

end_let
end_let
end_let
end_let
```

Bibliography

[AFV01] Luca Aceto, Wan Fokkink, and Chris Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 3, pages 197–292. North-Holland, 2001.

[AQR+04] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th Conference on Computer-Aided Verification (CAV'04), Boston, MA, USA*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer, July 2004.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1988.

[BDJM05] Damien Bergamini, Nicolas Descoubes, Christophe Joubert, and Radu Mateescu. Bisimulator: A modular tool for on-the-fly equivalence checking. In Nicolas Halbwachs and Lenore Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), Edinburgh, Scotland, UK*, volume 3440 of *Lecture Notes in Computer Science*, pages 581–585. Springer, April 2005.

[BDL+01] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Oliver Möller, Paul Pettersson, and Wang Yi. Uppaal: Present and Future. In *Proceedings of the 40th IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

[Bel10] Axel Belinfante. JTorX: A Tool for On-line Model-driven Test Derivation and Execution. In Javier Esparza and Rupak Majumdar, editors, *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10), Paphos, Cyprus*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[Bel14] Axel Belinfante. *JTorX: Exploring Model-Based Testing*. PhD thesis, University of Twente, September 2014.

[Ber89] Gérard Berry. Real time programming: Special purpose or general purpose languages. In *IFIP Congress*, pages 11–17, January 1989.

[Ber07] Gérard Berry. SCADE: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer, 2007.

[BFS05] Axel Belinfante, Lars Frantzen, and Christian Schallhart. 14 Tools for Test Case Generation. In Broy et al. [BJK+05], pages 391–438.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19:87–152, 1992.

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[BGM91] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):87–405, November 1991.

[Bha14] Puneet Bhateja. A TGV-like Approach for Asynchronous Testing. In *Proceedings of the 7th India Software Engineering Conference (ISEC'14), Chennai, India*, pages 13:1–13:6. ACM, February 2014.

[Bha17] Puneet Bhateja. Asynchronous testing of real-time systems. In Mohamed Mosbah and Michaël Rusinowitch, editors, *Proceedings of the 8th International Symposium on Symbolic Computation in Software Science (SCSS 2017), Gammarth, Tunisia*, volume 45 of *EPiC Series in Computing*, pages 42–48. EasyChair, April 2017.

[BHH+06] Eckard Böde, Marc Herbstritt, Holger Hermanns, Sven Johr, Thomas Peikenkamp, Reza Pulungan, Ralf Wimmer, and Bernd Becker. Compositional Performability Evaluation for Statemate. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems (QUEST'06), Riverside, California, USA*, pages 167–178. IEEE Computer Society Press, September 2006.

[BHR84] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstractions. *Theor. Comput. Sci.*, 37:77–121, 1985.

[BK11] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), Snowbird, UT, USA*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, July 2011.

[BLL+95] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, New Brunswick, NJ, USA*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, oct 1995.

[BMMW18] Josip Bozic, Lina Marsso, Radu Mateescu, and Franz Wotawa. A formal TLS handshake model in LNT. In Holger Hermanns and Peter Höfner, editors, *Proceedings of the 3nd Workshop on Models for Formal Analysis of Real Systems (MARS'18), Thessaloniki, Greece*, volume 268, April 2018.

[Bou98] Amar Bouali. Xeve, an ESTEREL verification environment. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98), Vancouver, BC, Canada*, volume 1427 of *lncs*, pages 500–504. Springer, July 1998.

[BR01] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01), Paris, France*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, July 2001.

[BSY17] Frank P. Burns, Danil Sokolov, and Alex Yakovlev. A Structured Visual Approach to GALS Modeling and Verification of Communication Circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 36(6):938–951, June 2017.

[CCG+19] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe, and Gideon Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.8). INRIA, Grenoble, France, January 2019.

[CGK+12] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems - tool paper. In P. Madhusudan and Sanjit A. Seshia, editors, *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12), Berkeley, CA, USA*, volume 7358 of *lncs*, pages 718–724. Springer, July 2012.

[CGK+13] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. An Overview of the mCRL2 Toolset and Its Recent Advances. In Nir Piterman and Scott A. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13), Rome, Italy*, volume 7795 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2013.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[Cha84] Daniel Marcos Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. Doctoral thesis, Stanford University, Department of Computer Science, October 1984.

[CJRZ02] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Grenoble, France*, volume 2280 of *Lecture Notes in Computer Science*, pages 470–475. Springer, April 2002.

[CMST16] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The kind 2 model checker. In Swarat Chaudhuri and Azadeh Farzan, editors, *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16), Toronto, ON, Canada*, volume 9780 of *lncs*, pages 510–517. Springer, July 2016.

[CPS89] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench. In Joseph Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*, pages 24–37. Springer, June 1989.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

[Dav85] Donald W. Davies. A Message Authenticator Algorithm Suitable for a Mainframe Computer. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology – Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques (CRYPTO'84), Santa Barbara, CA, USA*, volume 196 of *Lecture Notes in Computer Science*, pages 393–400. Springer, August 1985.

[dBORZ99] Lydie du Bousquet, Farid Ouabdesselam, Jean-Luc Richier, and Nicolas Zuanon. Lutess: A Specification-Driven Testing Environment for Synchronous Software. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99) Los Angeles, CA, USA*, pages 267–276. ACM, May 1999.

[DC88] Donald W. Davies and David O. Clayden. The Message Authenticator Algorithm (MAA) and its Implementation. NPL Report DITC 109/88, National Physical Laboratory, Teddington, Middlesex, UK, February 1988.

[DMK+06] Frederic Doucet, Massimiliano Menarini, Ingolf H. Krüger, Rajesh K. Gupta, and Jean-Pierre Talpin. A Verification Approach for GALS Integration of Synchronous Components. *Electronic Notes in Theoretical Computer Science*, 146(2):105–131, 2006.

[dMOR+04] Leonardo Mendonça de Moura, Sam Owre, Harald Rueß, John M. Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04), Boston, MA, USA*, volume 3114 of *lncs*, pages 496–500. Springer, July 2004.

[DR06] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.1. *RFC*, 4346:1–87, 2006.

[DRB+09] Francisco Durán, Manuel Roldán, Emilie Balland, Mark van den Brand, Steven Eker, Karl Trygve Kalleberg, Lennart C. L. Kats, Pierre-Etienne Moreau, Ruslan Schevchenko, and Eelco Visser. The Second Rewrite Engines Competition. *Electronic Notes in Theoretical Computer Science*, 238(3):281–291, 2009.

[DRB+10] Francisco Durán, Manuel Roldán, Jean-Christophe Bach, Emilie Balland, Mark van den Brand, James R. Cordy, Steven Eker, Luc Engelen, Maartje de Jonge, Karl Trygve Kalleberg, Lennart C. L. Kats, Pierre-Etienne Moreau, and Eelco Visser. The Third Rewrite Engines Competition. In Peter Csaba Ölveczky, editor, *Proceedings of the 8th International Workshop on Rewriting Logic and Its Applications (WRLA'10), Paphos, Cyprus*, volume 6381 of *Lecture Notes in Computer Science*, pages 243–261. Springer, 2010.

[dT01] René G. de Vries and Jan Tretmans. Towards formal test purposes. In *Formal Approaches to Testing of Software (FATES'01)*, pages 61–76. BRICS Notes Series, 2001.

[Eco18] Why uber's self-driving car killed a pedestrian. *The Economist*, May 2018.

[FFJ+12] Yliès Falcone, Jean-Claude Fernandez, Thierry Jéron, Hervé Marchand, and Laurent Mounier. More testable properties. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(4):407–437, August 2012.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (CAV'96), New Brunswick, New Jersey, USA*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer, August 1996.

[FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, September 1979.

[FM91] Jean-Claude Fernandez and Laurent Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In Kim G. Larsen and A. Skou, editors, *Proceedings of the 3rd Workshop on Computer-Aided Verification (CAV'91), Aalborg, Denmark*, volume 575 of *Lecture Notes in Computer Science*, pages 181–191. Springer, July 1991.

[FW88] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Software Eng.*, 14(10):1483–1498, 1988.

[GABR14] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 – A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14), Grenoble, France*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2014.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[Gar91] Hubert Garavel. Binary coded graphs: Definition of the bcg format. Rapport SPECTRE C28, Laboratoire de Génie Informatique – Institut IMAG, Grenoble, January 1991.

[Gar98] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), Lisbon, Portugal*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer, March 1998. Full version available as INRIA Research Report RR-3352.

[Gar15] Hubert Garavel. Revisiting Sequential Composition in Process Calculi. *Journal of Logical and Algebraic Methods in Programming*, 84(6):742–762, November 2015.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *Proceedings of the 6th join International Conference Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT'95), Aarhus, Denmark*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, May 1995.

[GBHG14] Alexander Graf-Brill, Holger Hermanns, and Hubert Garavel. A Model-based Certification Framework for the EnergyBus Standard. In Erika Abraham and Catuscia Palamidessi, editors, *Proceedings of the 34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems (FORTE'15), Berlin, Germany*, volume 8461 of *Lecture Notes in Computer Science*, pages 84–99. Springer, June 2014.

[GG13] Hubert Garavel and Susanne Graf. Formal Methods for Safe and Secure Computers Systems. BSI Study 875, Bundesamt für Sicherheit in der Informationstechnik, Bonn, Germany, December 2013.

[GH17] Alexander Graf-Brill and Holger Hermanns. Model-Based Testing for Asynchronous Systems. In Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti, editors, *Critical Systems: Formal Methods and Automated Verification — Proceedings of the Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems and the 17th International Workshop on Automated Verification of Critical Systems (FMICS-AVoCS 2017), Turin, Italy*, volume 10471 of *Lecture Notes in Computer Science*, pages 66–82. Springer, September 2017.

[GL88] Renaud Guillemot and Luigi Logrippo. Derivation of useful execution trees from lotos specifications by using an interpreter. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland)*, pages 311–327. North-Holland, September 1988.

[GL01] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea*, pages 377–392. Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 15(2):89–107, April 2013.

[GLS17] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. From LOTOS to LNT. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 3–26. Springer, October 2017.

[GM17] Hubert Garavel and Lina Marsso. A Large Term Rewrite System Modelling a Pioneering Cryptographic Algorithm. In Holger Hermanns and Peter Höfner, editors, *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems (MARS'17), Uppsala, Sweden*, volume 244 of *Electronic Proceedings in Theoretical Computer Science*, pages 129–183, April 2017.

[GM18] Hubert Garavel and Lina Marsso. Comparative Study of Eight Formal Specifications of the Message Authenticator Algorithm. In Holger Hermanns and Peter Höfner, editors, *Proceedings of the 3nd Workshop on Models for Formal Analysis of Real Systems (MARS'18), Thessaloniki, Greece*, volume 268, April 2018.

[GMM12] Henning Günther, Stefan Milius, and Oliver Möller. On the formal verification of systems of synchronous software components. In *Proceedings of the 31st International Conference on Computer Safety, Reliability, and Security (SAFECOMP'12), Magdeburg, Germany*, volume 7612 of *Lecture Notes in Computer Science*, pages 291–304. Springer, September 2012.

[GMR+06] Jan Friso Groote, Aad Mathijssen, Michel A. Reniers, Yaroslav S. Usenko, and Muck van Weerdenburg. The Formal Specification Language mCRL2. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems*, volume 06351 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl, Germany, 2006.

[Gol88] Ursula Goltz. On Representing CCS Programs by Finite Petri Nets. In Michal Chytil, Ladislav Janiga, and Václav Koubek, editors, *Proceedings of the 13th Symposium on Mathematical Foundations of Computer Science (MFCS'88), Carlsbad, Czechoslovakia*, volume 324 of *Lecture Notes in Computer Science*, pages 339–350. Springer, 1988.

[GR84] Ursula Goltz and Wolfgang Reisig. CSP-Programs with Individual Tokens. In Grzegorz Rozenberg, Hartmann J. Genrich, and Gérard Roucairol, editors, *Proceedings of the 1983 and 1984 European Workshop on Applications and Theory in Petri Nets (APN'84), Toulouse, France and Aarhus, Denmark*, volume 188 of *Lecture Notes in Computer Science*, pages 169–196. Springer, 1984.

[GS17] Hubert Garavel and Wendelin Serwe. The Unheralded Value of the Multiway Rendezvous: Illustration with the Production Cell Benchmark. In Holger Hermanns and Peter Höfner, editors, *Proceedings of the 2nd Workshop on*

Models for Formal Analysis of Real Systems (MARS'17), Uppsala, Sweden, volume 244 of *Electronic Proceedings in Theoretical Computer Science*, pages 230–270, April 2017.

[GT09] Hubert Garavel and Damien Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In Corina Pasareanu, editor, *Proceedings of the 16th International SPIN Workshop on Model Checking of Software (SPIN'09), Grenoble, France*, volume 5578 of *Lecture Notes in Computer Science*, pages 241–260. Springer, June 2009.

[GTL03] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. POLYCHRONY for system design. *Journal of Circuits, Systems, and Computers*, 12(3):261–304, 2003.

[GVZ00] Hubert Garavel, César Viho, and Massimo Zendri. System design of a cc-numa multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. Research Report RR-4041, INRIA, November 2000.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98), Vancouver, BC, Canada*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer, July 1998.

[HB02] Nicolas Halbwachs and Siwar Baghdadi. Synchronous Modelling of Asynchronous Systems. In *Proceedings of the 2nd International Conference on Embedded Software (EMSOFT'02), Grenoble, France*, volume 2491, pages 240–251. Springer, October 2002.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, March 2006.

[HM06] Nicolas Halbwachs and Louis Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proceedings of the 6st International Conference on Application of Concurrency to System Design (ACSD'06), Turku, Finland*, pages 3–14. IEEE Computer Society, June 2006.

[HN04] Alan Hartman and Kenneth Nagin. The AGEDIS tools for model based testing. In George S. Avrunin and Gregg Rothermel, editors, *Proceedings of the 6st InternationalSymposium on Software Testing and Analysis (ISSTA'04), Boston, Massachusetts, USA*, pages 129–132. ACM, July 2004.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hol97] Gerard J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *Proceedings of SPIN97 the 3rd SPIN Workshop (Twente University, Enschede, The Netherlands)*, April 1997.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[HP06] Anders Hessel and Paul Pettersson. Model-Based Testing of a WAP Gateway: An Industrial Case-Study. In Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Applications and Technology, Revised Selected Papers of the 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS) and the 5th International Workshop on Parallel and Distributed Model Checking (PDMC), Bonn, Germany*, volume 4346 of *Lecture Notes in Computer Science*, pages 116–131. Springer, August 2006.

[IET18] IETF. The transport layer security (tls) protocol version 1.4 draft-ietf-tls-tls13-24, February 2018.

[ISO86] ISO. Requirements for Message Authentication (Wholesale). International Standard 8730, International Organization for Standardization – Banking, Geneva, November 1986.

[ISO87] ISO. Approved Algorithms for Message Authentication – Part 1: Data Encryption Algorithm (DEA). International Standard 8731-1, International Organization for Standardization – Banking, Geneva, May 1987.

[ISO89] ISO/IEC. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1989.

[ISO90] ISO. Requirements for Message Authentication (Wholesale). International Standard 8730, International Organization for Standardization – Banking, Geneva, May 1990.

[ISO92] ISO. Approved Algorithms for Message Authentication – Part 2: Message Authenticator Algorithm. International Standard 8731-2, International Organization for Standardization – Banking, Geneva, September 1992.

[Jeb16] Fatma Jebali. *Formal Framework for Modelling and Verifying Globally Asynchronous Locally Synchronous Systems*. PhD thesis, Grenoble Alpes University, France, September 2016.

[JHR13] Erwan Jahier, Nicolas Halbwachs, and Pascal Raymond. Engineering functional requirements of reactive systems using synchronous languages. *8th IEEE International Symposium on Industrial Embedded Systems*, 8:140–149, 2013.

[JJ05] Claude Jard and Thierry Jéron. Tgv: Theory, principles and algorithms – a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, August 2005.

[JLM16] Fatma Jebali, Frédéric Lang, and Radu Mateescu. Formal Modelling and Verification of GALS systems using GRL and CADP. *Formal Aspects of Computing*, 28(5):767–804, 2016.

[JRB06] Erwan Jahier, Pascal Raymond, and Philippe Baufreton. Case studies with Lurette V2. *International Journal on Software Tools for Technology Transfer*, 8(6):517–530, September 2006.

[JRH19] Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. Reference Manual of the Lustre Manual (Version 6.101). Lustre manual, August 2019.

[KLN+15] Jin Hyun Kim, Kim G. Larsen, Brian Nielsen, Marius Mikucionis, and Petur Olsen. Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools. In Manuel Núñez and Matthias Güdemann, editors, *Proceedings of the 20th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2015), Oslo, Norway*, volume 9128 of *Lecture Notes in Computer Science*, pages 47–61. Springer, June 2015.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[KS15] Abderahman Kriouile and Wendelin Serwe. Using a Formal Model to Improve Verification of a Cache-Coherent System-on-Chip. In Christel Baier and Cesare Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), London, United Kingdom*, volume 9035 of *Lecture Notes in Computer Science*, pages 708–722. Springer, 2015.

[KTK09] Masaya Kadono, Tatsuhiro Tsuchiya, and Tohru Kikuno. Using the nusmv model checker for test generation from statecharts. In *Proceedings of the*

15th International Symposium Dependable Computing (PRDC'09), BShanghai, China, pages 37–42. IEEE Computer Society, November 2009.

[Lai91] M. K. F. Lai. A Formal Interpretation of the MAA Standard in Z. NPL Report DITC 184/91, National Physical Laboratory, Teddington, Middlesex, UK, June 1991.

[Lak06] Abdesselam Lakehal. *Critères de couverture structurelle pour les programmes Lustre. (Strurtural coverage criteria for Lustre programs).* Thèse de Doctorat, Joseph Fourier University, Grenoble, France, 2006.

[Lam93] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, November 1993.

[LMNS05] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing Real-time Embedded Software Using UPPAAL-TRON: An Industrial Case Study. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT'05), Jersey City, NJ, USA*, pages 299–306. ACM, 2005.

[LP05] Abdesselam Lakehal and Ioannis Parissis. Lustructu: A tool for the automatic coverage assessment of lustre programs. In *Proceedings of the 16th International Symposium on Software Reliability Engineering (ISSRE'05), Chicago, IL, USA*, pages 301–310. IEEE Computer Society, 2005.

[MA00] Bruno Marre and Agnès Arnould. Test Sequences Generation from LUSTRE Descriptions: GATeL. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE), Grenoble, France*, page 229. IEEE Computer Society, September 2000.

[Mat98] Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* Thèse de Doctorat, Institut National Polytechnique de Grenoble, April 1998.

[Mat00] Radu Mateescu. Efficient diagnostic generation for boolean equation systems. In Susanne Graf and Michael Schwartzbach, editors, *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00), Berlin, Germany*, volume 1785 of *Lecture Notes in Computer Science*, pages 251–265. Springer, March 2000. Full version available as INRIA Research Report RR-3861.

[Mat05] Radu Mateescu. On-the-fly State Space Reductions for Weak Equivalences. In Tiziana Margaria and Mieke Massink, editors, *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'05), Lisbon, Portugal*, pages 80–89. ERCIM, ACM Computer Society Press, September 2005.

[Mat06] Radu Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *Springer Interna-*

tional Journal on Software Tools for Technology Transfer (STTT), 8(1):37–56, February 2006. Full version available as INRIA Research Report RR-5948, July 2006.

[MDP14] Mouna Tka Mnad, Christophe Deleuze, and Ioannis Parissis. Synchronous programs testing language (SPTL). In Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, Carmelo Maria Torre, Jorge Gustavo Rocha, Maria Irene Falcão, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi, editors, *Proceedings of the 14th International Conference on On Computational Science and Its Applications (ICCSA'14), Guimarães, Portugal*, volume 8579, pages 683–695. Springer, June 2014.

[MDP+16] Mouna Tka Mnad, Christophe Deleuze, Ioannis Parissis, Jackie Launay, and Jean Baptiste Gning. Automated test generation for synchronous controllers. In Christof J. Budnik, Gordon Fraser, and Francesca Lonetti, editors, *Proceedings of the 11th Workshop on Automation of Software Test (AST@ICSE'16), Austin, Texas, USA*, volume 28, pages 1–7. ACM, May 2016.

[MG98] Radu Mateescu and Hubert Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In Tiziana Margaria, editor, *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98), Aalborg, Denmark*, pages 33–42. BRICS, July 1998.

[MGS12] Avinash Malik, Alain Girault, and Zoran Salcic. Formal semantics, compilation and execution of the GALS programming language dsystemj. *IEEE Trans. Parallel Distrib. Syst.*, 23(7):1240–1254, 2012.

[MGT+04] Mohammad Reza Mousavi, Paul Le Guernic, Jean-Pierre Talpin, Sandeep K. Shukla, and Twan Basten. Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks. In *Proceedings of European Conference on Design, Automation and Test (DATE'04), Paris, France*, pages 384–389. IEEE Computer Society, February 2004.

[Mil83] Robin Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MLD+13] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, 30(3):50–57, 2013.

[MLN04] Marius Mikucionis, Kim Guldstrand Larsen, and Brian Nielsen. T-UPPAAL: online model-based testing of real-time systems. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE), Linz, Austria*, pages 396–397. IEEE Computer Society, September 2004.

[MMPS19] Lina Marsso, Radu Mateescu, Ioannis Parissis, and Wendelin Serwe. Asynchronous testing of synchronous components in GALS systems. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Proceedings of the 15th International Conference on Integrated Formal Methods (IFM'19), Bergen, Norway*, pages 360–378. Springer, December 2019.

[MMS18] Lina Marsso, Radu Mateescu, and Wendelin Serwe. TESTOR: A modular tool for on-the-fly conformance test case generation. In Dirk Beyer and Marieke Huisman, editors, *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18), Thessaloniki, Greece*, pages 211–228. Springer, April 2018.

[MO08] Radu Mateescu and Emilie Oudot. Bisimulator 2.0: An On-the-Fly Equivalence Checker based on Boolean Equation Systems. In *Proceedings of the 6th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'08), Anaheim, CA, USA*, pages 73–74. IEEE Computer Society Press, June 2008.

[MPS+09] Wojciech Mostowski, Erik Poll, Julien Schmaltz, Jan Tretmans, and Ronny Wichers Schreur. Model-based testing of electronic passports. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'09), Eindhoven, The Netherlands*, volume 5825 of *lncs*, pages 207–209. Springer, November 2009.

[MT08] Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland*, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164. Springer, May 2008.

[Mun91a] Harold B. Munster. Comments on the LOTOS Standard. NPL Technical Memorandum DITC 52/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[Mun91b] Harold B. Munster. LOTOS Specification of the MAA Standard, with an Evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991. Available at `ftp://ftp.inrialpes.fr/pub/vasy/publications/others/Munster-91-a.pdf`.

[MvOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. Available from `http://cacr.uwaterloo.ca/hac`.

[PCdVA12] Jose Proenca, Dave Clarke, Erik de Vink, and Farhad Arbab. Dreams: A Framework for Distributed Synchronous Coordination. In *Proceedings of the*

27th Symposium On Applied Computing (SAC'12), Riva del Garda, Italy. ACM, 2012.

[PEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel.* Springer, 2007.

[PO90] Graeme I. Parkin and G. O'Neill. Specification of the MAA Standard in VDM. NPL Report DITC 160/90, National Physical Laboratory, Teddington, Middlesex, UK, February 1990.

[PO91] Graeme I. Parkin and G. O'Neill. Specification of the MAA Standard in VDM. In Søren Prehn and W. J. Toetenel, editors, *Formal Software Development – Proceedings (Volume 1) of the 4th International Symposium of VDM Europe (VDM'91), Noordwijkerhout, The Netherlands*, volume 551 of *Lecture Notes in Computer Science*, pages 526–544. Springer, October 1991.

[Pre11] Bart Preneel. MAA. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security (2nd Edition)*, pages 741–742. Springer, 2011.

[PRvO97] Bart Preneel, Vincent Rumen, and Paul C. van Oorschot. Security Analysis of the Message Authenticator Algorithm (MAA). *European Transactions on Telecommunications*, 8(5):455–470, 1997.

[PvO96] Bart Preneel and Paul C. van Oorschot. On the Security of Two MAC Algorithms. In Ueli M. Maurer, editor, *Advances in Cryptology – Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT'96), Saragossa, Spain*, volume 1070 of *Lecture Notes in Computer Science*, pages 19–32. Springer, May 1996.

[PvO99] Bart Preneel and Paul C. van Oorschot. On the Security of Iterated Message Authentication Codes. *IEEE Transactions on Information Theory*, 45(1):188–199, 1999.

[RPD96] Vincent Rijmen, Bart Preneel, and Erik De Win. Key Recovery and Collision Clusters for MAA. In *Proceedings of the 1st International Conference on Security in Communication Networks (SCN'96)*, 1996.

[RRJ08] Pascal Raymond, Yvan Roux, and Erwan Jahier. Lutin: a language for specifying and executing reactive scenarios. *EURASIP Journal on Embedded Systems*, 2008, 2008.

[RSD+04] S. Ramesh, Sampada Sonalkar, Vijay D'Silva, , Naveen Chandra, and B. Vijayalakshmi. A Toolset for Modelling and Verification of GALS Systems. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04), Boston, USA*, volume 3114 of *Lecture Notes in Computer Science*, pages 506–509. Springer, July 2004.

[Ser15] Wendelin Serwe. Formal Specification and Verification of Fully Asynchronous Implementations of the Data Encryption Standard. In Rob van Glabbeek, Jan Friso Groote, and Peter Höfner, editors, *Proceedings of the International Workshop on Models for Formal Analysis of Real Systems (MARS'15), Suva, Fiji*, volume 196 of *Electronic Proceedings in Theoretical Computer Science*, 2015.

[SP07] Besnik Seljimi and Ioannis Parissis. Automatic generation of test data generators for synchronous programs: Lutess V2. In *Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation (DOSTA'07) Dubrovnik, Croatia*, pages 8–12. ACM, September 2007.

[SSC+04] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, and Florence Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. In Giorgio C. Buttazzo, editor, *Proceedings of the fourth International Conference on On Embedded Software (EMSOFT'04), Pisa, Italy*, pages 259–268. ACM, 2004.

[Str82] Robert S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54:121–141, 1982.

[TB03] Jan Tretmans and Hendrik Brinksma. TorX: Automated Model-Based Testing. In A. Hartman and K. Dussa-Ziegler, editors, *Proceedings of the First European Conference on Model-Driven Software Engineering, Zurich, Switzerland*, pages 32–43, December 2003.

[TLK92] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Software Eng.*, 18(3):206–215, 1992.

[Tre92] Jan Tretmans. *A Formal Approach to Conformance Testing*. Twente University Press, 1992.

[Tre96] Jan Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer networks and ISDN systems*, 29(1):49–79, December 1996.

[Tre99] Jan Tretmans. Testing Concurrent Systems: A Formal Approach. In Jos C. M. Baeten and Sjouke Mauw, editors, *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer, August 1999.

[Tre08] Jan Tretmans. Model Based Testing with Labelled Transition Systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[Tre17] Jan Tretmans. On the existence of practical testers. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2017.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, August 2012.

[Utt05] Mark Utting. The role of model-based testing. In B. Meyer and J. Woodcock, editors, *Proceedings of the First European Conference on Verified Software: Theories, Tools, Experiments (VSTTE'05), Möhrendorf, Germany*, volume 4171, pages 510–517. Springer, October 2005.

[Vas04] Vasy. Reductor manual page. `https://cadp.inria.fr/man/reductor.html`, 2004.

[Vas10] Vasy. Bcg_cmp manual page. `https://cadp.inria.fr/man/bcg_cmp.html`, 2010.

[vGW89] R. J. van Glabbeek and W. Peter Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In Yves Ledru, editor, *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00), Grenoble, France*, pages 3–12, September 2000.

[Wea06] Alfred C. Weaver. Secure sockets layer. *IEEE Computer*, 39(4):88–90, 2006.

[WTK00] Tim Willemse, Jan Tretmans, and Arjen Klomp. A case study in formal methods: Specification and validation of the om/rr protocol. In Stefania Gnesi, Ina Schieferdecker, and Axel Rennoch, editors, *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'2000), Berlin, Germany*, GMD Report 91, pages 331–344, Berlin, April 2000.

[WW09] Martin Weiglhofer and Franz Wotawa. Asynchronous Input-Output Conformance Testing. In Sheikh Iqbal Ahamed, Elisa Bertino, Carl K. Chang, Vladimir Getov, Lin Liu, Hua Ming, and Rajesh Subramanyan, editors, *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009), Seattle, Washington, USA*, pages 154–159. IEEE Computer Society, July 2009.