Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 25 Mai 2016

Présentée par

Fatma Jebali

Thèse dirigée par **Frédéric Lang**
et codirigée par **Radu Mateescu**

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

Formal Framework for Modelling and Verifying Globally Asynchronous Locally Synchronous Systems

Thèse soutenue publiquement le **12/09/2016**,
devant le jury composé de :

Monsieur Nicolas Halbwachs
Vérimag, Président
Monsieur Alessandro Fantechi
Université de Florence, Rapporteur
Madame Virginie Wiels
ONERA, Rapporteur
Monsieur Jean-Pierre Talpin
Inria Rennes, Examinateur
Monsieur Éric Jenn
IRT Saint-Éxupéry, Examinateur
Monsieur Frédéric Lang
Inria Grenoble, Directeur de thèse
Monsieur Radu Mateescu
Inria Grenoble, Directeur de thèse

Acknowledgements

I am deeply grateful to my first supervisor, Frédéric Lang, for his invaluable help and guidance during my PhD, shaping the way I do research while giving me freedom to pursue my own ideas, and for being a constant source of self-transcendence. I am also heartily thankful to my second supervisor, Radu Mateescu, for his kindness and his wise and enthusiastic counsel through the thought-provoking discussions we had. From Frédéric and Radu, I have gained an incredible amount of life and research knowledge; for that, I cannot thank them enough.

I am also grateful to my examiners, Nicolas Halbwachs, Alessandro Fantechi, Virginie Wiels, Jean-Pierre Talpin, and Eric Jenn for their thorough reading of my thesis and their constructive comments. Special thank goes to Eric for providing me with the AutoFlight Control System as a case study and for insightful encouragement from an industry perspective.

Many thanks are due to the rest of the Convecs team, Hubert Garavel, Wendeline Serwe, and Gwen Salaün, for providing such a stimulating and cheerful environment of research. I owe much to Hubert for his enthusiasm, wisdom, and support providing me with much-needed advice along the way. I warmly thank Jingyan Jourdan-Lu and Eric Léo, my office mates, for being such pleasant company and for making the Bluesky project a wonderful experience. Thanks also to my fellows: Raquel Oliveira for graciously hosting me in her office during many Sundays; Gianluca Barbon and Lina Marsso, who appeared at critical times, for their soft presence and support; Abderahman Kriouile for inspiring me to apply my work on avionics systems; as well as Rim Abid, Lakhdar Akroun, Hugues Evrard, Lina Ye, José Ignacio Requeno, Kaoutar Hafdi, Imad-Seddick Arrada, Ajay Muroor-Nadumane, Mohammad-Ali Tabikh, Sai Srikar Kasi, and Zhen Zhang. I am grateful to Myriam Etienne, the Convecs team assistant, for her kindness and her ability to find solutions in tedious administrative situations.

I also thank the people with whom I collaborated through the Bluesly project: Ioannis Parissis, Chriptophe Deleuze, and Mouna Tka from the LCIS lab; Jean-Baptiste Gnuing, Guillaume Marie, Jackie Launay, and Vincent List from Crouzet Automatismes (now InnoVista Sensors).

I met many nice friends during my PhD. Particular thanks are due to Ferdaouss, my first Grenoble's friend; Hassan and Fatma, for being a second family to me; Alia, for being a source of happiness; Kaoutar, for her contagious serenity; as well as Nashwa, Imen, Wided, Jamel, Sonia, and Rihab.

My family gave me endless love, patience, and support in every step of my life. I am eternally grateful to my mum, from whom I learned finding my strength in difficult times; my dad, from whom I learned seeking perfection in everything I do; as well as Mohamed and Rihab, for all the moments of fun we had together.

To my parents

Abstract

A GALS (*Globally Asynchronous, Locally Synchronous*) system consists of several synchronous components that evolve concurrently, each with its own pace, and communicate altogether asynchronously. This thesis proposes a formal modelling and verification framework dedicated to GALS systems, with a focus on the asynchronous behaviour.

As a cornerstone of our framework, we have designed a formal language, named GRL (*GALS Representation Language*). GRL enables the behavioural specification of synchronous components, asynchronous communication, and constraints involving both component paces and the data carried by component inputs. To analyse GRL specifications, we took advantage of the CADP software toolbox for the verification of asynchronous concurrent processes, using state space exploration techniques. For this purpose, we have defined a translation from GRL to the LNT specification language supported by CADP. The translation has been implemented by a tool named GRL2LNT, thus enabling state spaces to be automatically derived from GRL specifications.

To enable the formal verification of GRL specifications, we have designed a property specification language, named muGRL, which is interpreted on GRL state spaces. The muGRL language is based on a set of patterns capturing properties of concurrent and GALS systems, which reduces the complexity of using full-fledged temporal logics. The semantics of muGRL are defined by a translation into the MCL temporal logic supported by CADP. Finally, we have illustrated how GRL, muGRL, and CADP can be applied to model and verify concrete GALS applications, including industrial case-studies.

Résumé

Un système GALS (*Globalement Asynchrone, Localement Synchrone*) est un ensemble de composants synchrones qui évoluent en même temps, chacun à son propre rythme, et qui communiquent de manière asynchrone. Cette thèse propose un environnement formel de modélisation et de vérification dédié aux systèmes GALS, en se focalisant sur le comportement asynchrone.

Notre environnement s'appuie sur un langage formel que nous avons conçu, appelé GRL (*GALS Representation Language*). GRL permet la spécification comportementale des composants synchrones, de la communication asynchrone, et des contraintes sur les rythmes des composants ainsi que sur les valeurs que prennent les entrées des composants. Pour analyser les spécifications GRL, nous utilisons CADP, une boîte à outils logicielle permettant la vérification de processus concurrents asynchrones par des techniques d'exploration d'espaces d'états. Dans ce but, nous avons défini une traduction de GRL vers LNT, un langage de spécification supporté par CADP. La traduction est implémentée dans un outil appelé GRL2LNT, permettant ainsi la génération automatique d'espaces d'états à partir des spécifications GRL.

Pour permettre la vérification formelle des spécifications GRL, nous avons conçu un langage de propriétés, appelé muGRL, qui s'interprète sur les espaces d'états de GRL. Le langage muGRL est basé sur un ensemble de patrons qui capturent les propriétés des systèmes concurrents et des systèmes GALS, réduisant ainsi la complexité d'utiliser les logiques temporelles classiques. La sémantique de muGRL est définie par traduction vers MCL, le langage de logique temporelle fourni par CADP. Enfin, nous illustrons l'usage de GRL, muGRL et CADP pour modéliser et vérifier des applications GALS concrètes, comprenant des études de cas industrielles.

Contents

Chapter 1

Introduction

Constructing correct software and hardware systems is challenging. System quality relies not only on good performance such as processing capacity, but also on the absence of errors. For hardware systems, defects may have severe economic consequences. For software used in safety-critical systems, a simple bug can have disastrous human consequences. *Concurrent systems*, which are composed of several (hardware or software) components possibly interacting with each other, are particularly vulnerable to errors. The number of possible concurrency errors (that is, errors due to wrong ordering of concurrent events) is exponential in the number of the concurrent components. Hence, a major goal when constructing concurrent systems is their correctness despite their complexity.

Formal methods provide languages, techniques, and tools to establish system correctness. The mathematical rigour of formal methods favours an early integration of verification in the design process. For example, they have been applied in the certification of avionics software systems [DO-11, MLD+13] and railway systems [FFG14].

Model-based verification builds on models describing the system *behaviour*, i.e., what the system may do during its execution, by means of events, in an abstract and precise way. In practice, models are usually derived from high-level formalisms endowed with precise semantics. Correctness properties, also written in high-level formalisms, can be checked over models. The efficiency of the verification task relies on the adequacy of the high-level formalisms with regards to the subtleties of intended systems.

Context

According to the nature of component composition and communication, concurrent systems can be classified into *synchronous* and *asynchronous*; for each class, well-adapted formalisms are tailored to capture system behaviour.

Synchronous concurrent systems are composed of several components running in lockstep

fashion and sharing a global clock. For these systems, synchronous languages, among which *Esterel* [BG92], *Lustre* [HCRP91], and *Signal* [LGGLBLM91], are appropriate modelling formalisms. They rely on the *synchrony assumptions*: a system is seen as a deterministic and infinite loop, whose iterations represent the clock ticks; within each loop iteration, computations and data-flow communication are assumed to occur in zero-delay. The synchrony assumptions make the modelling and verification tasks easy.

Asynchronous concurrent systems are composed of several components running independently without a global clock and interacting with each other. For these systems, process algebras, among which *CCS* [Mil89], *CSP* [Hoa85], and *LOTOS* [BB87], are appropriate modelling formalisms. They are equipped with built-in operators for asynchronous parallel composition; they provide abstraction means (e.g., nondeterminism); and they have equivalence relations to efficiently and precisely compare systems.

Correctness properties of concurrent systems include the absence of undesirable situations and the succession of events in time, which can be arbitrarily far from each other. To express properties, *temporal logics*, among which *LTL* [Pnu77] and *CTL* [EC82], are powerful means. They consist of a small set of temporal operators expressing the logical precedence of events over time.

This thesis is about formally modelling and verifying GALS (*Globally Asynchronous, Locally Synchronous*) systems [Cha84], which are a class of concurrent systems. A GALS system is composed of synchronous components running in asynchronous concurrency without sharing their clocks. Communication between components is also asynchronous, i.e., message exchange may take an arbitrary amount of time. For example, in a *flight control system*, individual components are designed to run synchronously, but the distributed nature of the global system introduces asynchrony. Other GALS instances include *networks-on-chip* and distributed PLCs (*Programmable Logic Controllers*).

In the general case, a GALS system may have arbitrary complexity. No assumption can be made on clock synchronisation and component periods, nor on asynchronous communication media and their latency. Each GALS instance induces its own assumptions. In particular, although synchronous components are generally *deterministic*, the absence of a shared clock may introduce *nondeterminism*. Another source of nondeterminism is unreliable communication media along which messages can be delayed, lost, duplicated, or reordered. This makes system evolution unpredictable and unreproducible, entailing a need for formal verification.

Motivation

The correctness of GALS systems relies on combining the verification approach for synchronous systems and the one for asynchronous systems. Each approach, applied individually, is unable to capture the behavioural subtleties for which the other approach is devised. Languages and tools for synchronous systems are deterministic by nature, thus

unadapted to analyse nondeterminism and asynchronous concurrency. Languages and tools for asynchronous systems lack built-in constructs dedicated to address the pure synchrony assumptions.

We have identified a relative lack of approaches dealing with asynchronous concurrency in existing design processes of GALS systems, compared to the intensive use of approaches dealing with synchrony. This lack is manifold. On the one hand, the GALS paradigm takes its roots in the industries that already integrated synchronous languages and corresponding tools in their development process. Consequently, the focus has been shifted towards pushing the limits of synchronous languages and tools to accommodate GALS behaviours. On the other hand, synchrony is easier to master than asynchrony, owing to the zero-delay assumption and determinism, which makes systems easy to design and debug. Contrarily, asynchronous concurrent languages and temporal logics require a substantial learning effort, which may discourage potential users. Last, the behaviour of GALS systems involves asynchronous concurrency and data handling, which are two major causes of combinatorial explosion (in the possible behaviours). Additional effort should be put to make careful modelling decisions and to choose adequate algorithmic approaches, e.g., compositional verification, to face combinatorial explosion.

To alleviate the use of verification tools for asynchronous systems, one needs to introduce DSLs (*Domain Specific Languages*) [vDKV00]. GALS-specific languages serve as intermediate format mapping GALS systems, whose synchronous components are possibly modelled using synchronous languages, to verification tools for asynchronous systems. Due to the different semantics and abstraction level, a direct connection from (synchronous) design languages to asynchronous languages could be complex. Instead, performing the translation in several steps reduces that complexity and enhances the connection modularity.

As regards behavioural modelling, a DSL should provide a clear distinction between synchronous components and the asynchronous ones defining their asynchronous composition and communication. Such a distinction enables to combine of verification tools for synchronous systems and those for asynchronous systems to address separately the DSL synchronous and asynchronous components. For synchronous components, possibly obtained from translation of existing synchronous languages, the DSL can be a (minimal) language used as target of back-end compilers for synchronous languages. To ensure the practical usability of the DSL, it should enable a natural description of relevant aspects of GALS behaviour, in a way close to the end-user intuition and expectation.

As regards correctness properties, their formulation in temporal logic can be difficult and error-prone, even for users familiar with formal methods and verification. One needs a formalism tailored to capture GALS behaviour and enabling a *concise* and *natural* expression of properties.

Contributions

This thesis proposes a formal framework to analyse GALS systems focusing on their asynchronous behaviour. In this respect, we take advantage of the CADP software toolbox [GLMS13] for the verification of asynchronous concurrent processes, using state space exploration techniques.

As a cornerstone of our framework, we have designed a formal language, named GRL (*GALS Representation Language*) [JLM14a]. GRL aims at offering a concise and modular description for the behaviour of GALS systems. Both traits of synchronous programming (determinism, atomicity) and process algebra (nondeterminism, asynchronous concurrency) are combined in one unified language, while keeping homogeneous syntax and semantics. GRL builds upon the following three core constructs:

Blocks denote the synchronous part of GRL, in which the synchrony assumptions are built-in. They provide a number of basic constructs to which synchronous language constructs can be translated.

Mediums denote asynchronous components describing communication media. They are provided with enough expressiveness to model general asynchronous communication, with different buffering mechanisms, including unreliable ones.

Environments denote asynchronous components abstracting the external environment of blocks. Two kinds of constraints with different abstraction levels are considered. *Data constraints* enable to express complex properties on the data carried by block inputs. *Activation constraints* enable to control the execution of blocks, such as relations between block paces, priorities, or failure. Furthermore, it is possible to combine both kinds of constraints for enhanced usage, such as complex test case scenarios.

All-in-one, GRL is intended to be sufficiently expressive and concise to model complex GALS systems, which is an originality compared to state-of-the-art approaches.

We formalise the semantics of GRL, using *structural operational semantics* (SOS) rules, in terms of lower-level models, i.e., state spaces. This enables rigorous specification of GRL programs and paves the way for formal analysis. State spaces underlying GRL are concise, exploiting the GALS assumptions such as the atomicity of synchronous components. This enhances the efficiency of verification. Data and activation constraints would also contribute to face combinatorial explosion.

After formally defining the syntax and semantics of GRL, we address its compilation into state spaces. For this purpose, we design a translation from GRL into LNT [CCG+16], the most recent specification language supported by CADP. LNT is a general-purpose language implementing concurrency theory results and equipped with state space generators. We formalise the translation function from GRL into LNT, which is fully implemented in a tool named GRL2LNT[1].

[1]The GRL2LNT tool has been implemented, mainly not by the author, in the framework of an

To analyse GRL specifications, we exploit the MCL language, a full-fledged temporal logic supported by CADP. To leverage the expressiveness of MCL while reducing its complexity of usage, we design a property description language, named muGRL. The muGRL language builds upon a *pattern system*, following the general-purpose approach [DAC99], which is also an originality of our approach. Patterns are high-level templates that capture frequently encountered situations in GALS applications, such as component halting and idleness, and are translatable into temporal logics. The interpretation models of muGRL are the state spaces generated by translating GRL specifications into LNT. The muGRL semantics are defined by a translation into MCL. As such, muGRL is intended to disseminate temporal logic power to potential GALS designers.

Last, we experiment our approach on concrete GALS applications, issued from academia and industry. This reinforces our conviction that our approach can address a large spectrum of GALS systems, ranging from deterministic applications to ones involving arbitrary nondeterminism.

Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 surveys the state-of-the-art concerning the formal modelling and verification of concurrent and GALS systems. Chapter 3 constitutes a tutorial for the GRL language. It presents the formal syntax of each GRL construct, its intuitive semantics, along with some illustrations. Chapter 4 presents the formal dynamic semantics of GRL. It details the structural operational semantic rules for the language constructs, stressing on the behavioural ones. Chapter 5 presents the formal translation functions from GRL into LNT. Examples are given for most of the functions, to enhance the readability and ease the comprehension. Chapter 6 constitutes a tutorial for the muGRL language. Chapter 7 shows the way GRL and muGRL can be applicable to concrete GALS applications. It also briefly reports a primary industrial use of GRL. Chapter 8 concludes and offers some thoughts on extensions to this work.

Parts of this manuscript have been published in conference proceedings and journals. The article [JLM14a] presents an overview of an earlier version of GRL, the complete and formal definition of GRL being available in an 82-pages research report [JLM14b]. Since then, we have revised and enhanced the syntax of the language. The article [JLM16] presents the latest version of GRL, including the material of chapters 3 and 4 as well as an informal presentation of chapter 5.

industrial project, named *Bluesky*, of the Minalogic French competitiveness cluster (www.minalogic.com/fr/projet/bluesky). The project addresses the design and validation of networks of PLCs (*Programmable Logic Controllers*).

Notations

We introduce some mathematical concepts and conventions used in this thesis.

General notations

A set is an ordered collection of objects, called its elements. The following operators over sets are used:

Symbol	Meaning
$a_1, \ldots, a_n$	possibly empty finite sequence of elements of length n
$a_0, \ldots, a_n$	non-empty finite sequence of elements of length n+1 (ϵ if empty)
$\{a_1, \ldots, a_n\}$	possibly empty set of elements $a_1, \ldots, a_n$ of size n
$\{a_0, \ldots, a_n\}$	non-empty set of elements $a_0, \ldots, a_n$ of size n+1 ({} if empty)
$\langle a_1, \ldots, a_n \rangle$	possibly empty list of elements $a_1, \ldots, a_n$ of size n (ϵ if empty)
$\langle a_0, \ldots, a_n \rangle$	non-empty list of elements $a_0, \ldots, a_n$ of size n+1
$a \in A$	a is an element of the set A
$A \subseteq B$	A is a subset of the set B
$\{a \in A \mid P(a)\}$	the set which contains only elements of A satisfying property P
$A \times B$	the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$ (Cartesian product)
$1..n$	interval whose elements range from 1 to n

We use the operator ++ for list concatenation. For a set of lists $\{\mathcal{L}_0, ..., \mathcal{L}_n\}$, we write $\mathop{\texttt{++}}_{i \in 0..n} \mathcal{L}_i$ as abbreviation for $\mathcal{L}_0 \texttt{++} \ldots \texttt{++} \mathcal{L}_n$.

The following logical operators are used:

Symbol	Meaning
$\neg$	negation
$\wedge$	conjunction
$\vee$	disjunction
$\Rightarrow$	implication

For a set of elements $\{a_0, \ldots, a_n\}$, we write $\bigwedge_{i \in 0..n} a_i$ and $\bigvee_{i \in 0..n} a_i$ as abbreviation for $a_0 \wedge \ldots \wedge a_n$ and $a_0 \vee \ldots \vee a_n$, respectively.

Syntactic description

This document defines and references several languages. Grammars of languages are context-free. Syntactic definitions are presented in *Extended Backus-Naur Form* [Sta96], i.e., as a set of so-called *productions*. Each production has the form "$\chi ::= \xi$", where χ is a non-terminal symbol defined by the meta-expression ξ, which consists of non-terminal symbols and terminal symbols composed using the following meta-operators:

Notation	Operation	Description
(ξ_0)	bracketing	ξ_0
$[\xi_0]$	option	ξ_0 or nothing
ξ_0*	possibly empty repetition	ξ_0, zero, one, or several occurrences
ξ_0+	non-empty repetition	ξ_0, one, or several occurrences
$\xi_1\ \xi_2$	concatenation	ξ_1 followed by ξ_2
$\xi_1 \mid \xi_2$	alternative	ξ_1 or ξ_2

Additionnaly, the following conventions are used:

- Non-terminal symbols and generic terminal symbols are written in italics and their occurrences can be distinguished using subscripts.
- The terminal symbol are either keywords written in bold font or key symbols are written in teletype font. For example, "`[]`", "`()`", and "`|`" denote terminal symbols distinct from the meta-operators "[]", "()", and "|".

Chapter 2

Background and State of the Art

A GALS system combines characteristics of synchronous and asynchronous systems, which both belong to the class of *reactive systems* [HP85, Ber89, Hal10]. These are systems in permanent interaction with the outside world. In this chapter, we first introduce reactive systems. We then present the synchronous and asynchronous approach to formally model and verify reactive systems. We focus in particular on the use of the CADP toolbox for verifying asynchronous concurrent systems. Finally, we present existing approaches to the formal analysis of GALS systems.

2.1 Reactive systems

Hardware and software programs and systems interact with their *environment*, that is, the outside world within which they evolve. They can receive inputs from their environment and produce the appropriate outputs, which have effect on their environment. A program is said *transformational* if it receives inputs and terminates after producing outputs. Usually, the same inputs induce the same outputs, in which case the program is said *deterministic*. A transformational program can be described as a mathematical function that transforms inputs into outputs. Examples are compilers and optimisation algorithms.

Not all systems and programs intend to yield a final result. A system might well aim to maintain some interaction with its environment. The environment continuously prompts the system by providing it with inputs and the system reacts by producing outputs. Examples are operating systems, communication protocols, and database management systems. Such systems are called *reactive*, and a computation of outputs from inputs is called a *reaction*. Reactive systems may be subject to strict timing constraints, in which case they are referred to as *real-time systems*. In a railroad-crossing control system, it is crucial to block vehicle crossing as soon as a train approach is detected.

Concurrency is inherent in reactive systems. First, a reactive system together with its

environment form a concurrent system. Second, reactive systems are often decomposed into several *concurrent components* or tasks that operate simultaneously. Concurrent components are usually reactive themselves; they interact with their environment and potentially with each other.

Nondeterminism is usually introduced by concurrency. Two different copies of the same concurrent system are likely to operate differently, while given exactly the same inputs. An example is when several components compete to acquire a resource and the resource operates depending on which component has won the race.

Implementations of concurrent components are multiples. They may run *sequentially*, a component finishing before the next starts, or *in parallel*, all components evolving at the same time. Parallel components may run over a multi-core processor or a multi-processor machine, to speed up computations. They may also run over spatially distant machines exchanging data through a network, in which case components are called *distributed*.

Reactive systems cannot be described as mathematical functions taking inputs and producing outputs, since they run continuously without necessarily terminating, they Rather, they are described in terms of a set of infinite sequences of states and transitions (or *actions*) between states. Such infinite sequences are usually called *executions*. We define the *behaviour* of a system as the set of its possible executions. The behaviour of a concurrent system is specified in terms of the behaviours of its components. Hence, it is essential to understand the way states corresponding to component behaviours can be combined and the consequences of such combinations.

To establish the correctness of reactive systems, one needs appropriate behavioural models on which formal verification can be performed using dedicated algorithms. An appropriate behavioural model should provide an abstract and modular description of both a reactive system, its environment, and its concurrent components. The model should provide a description of the interaction between the system and its environment as well as between concurrent components. Last but not least, the model should provide a suitable abstraction of time. In this thesis, we consider a *discrete* representation. Time is an infinite series of *discrete* instants, which can (or not) be equally separated. When discrete instants are not equally separated, we are considering *logical* time.

2.1.1 Formal models for reactive systems

This section surveys some existing formal models for reactive systems. In particular, we focus on *transition systems*, in which a system is modelled in terms of states and actions. Existing models differ in the way they abstract a system behaviour, each emphasising certain aspects disregarding the others. We classify models according to the following three dichotomies, which we believe adequate to the comprehension of this thesis.

Linear-time versus branching-time [Lam80] In linear-time models, the system behaviour is expressed as the set of its possible executions, i.e., linear sequences of states and actions. This model is well suited to deterministic systems since at each moment in time, the system has a unique future. In branching-time models, the system behaviour is expressed as *computation trees* that structure the possible executions. This model is adequate to capture nondeterministic behaviours, since at each moment in time, the system can have several futures. The two models differ in the way they deal with nondeterminism. As an illustration, consider two coffee machines [Hoa85]. The first machine, once a coin is inserted, gives the user a choice between coffee and tea, and serves the user's choice. The second machine, once a coin is inserted, makes internally a nondeterministic choice, and serves either coffee or tea. Both machines have the same set of possible executions {coin, coffee} or {coin, tea}. The branching-time view distinguishes the difference between the two machines while the linear-time view does not.

Action-based versus state-based [DNV90] In the *action-based* setting, the contents of states is abstracted away. The evolution of the system behaviour is encoded in actions. Actions correspond to the interaction of the system with its environment, i.e., inputs received and outputs sent by the system, as well as internal transitions performed by the system. Examples of action-based representations include *labelled transition systems* [Par81], Petri-nets [Pet62], and I/O automata [LT89]. The *state-based* setting is the dual of the action-based one, from a theoretical point of view. The evolution of the system behaviour is encoded inside states, by means of variables and other information stored in memory. Only the internal contents of states can be observed. Examples of state-based representations include *Kripke structures* [Kri63]. In practice, the action-based representation can be seen as a "black box" view of a system and state-based models as "white box" one.

Synchronous versus asynchronous concurrency Concurrent components can be composed either in a *synchronous* or in an *asynchronous* way. Synchronous concurrent components evolve in a lockstep fashion, cadenced by a single central clock. Each clock pulse prompts all concurrent components to react. The conjunction of component actions at the same clock pulse constitutes an action of the whole system. This concurrency model is mainly supported by synchronous languages, such as *Esterel* [BG92], *Lustre* [HCRP91], and *Signal* [LGGLBLM91].

Asynchronous concurrent components evolve independently without clock sharing. A first model for asynchronous concurrency is the so-called *interleaving semantics* [Mil89]. In this model, concurrency between components is reduced to a nondeterminism choice between the possible sequences of the component actions. This model of concurrency is mainly supported by process algebras, such as CCS [Mil89], CSP [Hoa85], ACP [BK85], and LOTOS [BB87].

Another model for asynchronous concurrency is the so-called *true-concurrency* [Mon92], also called *non-interleaving model*. In this model, concurrency is a primitive notion clearly distinguishable from sequential nondeterminism. The system behaviour is represented in terms of the causal relations among actions performed by components; two actions are concurrent if they are not causally related. This model of concurrency is mainly supported by Petri-nets and Kahn-nets [Kah74].

An illustration of the three models of concurrency (in an action-based setting) is given in Figure 2.1. Actions A and B are concurrent. The synchronous composition of actions A and B results in one action labelled AB. The interleaving semantics expresses that either action A occurs followed by action B or action B occurs followed by action A. The true-concurrency model can be understood as a system with two initial states, each with an outgoing transition, since actions A and B are not causally related.

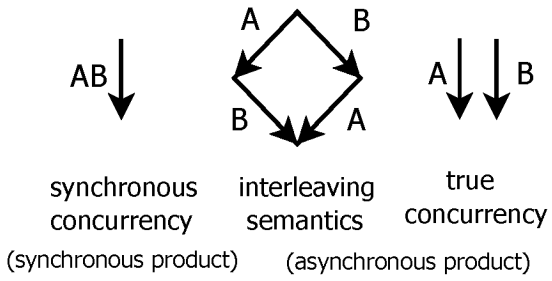


Figure 2.1: Models of concurrency in an action-based setting

2.1.2 Formal verification of reactive systems

To perform formal verification, one needs, in addition to a *model* describing all potential behaviours of the system, to describe the set of *properties* that must hold on the system. The verification problem consists in proving (automatically) that the model satisfies the properties. We briefly introduce the common formal verification approaches:

- *Static analysis* consists in verifying programs, without executing them, relying on the semantics of the language in which programs are written. The abstract interpretation technique [CC77] consists in abstracting the model to a smaller one, in such a way that if the desired property holds on the abstracted model, it must hold on the original one. Among the static analysers based on abstract interpretation, we cite Astrée [CCF+05] and Verasco [JLB+15]. Abstract interpretation is used extensively in transformational systems such as in compiler optimisation and sequential program verification. While being an automated technique, abstract interpretation cannot achieve 100% precision in the general case.
- *Theorem proving* consists in modelling the system as a set of mathematical definitions. The desired properties of the system are derived as theorems that stem

from those definitions. Proofs can be constructed either by hand or by using automatic theorem provers and interactive proof checkers. Although it cannot be fully automated, theorem proving is assisted by powerful proof assistant, among which Coq [FHB+97] and Isabelle [Pau89]. Theorem proving techniques are particularly useful in the case of general infinite-state reactive systems, e.g., systems containing unbounded data structures. They have been used in the context of reactive and real-time systems, but less for distributed concurrent systems.

- *Model checking* consists in modelling the system as a (finite) transition system that describes all the possible executions of the system. The desired properties of the system can be described by reasoning about the temporal ordering of events[1]. *Temporal logic* formalisms have been introduced in the late seventies [Pnu77] for this purpose. A temporal logic is a set of operators, expressing the logical precedence either between states (state-based temporal logic) or between actions (action-based temporal logic). Additionally, temporal logics can be interpreted either on linear-time models, by specifying properties of individual execution sequences, or on branching-time models, by taking into account the branching structure of the state space. According to these two dichotomies, a lot of formalisms have been proposed. The following table shows the most representative ones.

	linear-time	branching-time	linear- and branching-time
state-based	LTL [Pnu77]	CTL [EC82]	CTL* [EH86]
action-based	ALTL [GM03]	ACTL [DFGR93]	ACTL* [DNV90]

  Given a state space and a property, the *model checking* problem [CGP00] consists in determining whether the state space satisfies the property or not. If the property does not hold on the state space, it is desirable to obtain a diagnostic (or counterexample) showing an undesirable behaviour present in the state space. This problem is solved by model checking algorithms, which traverse the state space and halt as soon as the truth value of the property has been determined. State space traversing techniques are typically grouped in two classes:

  – *enumerative* techniques consider each state of the system separately
  – *symbolic* techniques manipulate sets of states represented using either decision diagrams or logical formulas. The satisfiability of those representations is determined using SAT and SMT solvers.

  Another way to describe nontrivial properties is *equivalence checking*. It consists in expressing a specification of the system in terms of input/output relations, then deriving whether the system and its specification are behaviourally equivalent.

It is common to combine several verification techniques (e.g. [Hun93, HS96, Amj04]) to exploit their best capabilities while reducing their shortcomings. In this thesis, we focus

[1] The term "temporal ordering" should not be confused with the real-time aspect of reactive systems. The meaning here is the relative order of events, time being abstracted.

on the model checking technique, adequate to address the subtleties of concurrency.

While transition systems are adequate to model concurrent components interacting with each other, building them manually is complex and error-prone. Moreover, the slightest modification in the specification may involve drastic changes in the structure of the transition system, making it cumbersome to debug and modify. For these reasons, transition systems are often derived from descriptions written in high-level formalisms, by automatic translation. There are several high-level formalisms, each emphasising a specific class of systems.

In the sequel, we present the synchronous approach, well-adapted to deal with sequential and parallel systems such as hardware circuit designs and embedded systems. We also present the asynchronous approach, well-adapted to deal with parallel and distributed systems such as telecommunication protocols and distributed software.

2.2 The synchronous approach

A reactive system is synchronous if it reacts instantaneously to its environment and has a deterministic behaviour.

A reaction is assumed instantaneous if on the arrival of some inputs from the environment, a system reacts fast enough to produce the corresponding outputs, before the arrival of the next inputs. Hence, the system behaves as an infinite loop, called *synchronous loop*, each iteration corresponding to a reaction. In the remainder of this thesis, we will use the terms *step* to denote the reaction of a synchronous system and *activation* its ability to perform a step at a specific logical instant.

Synchronous concurrent components composing the system are cadenced by its synchronous loop. Component (micro-) steps may be *idle*, i.e., their inputs and outputs keep the same values as in the previous step. Messages emitted by components are received by other components in the same step; such communication is called *instant broadcast*. Instantaneous computations and communication are usually called *zero-delay* (or *synchrony*) assumptions.

A consequence of the synchrony assumptions is the determinism of the synchronous loop: given the same initial state and sequence of inputs, the same sequence of outputs will be produced. Determinism is desirable in safety-critical environments, in which a simple bug can have extreme consequences. Another consequence of the synchrony assumptions is the potential presence of temporal paradoxes, called *causality problems*. Examples include instantaneous dependencies between inputs and outputs.

2.2.1 Synchronous languages

Theoretical foundations can (a priori) be traced back to R. Milner's synchronous process algebra SCCS (*Synchronous Calculus of Communicating Systems*) [Mil83], which

extends CCS with primitives to encode synchrony. SCCS gave birth to other process calculi such as Meije [AB84]. Several *synchronous programming languages* have followed applying the synchronous approach. According to their programming style, synchronous languages can be classified into *imperative* and *declarative* languages. We survey the most representative ones of each class.

Synchronous imperative languages are inspired by classical imperative languages, i.e., in which the program structure reflects the order in which operations execute. They allow a modular description of reactive systems that require complex control structures. Esterel [BG92] has a textual syntax close to parallel programming languages such as ADA and Occam. It provides, in addition to the classical algorithmic control structures, concurrency primitives inspired by SCCS and Meije, as well as preemption structures. Argos [Mar91] is a graphical language, based on the Statecharts [Har87] formalism. An Argos program consists of hierarchical *Mealy machines*. These are finite-state automata in which outputs and next-state are both determined by the current-state and the current inputs.

Synchronous declarative languages are inspired by earlier studies on dataflow models [Kah74, McG82]. They allow the description of reactive systems that perform intensive data computation. Programs are described as networks of interconnected operators, evolving in parallel, and triggered by input arrivals. Lustre [HCRP91] is a functional language with textual syntax. A Lustre program is based on Mealy machines, the notion of state being implicit, unlike Argos. Signal [LGGLBLM91] is a relational language, defining relations between input and output *flows* (timed sequences of values), rather than simple functions as in Lustre. Contrarily to other synchronous languages, Signal programs are not necessarily deterministic. Each component induces its own constraints, which restrict the nondeterminism of the program. The Signal compiler is able to check the determinism of the conjunction of all constraints. This is the essence of the so-called *multiclock* or *polychronous* semantic model.

Causality problems in synchronous languages are either forbidden using static constraints (e.g., in Lustre) or resolved by the compilers using causality analysis algorithms. Examples of such algorithms are conditional dependence graph in Signal and computation of fixpoints in Esterel. Additionally, most synchronous languages are equipped with delay operators, which keep track of the values carried by expressions from one program step to the next. Based on how delay operators are used in a program, the compiler builds automatically an *internal state*.

Other proposals extend existing general-purpose languages with synchronous behaviour. Reactive-C provides a programming style similar to the C language. SynchCharts [And95] is a graphical language combining features from Esterel and Argos. Esterel-C (ECL) [LS99] and Java-Esterel (Jester) [AFFSV01] combine Esterel-like constructs with respectively C and Java languages. Lucid [WA85] is a higher-order functional language combining Lustre-like constructs and built upon Ocaml (Objective Caml) [LDG+03].

2.2.2 Functional verification

Due to the massive use of synchronous systems in safety-critical environments, many analysis techniques have been exploited for synchronous systems encompassing automated test, model checking, SMT-Solving, and abstract interpretation. We focus here on the techniques related to our work, namely the verification by model checking. Functional properties that a synchronous system should satisfy fall into two classes: *safety properties*, expressing that something bad will never happen; and *bounded liveness properties*, which are timing properties expressing that something good will happen within a bounded future.

A convenient way to express properties is an application of [VW86], where the negation of a property is described by an automaton. The synchronous product of this automaton with the program ensures that no trace of the program is accepted by the automaton. Since almost all synchronous languages synthesise finite automata (e.g., Mealy machines) from programs and since the parallel composition in those languages is synchronous, properties can be expressed directly in the synchronous language. These properties, called *synchronous observers* [HLR93], are auxiliary programs which observe the inputs and outputs of the program under verification and decide whether it is correct.

Many software verification tools have been developed to model check synchronous systems. *Xeve* [Bou98] is a model checker for Esterel programs. The tool compiles synchronous programs into a finite state automaton over which properties such as deadlock and starvation absence can be checked. *Lesar* [HR99] is a model checker, which compiles a Lustre program into a finite state automaton and implements the verification by observers. Both model checkers interface with automata-based tools such as *Auto* and *Autograph* [RdS90]. In [MRBS01], Signal programs are verified by means of equation systems, thus avoiding state space enumeration. Properties such as invariance and reachability can be verified.

Model checking has been mainly used to verify properties depending only on logical dependence between events. Traversing the set of control states of a validation program can be either enumerative or symbolic. For properties involving numerical values, such as bounded liveness properties, abstract interpretation techniques are seemingly more appropriate [HPR97]. However general liveness properties involving unbounded future are rarely addressed in synchronous languages. Expressing such properties requires more expressive and complex formalisms such as temporal logics and Büchi automata.

2.3 The asynchronous approach

A reactive system is asynchronous if the time in which an event occurs and its duration are considered of less concern.

While the asynchronous approach deliberately abstracts from the precise timing of events, the order of some events (sequential composition) and their simultaneity can

be described. This way of modelling provides asynchronous models with simplicity and abstraction, making them appropriate for modelling distributed and concurrent systems.

2.3.1 Communication models

Communication mechanisms are required to enable interaction between concurrent components, as the time in which input and output events occur is unspecified. Existing communication mechanisms include *shared memories* and *message-passing* communication.

Shared memory communication, introduced by Dijkstra [Dij65], enables concurrent components to communicate by altering the contents of shared locations. Languages adopting this mechanism include Java and C#. The access of concurrent components to shared locations should be controlled, for example, by using *mutual exclusion* protocols. Well-known examples of such protocols are Peterson's and Dekker's protocols.

Message-passing enables concurrent components to communicate with each other by exchanging messages. Message exchange can be either *synchronous* or *asynchronous*.

Asynchronous message-passing takes arbitrary delay, i.e., the elapsed time between message emission and reception is abstracted away. This requires the introduction of buffers or *channels* as proposed by Dijkstra [Dij72] which serve to store messages before their transmission. As such, asynchronous message-passing does not force participant components to wait for each other to communicate. Message-passing channels has been adopted by specification languages for communication protocols, such as the ITU standard SDL [BHS91] and Promela [Hol91]. In Promela, channels store messages in first-in first-out order, by default. If synchronous communication is required, it can be modelled by setting the channel size to zero.

Synchronous message-passing requires messages emitted by a component to be received by other components at the same time instant. In the asynchronous abstraction of time, this requires the introduction of *communication events*, called *synchronisation* or *rendezvous* between the participant components. Synchronous message-passing is intended to be independent of the medium used to communicate. The communication medium, which may be a shared location, could itself be modelled as a subordinate component that synchronises with emitters and receivers. Synchronisation is blocking, i.e., it happens only when all participants are ready to communicate. As such, the emitter component blocks until message reception, after which the different components evolve independently. Beyond message exchange, (dataless) synchronisation can be used to express the simultaneity of specific events of concurrent components. Synchronous message-passing is the main interaction paradigm used in process algebras such as CCS [Mil89], CSP [Hoa85], ACP [BK85], and LOTOS [BB87].

Hiding and *nondeterminism* operators, which are specific to some asynchronous languages, provide behavioural descriptions with high abstraction capability. The hiding

operator is essentially present in process algebra. It transforms events into *invisible* ones, i.e, event occurrence is neither detectable nor controllable by the environment. Synchronisation on invisible events is forbidden.

Nondeterminism is aimed at accurately disregarding irrelevant aspects of the actual system. Examples of situations in which nondeterminism is helpful are the following:

- Modelling concurrency by interleaving, thus abstracting from the speed of concurrent components.
- Abstracting from complex details of the physical environment.
- Abstracting from implementation details either because these are considered irrelevant or because the aim is to develop a simplified system meeting primary specifications before refining it to meet more detailed ones.

2.3.2 Functional verification

State space exploration techniques, including reachability analysis and model checking, are the most widespread approaches for dealing with concurrent systems containing complex data structures. CADP [GLMS13] and Spin [Hol04] are seemingly the two oldest model checkers that are still actively maintained and that benefit from a worldwide user community. Both tools support *on-the-fly* techniques, which consist in constructing and exploring state spaces on demand, guided by the verification task instead of generating state spaces exhaustively and then performing verification. This provides a way to fight against state space explosion, essentially caused by asynchronous concurrency and complex data structures.

CADP supports several input specification languages, among which LNT [CCG+16], LOTOS and FSP [MK06]. They rely on an action-based semantic model; by considering systems whose behavioural semantics can be represented using labelled transition systems. Model checkers of CADP are based on branching-time logics, which are adequate with bisimulation reductions and compositional verification. More details about the CADP toolbox will follow in Section 2.4.

SPIN supports Promela as input language. Promela relies on a state-based semantic model and considers systems whose behavioural semantics can be represented using Kripke structures. Correctness properties can be specified as process invariants, using assertions, as LTL formulas, or as formal Büchi automata.

Among other well-known tools, CWB [CPS89] (*Concurrency Workbench*) deals with CCS process algebra. CWB-NC [CLS00] (*Concurrency Workbench of the New Century*) is the continuation of research started in the Concurrency Workbench project. FDR [GABR14] is a model checker for verifying systems modelled in CSP. The mCRL2 toolset [CGK+13] is based on a variant of the ACP process algebra equipped with abstract data types.

2.4 The CADP toolbox for the verification of asynchronous systems

CADP is a modular software toolbox implementing the results of concurrency theory in the context of asynchronous concurrent systems. Started in the mid 80s, CADP includes today more than 50 tools and code libraries, among which compilers for various formal specification languages, equivalence checkers, model checkers, compositional verification tools, and performance evaluation tools. We focus here on some salient features of the languages and tools related to our work and required to the comprehension of this thesis.

LNT (*Lotos New Technology*) [CCG+16] is a specification language derived from the ISO standard E-Lotos [ISO01]. The LNT.OPEN tool translates LNT specifications into LTSs, given in BCG (*Binary Coded Graphs*) file format, suitable for on-the-fly exploration. Section 2.4.2 presents informally a subset of the LNT language.

MCL (*Model Checking Language*) [MT08] is an expressive temporal logic, extending the alternation-free μ-calculus [EC82] with generalised regular expressions, data-based constructs, and fairness operators. The EVALUATOR 4.0 model checker implements an efficient on-the-fly model checking procedure for MCL. It also exhibits full diagnostics (examples and counterexamples) as subgraphs of the LTS illustrating the truth value of MCL formulas. Section 2.4.3 presents informally a subset of the MCL language.

LTS minimisation is possible by using equivalence checking, which collapses the equivalent states in the LTS. Several equivalence relations are implemented in CADP, including strong [Par81], branching [vGW89, vGW96], and divergence-sensitive branching [vGW89, vGW96] bisimulation relations. A definition of those relations together with a formalisation of LTSs is given in Section 2.4.1.

SVL (*Script Verification Language*) [GL01, Lan02] is both a high-level scripting language proposed to CADP end-users and a compiler that translates SVL scripts into Bourne shell scripts. SVL enables to express complex verification scenarios, including property specification, LTS minimisation, abstraction, comparison, which orchestrates calls to the CADP tools.

2.4.1 Labelled Transition Systems (LTS)

Definition 2.1. (Labelled transition system) An LTS is a quadruple $(\mathbf{S}, \mathbf{L}, \rightarrow, s_0)$ where:

- $\mathbf{S}$ is a set of states.
- $\mathbf{L}$ is a set of labels.
- $\rightarrow \subseteq \mathbf{S} \times \mathbf{L} \times \mathbf{S}$ is the labelled transition relation.
- $s_0 \in S$ is the initial state.

■

We write $s \xrightarrow{\ell} s'$ as a shorthand for $(s, \ell, s') \in \rightarrow$. There exists a label, written τ or i, called the *invisible label*, which denotes internal actions. All labels different from τ are called the *visible* labels. An LTS is finite if its sets of states and transitions are both finite.

LTS equivalences Several equivalence relations between LTSs are available in the literature, differing mainly in the way they treat invisible labels. We focus on a few of them, namely strong bisimulation, branching bisimulation and its divergence-sensitive variant.

Definition 2.2. (Strong bisimulation) A strong bisimulation is a symmetric relation $R \subseteq S \times S$ such that if $(s_1,\ s_2) \in R$, then for all $s_1 \xrightarrow{a} s_1'$:

- there exists s_2' such that $s_2 \xrightarrow{a} s_2'$, and
- $(s_1',\ s_2') \in R$.

Two states s_1 and s_2 are strongly bisimilar if there exists a strong bisimulation R such that $(s_1,\ s_2) \in R$. Two LTSs are strongly bisimilar if their initial states are strongly bisimilar. ■

Definition 2.3. (Branching bisimulation) A branching bisimulation is a symmetric relation $R \subseteq S \times S$ such that if $(s_1,\ s_2) \in R$, then for all $s_1 \xrightarrow{a} s_1'$:

- either $a = \tau$ and $(s_1',\ s_2) \in R$, or
- there exists a sequence $s_2 \xrightarrow{\tau^*} s_2' \xrightarrow{a} s_2''$ such that $(s_1,\ s_2') \in R$ and $(s_1',\ s_2'') \in R$.

Two states s_1 and s_2 are branching bisimilar if there exists a branching bisimulation R such that $(s_1, s_2) \in R$. Two LTSs are branching bisimilar if their initial states are branching bisimilar. ■

Branching bisimulation does not distinguish between inaction and a cycle of internal actions. Divergence-sensitive branching bisimulation (or divbranching bisimulation for short) is introduced to take into account cycles of internal actions.

Definition 2.4. (Divergence-sensitive branching bisimulation) A divbranching bisimulation is a branching bisimulation R such that if $(s_1^0,\ s_2^0) \in R$ and there is an infinite sequence $s_1^0 \xrightarrow{\tau} s_1^1 \xrightarrow{\tau} s_1^2 \xrightarrow{\tau} \ldots$ with $(s_1^i,\ s_2^0) \in R$ for all $i \geq 0$, then there is an infinite sequence $s_2^0 \xrightarrow{\tau} s_2^1 \xrightarrow{\tau} s_2^2 \xrightarrow{\tau} \ldots$ such that $(s_1^i,\ s_2^j) \in R$ for all $i,\ j \geq 0$.

Two states s_1 and s_2 are divbranching bisimilar if there exists a divbranching bisimulation R such that $(s_1, s_2) \in R$. Two LTSs are divbranching bisimilar if their initial states are divbranching bisimilar. ■

2.4.2 The LNT language

The LNT formal language is rooted in a core powerful language, combining mainstream imperative and functional traits, which is smoothly extended with concurrency-related primitives. LNT is endowed with formal operational semantics defined in terms of LTSs (see [CCG+16] for a detailed presentation).

Types, statements, and functions

LNT provides constructed data types, statements built upon standard algorithmic control structures, and functions.

Types LNT types encompass basic types such as Boolean, integers, floating-point numbers, and character strings, as well as user-defined (possibly unbounded) data types such as records, unions, lists, sets, and arrays. The following is an example which defines a simple enumerated LNT data type *temperature*:

```
1  type temperature is
2     low, normal, high, very_high
3     with "!=", "==", "<", "<=", ">", ">="
4  end type
```

The "**with**" clause specifies the predefined functions for type *temperature*.

Statements LNT statements build upon standard algorithmic control structures, such as variable assignment, sequential composition, conditional (**if-then-else**), pattern matching (**case**) statements, and loops (**for**, **while**). In particular, statement "**var** X : T **in** I **end var**" declares variable X of type T in the scope of statement I. Hence, LNT dissociates between variable declarations (between the keywords **var** and **end var**) and variable modifications (inside I).

The following is an example of a statement which declares a variable, to which it assigns a value.

```
1  var ambient: temperature in -- declaration
2    ambient := normal         -- assignment
3  end var
```

Functions LNT functions can have **in** parameters (call by value), **out** parameters (call by reference, the function being in charge of producing a value for the parameter), and "**in out**" (call by reference, the function being allowed to read and update the parameter value). Actual parameters are preceded by symbols "`!`", "`?`", and "`!?`", respectively. Functions are deterministic and execute atomically without producing transitions in the generated LTS.

The following is an example of a function *check_temperature* reading the ambient temperature and raising an alarm if the temperature is high:

```
1  function check_temperature (in ambient: temperature, out alarm: bool) is
2    if (ambient >= high) then
3      alarm := true
4    else
5      alarm := false
6    end if
7  end function
```

The following is an example combining a call to the function *check_temperature*, with the **var** operator and variable assignment.

```
1  var ambient: temperature, alarm: bool in
2    ambient := normal;
3    eval check_temperature (ambient, ?alarm)
4  end var
```

Note that LNT types, statements, and functions look similar to mainstream programming languages, which favours the acceptance of the language by users compared to classical process algebras.

Behaviours and processes

Processes include functions, with the addition of gates, which are used for communication, and other behaviours built upon gates and process algebraic operators, including nondeterministic statements and parallel composition.

Nondeterministic assignment The behaviour "X **:= any** T [**where** V]" assigns to the variable X an arbitrary value of type T. The optional **where** clause with a Boolean expression V, supposed to use X, enables to express Boolean constraints on the possible values.

Nondeterministic choice The behaviour **select** B_1 **[]** ... **[]** B_n **end select** may execute either B_1, ..., or B_n.

The following are two semantically equivalent examples of nondeterministic assignment (left-hand side) and nondeterministic choice (right-hand side), involving type *temperature*:

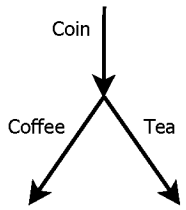
```
1  ambient := any temperature
2             where (ambient != very_high)
```

```
1  select
2    ambient := low
3  [] ambient := normal
4  [] ambient := high
5  end select
```

Gate communication The behaviour "G (O_1 ... O_n)" defines a communication on gate G. Offers O_1, ..., O_n describe the data exchanged during the communication. Each offer is either an emission of some expression or a reception of some value in a variable, in which case it is prefixed by ?. There exists an internal gate noted i, which must be used without offer.

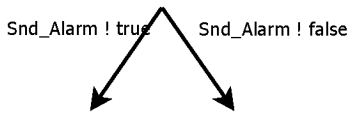
The following is an example of a behaviour for the coffee machine (Section 2.1.1, page 9), where *Coin*, *Coffee*, and *Tea* denote communication actions without offer. The behaviour defines the LTS on the right-hand side.

```
1  Coin;        — coin insertion
2  select       — drink serving
3     Coffee
4  [] Tea
5  end select
```



The following is an example of a behaviour assigning to variable *alarm*, a nondeterministically chosen value, which is emitted on gate *Snd_Alarm*. Such behaviour defines the LTS on the right-hand side:

```
1  alarm := any Bool;
2  Snd_Alarm (alarm)
```



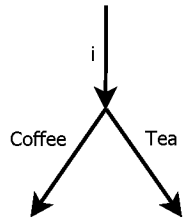
Channels Similarly to a variable, a gate is typed by a channel, which defines its profiles, i.e., the number and types of the values exchanged on the gate. There exists a predefined channel **channel none is () end channel**, with which any gate intended for dataless synchronisation can be declared. The following are some examples of channel definition.

```
1  channel Bool is (Bool) end channel
2  channel Temperature_Bool is (temperature, bool) end channel
```

Hiding Similarly to variables, gates can be declared either as formal gates in the process definition or locally inside a process. The behaviour "**hide** G_0 **:** Γ_0 **,** ... **,** G_n **:** Γ_n **in** B **end hide**" declares gates G_0, ..., G_n of respective channels Γ_0, ..., Γ_n that are only visible in the scope of behaviour B, i.e., hidden from the environment of B. Actions on hidden gates in the behaviour are substituted by the internal action i.

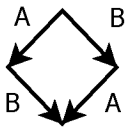
The following is an example of behaviour in which actions on gate *Coin* are hidden.

```
1  hide Coin: none in
2    Coin;
3    select
4       Coffee
5    [] Tea
6    end select
7  end hide
```



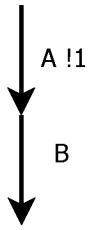
Parallel composition The behaviour "**par** G_0 **,** ... **,** G_n **in** B_0 || ... || B_m **end par**" defines a parallel composition of behaviours B_0, ..., B_m. Behaviours communicate by rendezvous on the set of gates $\{G_0, \ldots, G_n\}$, called *synchronisation set*. If the synchronisation set is empty, no communication occurs, in which case behaviours are said to execute in *pure interleaving*. Communication is blocking. If a behaviour is waiting for a communication whose gate belongs to the synchronisation set, then this communication can happen only if all behaviours B_0, ..., B_m can make this communication simultaneously. The following is an example of two actions A and B in pure interleaving:

```
1  par
2     A
3  || B
4  end par
```



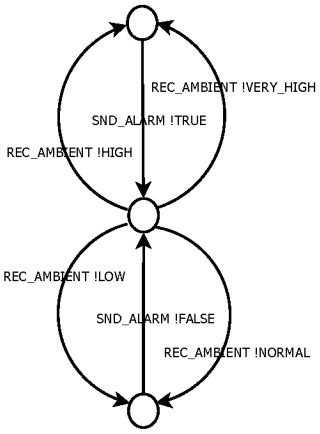
The following is an example of a parallel composition in which all behaviours synchronise on gate A. Behaviours do not synchronise on gate B (line 7), since the gate does not belong to the synchronisation set (line 1).

```
1  par A in
2    A (1)
3    || var X: Nat in
4          A (?X)
5       end var
6    || var X: Nat in
7          A (?X); B
8       end par
9  end par
```



Processes Processes allow behaviours to be named. A process can be parameterised by both data variables and gates. The following is an example of a process modelling a reactive system. Forever loops, which are absent in many traditional process algebras, allow a simple implementation of reactive systems.

```
1   process Check_Temperature [Rec_Ambient: temperature,
2                              Snd_Alarm  : bool]
3   is
4     var ambient: temperature, alarm: bool in
5       loop  -- forever
6         -- receive the ambient temperature
7         Rec_Ambient (?ambient);
8         -- check the temperature value
9         eval check_temperature (ambient, ?alarm);
10        -- emit an alarm
11        Snd_Alarm (alarm)
12      end loop
13    end var
14  end process
```



2.4.3 The MCL language

MCL enables a concise formulation of temporal properties, possibly parameterised by data values. The interpretation model is an LTS whose actions (transition labels) contain a gate name G followed by a list of values $v_1, ..., v_n$, exchanged during the rendezvous on G. Three kinds of formulas can be defined.

Action formula MCL action formulas, noted A, characterise actions of the LTS. An action formula is built from *action patterns* and the usual Boolean connectors. Action pattern "{G ?X:T **where** V}" matches every action of the form "G v", where v is a value of type T that is assigned to variable X, provided the Boolean expression V (which possibly uses X) evaluates to true. Variable X is exported to the enclosing formula. Action pattern "{G !e}" matches every action of the form "G v" where v is the value obtained by evaluating expression e. Action pattern "{G **?any**}" denotes a *wildcard* matching an arbitrary value regardless of its type.

Of course, it is possible to combine value matching ("!e"), value extraction ("?X:T"), and wildcard in the same action formula, for matching actions containing several values. Gate name G can also be extracted and manipulated as ordinary value of type *String*.

The following are some examples of action formulas.

The action formula: {*Rec_Ambient* ?*ambient*:*String* **where** *ambient* <> "*very_high*"} matches actions: {*Rec_Ambient* !"*low*"}, {*Rec_Ambient* !"*normal*"}, and {*Rec_Ambient* !"*high*"}, but does not match action: {*Rec_Ambient* !"*very_high*"}.

The action formula: {*Snd_Alarm* **?any**} matches actions: {*Snd_Alarm* **!true**} and {*Snd_Alarm* **!false**}.

Regular formula MCL regular formulas, noted R, characterise sequences of transitions in the LTS. A regular formula is built from action formulas and regular expression operators such as concatenation ("R_1 `.` R_2"), choice ("R_1 `|` R_2"), unbounded iterations ("R^*" and "R^+"), iterations bounded by counters ("R `{`n`}`"), etc.

For example, the regular formula: `{`*Coin*`}` `.` `(` `{`*Coffee*`}` `|` `{`*Tea*`}` `)` matches the following action sequences: $s_0 \xrightarrow{Coin} s_1 \xrightarrow{Coffee} s_2$ and $s_0 \xrightarrow{Coin} s_1 \xrightarrow{Tea} s_2$.

State formula MCL state formulas, noted F, characterise states of the LTS by specifying (finite or infinite) tree-like patterns going out from these states. A state formula is built from Boolean connectors, possibility ("`<`R`>`F") and necessity ("`[`R`]`F") modalities containing regular formulas, minimal ("**mu** X `.` F") and maximal ("**nu** X `.` F") fixed point operators, and the infinite looping ("`<`R`>@`") and finite saturation ("`[`R`]-|`") operators.

The following are some examples of state formulas. Deadlock absence can be detected by the following property, stating that every state has at least one successor: **`[true`*****`]<true>true`**. Cycles of internal actions can be detected by the property: **`<tau>@`**.

2.5 Globally Asynchronous Locally Synchronous (GALS) systems

Both synchronous and asynchronous languages and their dedicated verification tools have been used to deal with GALS systems. In this section, we review some of the existing approaches in this context.

2.5.1 GALS systems in synchronous languages and dedicated tools

The synchrony assumptions are hard to maintain in many classes of applications. Examples include distributed embedded software and large hardware designs in which precise clock distribution is infeasible or overly expensive. While local components of such systems are best modelled under the synchrony assumptions, communication mediums introduce asynchrony in the behaviour of the global system. At the same time, design and verification frameworks for synchronous systems are efficient and already integrated in the design process of many industrial systems. Hence, the motivation of some research track is to maintain the well-established methods and tools for synchronous systems for most of the design process.

Theoretical work on addressing asynchrony using synchronous formalisms can be traced back to the early eighties, when Milner showed that SCCS can simulate CCS [Mil83]. The main requirement is the ability to model a nondeterministic activation of synchronous components. Following Milner's approach, several other approaches [HB02, GG03, HM06] emulate asynchrony in synchronous formalisms, for example by means

of additional inputs used as activation conditions and to which arbitrary values are assigned.

While Milner's and related approaches adopt a single clock model, other authors propose a model with multiple loosely coupled clocks, called multiclock model. This model has been used to deal with systems intended to run in asynchronous distributed implementations based on lossless message-passing [GG10, GTL03, GG07]. Signal [GTL03] was the first language supporting the multiclock model, followed by Esterel giving birth to the Multiclock Esterel [BS01]. The CRP language (*Communicating Reactive Processes*) [BRS93] was the result of earlier work combining Esterel and CSP. A translation of CRP into the Meije process calculus has been proposed, thus enabling verification to be performed.

Another track of research [BCG99, PBCB06, PBDSST09, BBS12, AL03] is the synthesis of semantic-preserving GALS systems from synchronous programs, foreseeing their distribution. This approach favours correct-by-construction deployment of synchronous programs over GALS architectures. Several theoretical results on this concern are already supported by the Signal compiler. The tools *SynDEx*[2] and *Ocrep*[3] enable the automatic generation of distributed implementations, starting from a specification written in a synchronous language. However, these approaches generally do not deal with the formal verification of GALS systems.

All the aforementioned approaches mainly address *deterministic GALS systems* in which communication media are reliable: all messages are delivered in the order in which they have been received. However, a wide range of modern applications support unreliable communication media, such as recent LTTA (*Loosely Time-Triggered Architectures*) [BBC10, Sme13], which tolerate bounded loss of messages. Message loss is encountered by using an LTTA protocol, which ensures correct message transmission. In addition, modelling GALS systems in synchronous languages requires real-time guarantees, such as bounded computations and communication delays. Such guarantees may be unknown in the general case, or at least difficult to synthesise in some distributed applications. Examples of this kind are networks of PLCs, which evolve at arbitrary paces, the communication protocol (e.g., Modbus) being responsible of correct message transmission.

Last but not least, verification tools of synchronous languages do not support logics with sufficient expressiveness to capture general liveness and fairness properties, required for the verification of asynchronous concurrent systems. Synchronous model checkers (e.g., [HLR93, Bou98]) can express safety and bounded response properties but not properties in which the expected response may occur within unpredictable delay, as was mentioned in the end of Section 2.2.2.

To address general GALS systems, whose synchronous components evolve at unrelated paces and communicate along unreliable media with no real-time guarantees, verification

[2]http://www.syndex.org/
[3]http://pop-art.inrialpes.fr/~girault/Ocrep/

frameworks for asynchronous systems are more appropriate.

2.5.2 GALS systems in asynchronous languages and dedicated tools

Addressing GALS systems in asynchronous languages and their dedicated verification tools have genuine benefits. Asynchronous languages provide built-in parallel composition and abstraction operators to reason about asynchronous concurrent systems, abstractly and compositionally. Such operators enjoy useful compositionality properties such as congruence results. Hence, efficient state-space reduction techniques (e.g., partial order reduction) and compositional verification can be applied, for scaling to large systems. Compositional verification for asynchronous systems [GLM15] can be used to complement compositional verification approaches used for synchronous systems, such as assume-guarantee reasoning techniques (e.g., [BCMW15, GGTG10]). On the other hand, verification tools for asynchronous systems support logics with sufficient expressiveness to capture complex properties. Examples are succession of events in time (arbitrarily far from each other), cycles denoting infinite executions, and general liveness properties.

We have identified two main approaches in the literature addressing GALS systems in design and verification frameworks for asynchronous languages. A first approach consists in translating a GALS-specific language into a process language.

A translation from CRSM (*Communicating Reactive State Machines*) [Ram98], a visual language built upon CRP, into Promela is proposed in [RSD+04]. Verification is achieved by means of distributed observers to get rid of using temporal logics. The reliance of CRSM on Esterel entails a lack of data-driven support in the language. Indeed, most of the data-handling part of Esterel is deferred to the host language (e.g., C, C++, Java).

SystemJ [MSRG10] extends Java with Esterel-like synchronous model and CSP-like asynchronous model. Hence, unlike CRSM, it inherits the rich data-computation capabilities of Java. Components (called *clock-domains*) of SystemJ are deterministic and their asynchronous composition introduces nondeterminism. Such nondeterminism is still difficult to verify in the SystemJ framework. Efficient code can be automatically generated from SystemJ programs, but relies on Java virtual machines as target. This makes the language unsuitable for systems with limited resources. Recently [PMS15], a translation has been defined from a subset of SystemJ to LTL formulas, from which networks of Mealy automata are synthesised and translated into Promela, thus making possible the verification using SPIN.

Another approach consists in combining synchronous languages and asynchronous process languages. Synchronous components are encapsulated in asynchronous processes (called *wrappers*) to interface with other components. Asynchronous behaviour is described by introducing additional components, in the asynchronous language, to implement communication media.

This approach has been first implemented in [DMK+06], where Signal modules are compiled into C programs, which are encapsulated into Promela wrappers. Wrappers describe an infinite loop of atomic steps, by using the *atomic* construct of Promela. In each loop iteration, all possible values of inputs are generated; then, the C program is invoked together with clock constraints; finally, only if the clock constraints are met, outputs are computed. The asynchronous composition of wrappers is ensured via specific hardware communication buses, based on an early version of an LTTA protocol. Buses are abstracted as Promela finite FIFO channels, which are proven equivalent to one-place channels. Verification is performed by using LTL (*Linear Time Logic*) formulas.

The Signal-Promela approach, relying on state-based with linear-time semantics, is followed up by an approach combining the SAM synchronous language and the LNT asynchronous language in an action-based setting with branching-time semantics [GT09]. In this approach, SAM automata, which are extended Mealy machines, are translated into LNT functions that are encapsulated into LNT wrapper processes. Atomicity of synchronous components is described in functions but not their wrapper processes. As a result, individual input and output actions of the different LNT wrappers can interleave arbitrarily. Furthermore, the asynchronous composition between processes is completely arbitrary, since no constraints are put on their executions, contrarily to the Signal-Promela approach. As such, the maximal degree of nondeterminism is considered. This approach is used to check that an airplane-ground communication protocol, based upon TFTP/UDP (*Trivial File Transfer Protocol/User Datagram Protocol*), ensures correct message transmission. Abstractions and compositional verification are used to cope with state space explosion. Verification by model checking and performance evaluation are applied by using CADP.

The Signal-Promela and SAM-LNT approaches address specific GALS applications with specific activation strategies and communication protocols. Hence, their usage is not transferable to general GALS systems. While both approaches pave the way for addressing GALS systems in verification tools for asynchronous systems, one may wonder if having two different modelling languages is easy to learn.

Chapter 3

The GRL Language for GALS Behavioural Description

This chapter presents the GRL language, a new formal language for the behavioural description of GALS systems. It serves as a tutorial for GRL, the semantics of which will be presented in chapter 4. We first present a running example. Then, we introduce the design choices of the synchronous and asynchronous models adopted in GRL. We present the formal syntax and intuitive semantics of GRL constructs. Finally, we compare the expressiveness of GRL with regards to some existing approaches in modelling GALS systems.

3.1 A GALS example

> It is hard to learn the characteristics of a programming language by reading a formal definition of that language, until several examples have been studied
>
> D. E. Knuth, 1967

In this section, we present an example of a GALS system. The example will serve to illustrate the formal syntax of GRL in subsequent sections. It consists of a car park management system, whose goal is to control the availability of a double-storey car park[1]. The car park has one principal entrance and one principal exit gate. Each storey of the car park has also its own entrance gate. Gates are equipped with barrier systems enabling automatic vehicle detection. The car park availability is displayed to drivers via exterior lights mounted at the car park entrance.

[1] This application is a toy example designed in the framework of the Bluesky project, with the help of the industrial partners.

The actual system comprises four *Programmable Logic Controllers* (PLCs) that communicate to each other the entrance and exit activity. An *entrance PLC* and an *exit PLC* manage the entrance and exit gates, respectively. Two *storey PLCs* manage the gates of the two storeys.

The behaviour of the system is as follows. The availability of the car park is managed by the entrance PLC. The PLC checks continuously whether there still are unoccupied parking spots, in which case a green light is maintained on; otherwise, a red light is turned on. Once a car arrives and asks for entering, a request to open the gate is detected by the entrance PLC. The request can be either granted or denied, depending on the car park availability. If the access is granted, the PLC delivers a ticket indicating a storey reference. Then, the entrance gate remains open for a fixed amount of time and a yellow light is turned on until the gate closure. The car should go to the storey referenced in the ticket. A storey PLC grants the access only if the car ticket indicates the corresponding storey reference; otherwise, the car is informed that its destination is wrong. When a car asks for leaving the car park, an exit request is detected by the exit PLC, which opens the gate immediately. Once the car leaves, the exit PLC informs the storey PLC referenced in the car ticket, which in turn informs the entrance PLC to update the car park availability.

3.2 Overview of GRL

GRL syntax is presented in Tables 3.1 (page 31) to 3.9 (page 51). The generic terminal symbols and non-terminal symbols are summarised in the following table.

	Symbol	Description
Generic terminal symbols	P	*module identifier*
	S	*system identifier*
	B	*block identifier*
	N	*environment identifier*
	M	*medium identifier*
	C	*constant identifier*
	F	*function identifier*
	f	*record field identifier*
	T	*type identifier*
	K	*literal constant*
	X	*variable*
Non-terminal symbols	E	*expression*
	I	*statement*

3.2.1 Modules

A GRL specification can be structured in several *modules*, allowing single monolithic specifications to be split into reusable pieces of manageable size. The syntax of modules

is given by the grammar in Table 3.1. A module can import other modules, provided there are no circular dependencies. For example, in the module:

```
1  module Car_Park (Entrance, Storey1, Storey2, Exit) is
2    ...
3  end module
```

none of the imported modules *Entrance*, *Storey1*, *Storey2*, *Exit* must import module *Car_Park*, even when the import relation is extended to its transitive closure.

A module can contain the following constructs:

- *types*; *named constants* of any type;
- *blocks*, which denote synchronous components;
- *mediums*, which denote asynchronous communication mediums;
- *environments*, which describe constraints of the external environment on blocks; and
- *systems*, inside which are composed blocks, mediums, and environments.

The lexical scope of these constructs encompasses both the current module and its importing modules. In the sequel, blocks, mediums, and environments are called *components*.

module	::=	**module** P [(P_0, ..., P_n)] **is** (*type_definition* | *constant* | *block* | *medium* | *environment* | *system*)* **end module**

Table 3.1: Syntax of GRL modules

Example 3.1. The car park application can be described in GRL as follows (see Example 3.18 for excerpts of the system). Since each PLC has a synchronous behaviour, each of them will be described by a GRL block, named respectively *Entrance*, *Storey1*, *Storey2*, and *Exit* (see Figure 3.1 for a schematic view of block *Exit* and Example 3.7, page 42, for its corresponding GRL code).

PLCs are spatially distributed and communicate with each other. Thus, we will introduce two mediums, named *Exit_to_Storey1* and *Storey1_to_Entrance*, to describe communication from *Exit* to *Storey1* and from *Storey1* to *Entrance*. Similarly, we will introduce mediums *Exit_to_Storey2* and *Storey2_to_Entrance* (all mediums are instances of component *Sampling*, defined in Example 3.15, page 50).

A ticket given to a car contains two fields. Each field references a storey. We wish only one field to be selected in a ticket. We will describe this constraint in an environment *Env_Storey* (see Example 3.8, page 45). Finally, the same program will be implemented on both storey PLCs. This entails that both PLCs have nearly the same period. We will describe this constraint in an environment *Quasi_Synch_2* (see Example 3.12, page 47). □

3.2.2 Synchronous blocks

The synchronous model of GRL is rooted in imperative programming style with functional flavour. It provides a built-in definition of the synchronous loop (which is a deterministic infinite loop) and internal state notions. The discrete, logical model of time, adopted in synchronous programming, is considered. Synchronous concurrency is not supported. Communication is carried out by instantaneous broadcasting.

More concretely, the behaviour of a block is described as a (potentially unbounded) sequence of discrete deterministic steps. Throughout these steps, an internal state represented by state variables is maintained. Each step consists in first reading inputs, then computing outputs and the next internal state, which both depend on the inputs and internal state of the current step. These activities are performed simultaneously, making the step atomic, as is assumed in synchronous programming.

Blocks can be composed to interact with each other inside higher-level blocks, in a modular way. Modularity enables a textual description of hierarchical block compositions, as the one illustrated in Figure 3.1. Lower-level blocks are called *subblocks*. A block that is not a subblock of another block is called *highest-level block*. Composition of subblocks is synchronous, i.e., in every step of the enclosing block, each subblock performs a (micro-) step. To enable interaction between subblocks, inputs of some subblocks can be connected to outputs of preceding subblocks. Such interaction occurs instantaneously, as is assumed in synchronous programming. Accordingly, outputs produced by a subblock are consumed by the other ones in the same step of the enclosing block. This way, data is processed along causal dependencies between subblocks, making the behaviour of blocks deterministic.

3.2.3 Asynchronous composition of blocks

The asynchronous model of GRL is rooted in process algebras, while keeping the same imperative programming style as the synchronous model. The abstraction of time adopted at the synchronous level cannot be preserved anymore at the asynchronous level. Asynchronous concurrency is captured by the interleaving semantics. The basic model of communication is synchronous rendezvous. Asynchronous communication is enabled by means of dedicated components, i.e., mediums. Nondeterminism can be explicitly expressed through dedicated primitives.

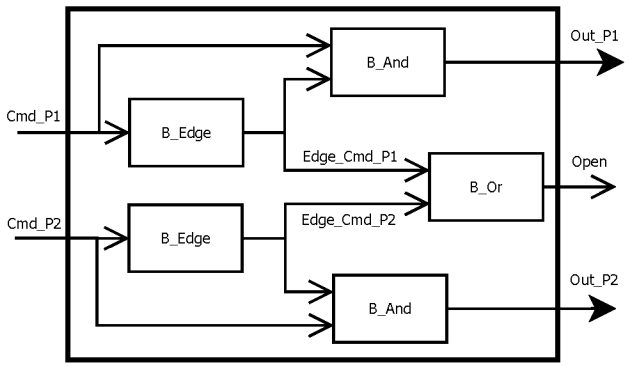


Figure 3.1: Schematic representation of the exit PLC (car park application)

More concretely, highest-level blocks are composed inside systems in asynchronous concurrency. The atomic deterministic steps of concurrent blocks interleave arbitrarily, without causal dependency. Contrarily to subblock composition, synchronous interaction between highest-level blocks is forbidden to prevent them from constraining the steps of each other. By default, two consecutive steps of a block may occur arbitrarily far from each other, unless specified differently by using *activation constraints* (see Section 3.5.2). This abstraction makes GRL expressive enough to model general GALS systems, while abstracting away from implementation details.

Blocks interact synchronously with mediums and environments. Connections between components are made by means of *channels*, which are tuples of variables, over which rendezvous take place. Channels are unidirectional, i.e., a channel is used by a component either only for *reception* or only for *emission* of tuples of values. Interactions between components are necessarily initiated by blocks, to which environments and mediums respond. In this respect, environments and mediums are passive components, executing only if requested by blocks, which are active components. Communications between blocks occur through mediums and is thus asynchronous.

Environments and mediums provide GRL with enough expressiveness to model and reason about general GALS systems. First, both of them are definable by the user, similarly to blocks. In addition, their behaviour may exhibit nondeterminism, a key feature providing descriptions with accuracy and high abstraction capability.

Environments enable constraints on block behaviour to be expressed at different levels of abstraction. On the one hand, they provide inputs to blocks and react to their outputs. Connections between blocks and environments are carried out using *input* channels (sets of inputs) and *output* channels (sets of outputs). An output channel of a block can be connected to an input channel of an environment, and conversely. On the other hand, environments can adjust the degree of asynchronous concurrency in block composition by

type	::=	**bool** | **nat** | **nat16** | **nat32** | **int int16** | **int32** | **char** | **string**
	|	T
type_definition	::=	**type** T **is**
		type_expression
		end type
type_expression	::=	**range** m ... n **of** *type*
	|	**enum** C_0 , ... , C_n
	|	**record** f_0 : *type*$_0$, ... , f_n : *type*$_n$
	|	**array** [m...n] **of** *type*

Table 3.2: Syntax of GRL type definitions

setting constraints on block activations. This allows to master the possible interleavings between blocks, e.g., a block cannot execute indefinitely in the detriment of the others. Activation strategies can model, at a suitable level of abstraction, realistic situations such as halting, priorities, and relative paces of synchronous components modeled as highest-level blocks.

Mediums enable blocks to communicate asynchronously. Connections between blocks and mediums are carried out similarly to the ones between blocks and environments, but on dedicated channels called *receive* and *send* channels. A medium receives messages from or sends messages to its connected blocks whenever requested. Messages can be stored in the internal state of the medium, thus enabling message buffering. Additionally, nondeterminism allows behaviours such as message loss, duplication, or reordering to be described naturally.

In the following sections, we first introduce GRL basic structures. We then present the behavioural constructs of GRL.

3.3 Basic GRL

In this section, we present the types, expressions, statements, and constants of GRL.

3.3.1 Type definitions

The syntax of types is given by the grammar in Table 3.2. GRL data types encompass predefined types such as Booleans (**bool**) or naturals of different sizes (8-bit **nat**, 16-bit **nat16**, or 32-bit **nat32**). Types can also be defined by the user (non-terminal *type_definition*). At the time of writing, types definable by the user are *ranges*, *enumerations*, *records*, and *arrays*.

Range types, defined using keyword **range**, denote finite intervals of numbers ranging from m to n, which must be literal constants of type *type*. Type *type* itself must be one of **nat**, **nat16**, **nat32**, **int**, **int16**, or **int32**. Here is an example which defines an interval of naturals from 0 to a value *nb_max_cars*, where *nb_max_cars* is a global constant assumed to be already defined (see Section 3.3.4).

```
1  type t_capacity is
2     range 0 ... nb_max_cars
3  end type
```

Enumerated types, defined using keyword **enum**, denote finite and ordered sets of symbolic values (identifiers) $C_0, \ldots, C_n$. Here is an example which defines an enumerated type *t_answer* that could be used in the car park application. The type has three values indicating the answer of the entrance PLC on the detection of an entrance request.

```
1  type t_answer is
2     enum idle, granted, denied
3  end type
```

Record types, defined using keyword **record**, denote fixed-size tuples of elements indexed by field names. Here is an example which defines a record type *t_msg*. The type stores information about a message.

```
1  type t_msg is
2     record cmd: bool, idx: nat, stat: t_capacity, ans: t_answer, full: bool
3  end type
```

Array types, defined using keyword **array**, denote fixed-size sets of elements indexed by natural numbers ranging from m to n, which must be literal naturals. Here is an example which defines an array *t_queue* of records of type *t_msg*, where *size_queue* is a global constant assumed to be already defined (see Section 3.3.4).

```
1  type t_queue is
2      array [0 ... size_queue] of t_msg
3  end type
```

3.3.2 Expressions

The syntax of GRL expressions is given by the grammar in Table 3.3. Predefined functions that can be used in a GRL module are unary operations, binary operations, type conversion functions, functions on arrays, and functions on records. Consider for example the record type *t_msg* (see Section 3.3.1). A predefined function *t_msg* is automatically generated. The call to function *t_msg* below returns a record in which each field is set to a value expression, where *idx_pre*, *updated_cars*, and *cst_max_cars* are variables of type **nat**, **bool**, and **bool**, respectively.

```
1  t_msg (true, idx_pre, 1, false, (updated_cars == cst_max_cars))
```

3.3.3 Statements

The syntax of statements is given by the grammar in Table 3.4. GRL statements are inspired by LNT ones. They extend standard algorithmic control structures with subblock invocations, communication primitives, and nondeterministic statements. Subblock invocations and communication primitives will be explained at appropriate places throughout the current chapter. Nondeterministic statements are similar to LNT ones.

E	::=	X	variable
	|	`(` E_0 `)`	parenthesised expression
	|	E_0 `.` f	record field access
	|	E_1 `[` E_0 `]`	array element access
	|	*unary_operator* E_0	unary operation
	|	E_1 *binary_operator* E_2	binary operation
	|	K [**of** *type*]	literal constant
	|	F `(` E_0 `,` ... `,` E_n `)`	predefined function

Table 3.3: Syntax of GRL expressions

I	::=	**null**	no effect
	|	X `:=` E	assignment
	|	X `[` E_0 `]:=` E_1	array element access
	|	X `.` f `:=` E	record field access
	|	I_0 `;` I_1	sequential composition
	|	**if** E_0 **then**	conditional
		I_0	
		elsif E_1 **then**	
		I_1 ...	
		elsif E_n **then**	
		I_n	
		else	
		I_{n+1}	
		end if	
	|	**while** E **loop** I_0 **end loop**	while loop
	|	**for** I_0 **while** E **by** I_1 **loop** I_2 **end loop**	for loop
	|	**case** E **is**	case selection
		K_0 `->` I_0	
		`|` ... `|`	
		K_n `->` I_n	
		`|` [**any** `->` I_{n+1}]	
		end case	
	|	B [`{`*args*`}`] `(`*args*`)`	subblock invocation
		/* The following statements are forbidden in blocks and reserved to environments and mediums */	
	|	**enable** B	activation signal
	|	**when** `<` X_0 `,` ... `,` X_n `>` `->` I_0	emission data signal
	|	**when** `?<` X_0 `,` ... `,` X_n `>` `->` I_0	reception data signal
	|	X := **any** T [**where** E]	nondeterministic assignment
	|	**select** I_0 `[]` ... `[]` I_n **end select**	nondeterministic choice
args	::=	arg_0 `,` ... `,` arg_n	
arg	::=	E | `_` | `?`X | `?_` | *any* T	

Table 3.4: Syntax of GRL statements

3.3.4 Global constant definitions

The syntax of constant definitions is given in Table 3.5. Constants are defined by keyword **const** and are necessarily initialised. A constant defines a variable whose value, once

initialised, can never be changed.

var	::=	X_0 , . . . , X_m : *type* := E_0
vars	::=	var_0 , . . . , var_n
constant	::=	**const** *vars*

Table 3.5: Syntax of GRL constants

Here are examples of constants of record type *t_msg* and array type *t_queue*, respectively.

```
1   const no_message : t_msg   := t_msg   (false, 0, 0, idle, false)
2   const empty_queue: t_queue := t_queue (no_message)
```

3.4 Blocks

In this section, we first present how GRL blocks can be defined and composed inside other components. We then discuss some design choices that distinguish GRL blocks from classical synchronous languages.

3.4.1 Block definition

The syntax of blocks is given by the grammar in Table 3.6. A block specification consists of formal parameters, local variables, subblock *aliasing*, and a statement I defining the block behaviour. This statement must be deterministic, i.e., use only the constructs described in the first 10 alternatives of the production defining I in Table 3.4. It consists of subblock invocations combined with standard algorithmic control structures.

block	::=	**block** B {$vars_c$}
		(**in** $vars_{i_0}$, . . . , **in** $vars_{i_m}$, **out** $vars_{o_0}$, . . . , **out** $vars_{o_n}$)
		[**receive** $vars_{r_0}$, . . . , **receive** $vars_{r_p}$, **send** $vars_{s_0}$, . . . , **send** $vars_{s_q}$]
		is
		alias B_0 {$args_0$} **as** B'_0 , . . . , B_k {$args_k$} **as** B'_k
		static var $vars_{p_0}$ . . . **static var** $vars_{p_r}$
		var $vars_{t_0}$ . . . **var** $vars_{t_s}$
		I
		end block
	|	**block** B {$vars_c$}(**in** $vars_{i_0}$, . . . , **in** $vars_{i_m}$, **out** $vars_{o_0}$, . . . , **out** $vars_{o_n}$) **is**
		!**c** *string* | !**Int** *string*
		end block

Table 3.6: Syntax of GRL blocks

Formal parameters are declared with types and possibly default values. They are classified into *constant*, *input*, *output*, *receive*, and *send* parameters. Constant parameters,

enclosed in braces, denote configuration data. A constant parameter is read-only, i.e., its value should not be changed in the body I of the block. Input and output parameters are preceded by keywords **in** and **out**. They enable blocks to interact either with other blocks (for subblocks) or with the environment (for highest-level blocks). Receive and send parameters are preceded by keywords **receive** and **send**. They enable highest-level blocks to interact with mediums, and consequently to asynchronously communicate with other highest-level blocks.

Receive and send parameters are introduced to make a clear distinction between synchronous interactions of a block with its environment on the one hand, and asynchronous communication with other blocks through mediums on the other hand. Accordingly, such parameters occur necessarily in highest-level blocks and cannot be used in subblocks inside other blocks. Conversely, a block having only input and output parameters cannot be used to communicate asynchronously with other highest-level blocks inside systems. If asynchronous communication is required, the block should be encapsulated inside another block having send and receive parameters.

Local variables are either *temporary* or *static*. The scope of both kinds of variables is limited to the enclosing block.

Temporary variables are preceded by keyword **var** and are optionally initialised at declaration time. Once a step starts, each temporary variable is first assigned its initialisation value (if any), which can be used in computations within the step. The updated value of the variable is lost at the end of the step, i.e, when returning from the block.

Static variables are preceded by keywords **static var**. Their initialisation at declaration time is mandatory, contrarily to temporary variables. When a block first step starts, each static variable is assigned its initialisation value. When the block subsequent steps start, each variable takes the value it had at the end of the previous step. In other words, the values of static variables updated within a step are kept stored for subsequent steps. Consequently, static variables are adequate to represent the internal state of the block.

The difference between static and temporary variables is their lifetime, which we illustrate in the listing below (left-hand side). The table on the right-hand side, shows the evaluation of variables x and y at the end of each of the first four steps of block B.

```
1  block B is
2    static var x:nat := 0
3    var y:nat := 0
4    x := x + 1;
5    y := y + 1
6  end block
```

step	1	2	3	4	...
x	1	2	3	4	...
y	1	1	1	1	...

Example 3.2. Block *B_Edge* below encodes an edge detector. It observes a logic signal *Logic_Signal* and decides whether the signal value has changed since the last block step. The static variable *Pre_Signal* stores the last value carried by the signal.

The block is parameterised to detect either rising or falling edges on *Logic_Signal*. The rising edge mode detects changes from *false* to *true*, and is activated by the constant parameter *Rising_Mode*. The falling edge mode detects changes from *true* to *false*, and is activated by the constant parameter *Falling_Mode*. According to the default setting of constant parameters, the rising edge mode is enabled by default.

```
1  block B_Edge {Rising_Mode: bool := true, Falling_Mode: bool := false}
2              (in  Logic_Signal : bool := true,
3               out Edge_Detected: bool) is
4    static var Pre_Signal: bool := false
5    var Rise, Fall       : bool
6    Rise          := Logic_Signal and not (Pre_Signal);
7    Fall          := not (Rise);
8    Edge_Detected := (Rising_Mode and Rise) or (Falling_Mode and Fall);
9    Pre_Signal    := Logic_Signal
10 end block
```

□

Alternatively, the behaviour of a block can be specified in an external language, a feature inspired by process languages (e.g., LNT and Promela). So far, the supported external languages are C and LNT. External C and LNT functions are declared using pragmas "**!c**" and "**!lnt**", respectively.

Example 3.3. To illustrate the use of external C code, consider the C function *Shift* below, which applies shift operations on a natural number. Type *GRL_Int16* is used in the function interface (line 2, C code). Before using a parameter, it should be converted to the C domain. This is done by the predefined function *GRL_Int16_To_Signed_Char* (line 4, C code). Then, before returning from the function, the result is converted to the GRL domain by using the predefined function *GRL_Signed_Char_To_Int16* (lines 6-7, C code).

The C function is written in a file with extension ".c", which is imported in the current GRL module. So doing, it can be encapsulated inside block *C_Shift*.

```
1  -- GRL file importing the C file named Shift
2  module External_C (Shift) is
3    block C_Shift (in Num :int16, out left, right :int16)
4      is
5        !c "Shift"
6    end block
7    ...
8  end module
```

```
1  -- C file named "Shift.c"
2  void Shift (GRL_Int16 Num, GRL_Int16* left, GRL_Int16* right)
3  {// convert types to the C domain
4   unsigned char arg_number = GRL_Int16_To_Signed_Char (Num);
5   // compute outputs and reconvert types to the GRL domain
```

```
6    *left  = GRL_Signed_Char_To_Int16 (arg_number << arg_bits);
7    *right = GRL_Signed_Char_To_Int16 (arg_number >> arg_bits);
8  }
```

□

Example 3.4. The use of external LNT code is straightforward as illustrated by the code below. Function *check_temperature* is assumed to be defined in a file with extension ".lnt" that is imported in the current GRL module.

```
1  -- GRL file importing the LNT
2  -- file named Temperature
3  block LNT_check_temperature
4    (in ambient: temperature,
5     out alarm : bool)
6  is
7     !lnt "check_temperature"
8  end block
```

```
1  -- Excerpt of the LNT file named Temperature
2  function check_temperature
3    (in ambient: temperature,
4     out alarm : bool)
5  is
6     ...
7  end function
```

□

Although including external code enhances user convenience, the external code should be defined to comply with GRL semantics. To enable functional verification, external C code should be side-effect-free, i.e., the same code called with the same input values in different contexts should return the same output values. In particular, blocks defined using external code must not have static variables. Fragments of external LNT code, however, have formal semantics and can thus be used safely, provided they do not use themselves external C code with side effects.

3.4.2 Subblock composition

A block can be invoked inside other components. Each invocation corresponds to an *instance* of the block. An instance is a copy of the block (and of a component in the general case) with a separate internal state. Instances of blocks invoked inside components are called subblocks.

At invocation time, actual parameters of subblocks are set. Actual output parameters are distinguished by a question mark. The question mark indicates that the parameter will have a value assigned when returning from the block. Underscores can be used as actual parameters. An underscore indicates that the actual parameter is presumed irrelevant for the caller component. For each constant and input parameter "_", the default value of the corresponding formal parameter in the block definition is used in each step. For each output parameter "?_", the value assigned to the corresponding formal parameter when returning from the block is just ignored.

Example 3.5. Below are three possible invocations of block *B_Edge* (see Exam-

ple 3.2). Note that the last invocation is useless, since the block output is unconnected.

```
1   B_Edge {true, false}(reset, ?detected)
2   B_Edge {_, _}       (_,     ?detected)
3   B_Edge {_, _}       (reset, ?_)
```

Correspondence between formal parameters of *B_Edge* and actual parameters used in the three invocations is given in the following table.

Formal parameter		Rising_Mode	Falling_Mode	Logic_Signal
Actual parameters	1	*true*	*false*	value of *reset*
	2	*true*	*false*	*true*
	3	*true*	*false*	value of *reset*

□

Subblocks can be *aliased*, i.e., assigned different names, by using keyword **alias**. If an aliased subblock has constant parameters, the corresponding actual parameters should be set at aliasing time instead of invocation time. This simplifies the presentation, especially since constant parameters do not participate in subblock interaction.

Example 3.6. In the code below, *Rising_Edge* is an alias of block *B_Edge* with the default values of constant parameters. *Falling_Edge* is another alias of the block, parameterised to detect falling edges.

```
1   alias B_Edge {_, _}       as Rising_Edge
2   alias B_Edge {false, true} as Falling_Edge
```

□

In the current version of GRL, there is no synchronous parallel composition operator. Rather, subblocks are composed in a sequential way. Their invocation should follow a user-defined topological order, in accordance with the causal dependency between their input-output connections. Subblocks with no causal dependency can be invoked in an arbitrary order.

Example 3.7. Block *Exit* below corresponds to the block composition depicted in Figure 3.1. The block implements a possible correct order for subblock invocations. For example, the actual output parameter "?*Edge_Cmd_P1*" of block *B_Edge* (line 5) should be broadcast to subblocks *B_And* (line 6) and *B_Or* (line 11). Hence, *B_Edge* should be invoked before *B_And* and *B_Or*.

Actual parameters *Cmd_P1*, *Cmd_P2*, *Open*, *Out_P1*, and *Out_P2* of subblocks *B_Edge*, *B_And*, and *B_Or* are declared as formal parameters of the enclosing block *Exit*. These parameters are intended to interact with the outside world of block *Exit*. Contrarily, actual parameters *Edge_Cmd_P1* and *Edge_Cmd_P2* are declared as temporary variables. They are used as input-output connections, internal to the block *Exit*.

```
1  block Exit (in Cmd_P1, Cmd_P2: bool, out  Open   : bool)
2             [send Out_P1       : bool, send Out_P2: bool] is
3    var Edge_Cmd_P1, Edge_Cmd_P2 : bool
4    -- if a car parking in Storey1 arrives
5    B_Edge {true, false}(Cmd_P1, ?Edge_Cmd_P1); -- does the car ask for leaving?
6    B_And  (Edge_Cmd_P1, Cmd_P1, ?Out_P1);      -- inform Storey1
7    -- if a car parking in Storey2 arrives
8    B_Edge {true, false}(Cmd_P2, ?Edge_Cmd_P2); -- does the car ask for leaving?
9    B_And  (Edge_Cmd_P2, Cmd_P2, ?Out_P2);      -- inform Storey2
10   -- if a car asks for leaving, open the gate
11   B_Or   (Edge_Cmd_P1, Edge_Cmd_P2, ?Open)
12 end block
```

□

Due to the absence of synchronous parallel composition operators, there are no causality problems in GRL. Undesirable cyclic dependencies between subblocks can be captured by the following semantic rules:

1. variables are necessarily assigned values before being read.
2. static variables are necessarily initialised at declaration time.

For example, GRL semantics forbid the following program because variable x is used without being initialised by $B1$.

```
1  block B is
2     var x, y: t
3     B1 (x, ?y);
4     B2 (?x, y)
5  end block
```

However, the following three programs are permitted, although programs 1 and 2 are not equivalent.

```
1  -- program 1
2  static var pre_x: t := e
3  var y: t
4  B1 (pre_x,  ?y);
5  B2 (?pre_x, y);
```

```
1  -- program 2
2    var x, y: t
3    x :=  e;
4    B1 (x, ?y);
5    B2 (?x, y)
```

```
1  -- program 3
2  x := 1;
3  x := x + 1;
4  -- "cyclic" dependency permitted
5  -- (imperative style)
```

3.4.3 Discussion and related work

The initial intention for defining GRL is to have an intermediate format that maps GALS systems, whose synchronous components are modeled using synchronous languages, to verification tools for asynchronous systems. For this reason, GRL does not include a full-fledged synchronous language. Rather, it provides a minimal but sufficient number of core constructs, to which the built-in constructs of synchronous languages can be translated with reasonable effort. We discuss below the design decisions distinguishing GRL blocks from classical synchronous languages.

Absence of parallel operators Synchronous languages are generally equipped with synchronous parallel operators while GRL is not. Composition between GRL blocks is sequential, requiring a correct order between blocks to be defined. If the GRL code was generated from a synchronous language, one would expect the front-end compiler to automatically provide such a correct order. This is reasonable since each compiler of a synchronous language implements a causality analysis algorithm.

We deliberately disallow synchronous concurrency. This enhances the integrability of GRL as back-end of various synchronous compilers as GRL cannot interfere with causality analysis algorithms that synchronous compilers implement. Also, most synchronous compilers generate efficient sequential code which can be usefully integrated in GRL.

Absence of delay operators In GRL, delay operators are deliberately absent. Instead, GRL makes explicit the internal state of blocks by means of static variables. Accessing and updating the internal state of blocks is in charge of the GRL user. Of course, delay operators could be encoded in GRL libraries and imported in GRL modules.

We believe that GRL is expressive enough to implement delay operators with no difficulty. As an illustration, consider the Lustre node *Counter* below. The node uses the operator "pre" (for previous), which gives the last value carried by its operand N.

```
1  node Counter  (init, incr: int, reset: bool) returns (N: int);
2     let
3        N = init -> if reset then init
4                    else pre(N) + incr
5     tel
```

An implementation of that node in GRL is block *Counter* below. Variable *preN* implements the Lustre "pre (N)" expression, by storing the value to be used in the next step. The Boolean variable *first* is used to emulate the Lustre operator "→" (followed by).

```
1  block Counter (in init:int, incr:int, reset: bool, out N: int) is
2     static var preN:int := init, first:bool := true
3        if reset or first then N := init; first := false
4        else N := preN + incr
5        end if;
6        preN := N
7  end block
```

Explicit loops Another difference with standard synchronous languages is the presence of explicit computation loops in GRL. For and while loops are useful when used as iterators. Problems such as loop boundedness should be ensured by the front-end compiler and are out of the scope of GRL. If ensuring loop boundedness were necessary, bounded operators could be encoded in GRL libraries. Those libraries, once verified, can be used safely in GRL programs.

3.5 Environments

Environments provide block inputs and react to block outputs, thus putting constraints on the data carried by the blocks. Additionally, they define constraints on block activation, thus enabling the description of activation strategies.

The syntax of environments is given by the grammar in Table 3.7. An environment specification consists of formal parameters encompassing constant, input, and output parameters; *activation* parameters prefixed by keyword **block** and denoting block identifiers; static and temporary variables; subblock aliasing used as routines; and a statement I defining the environment behaviour. This statement can be nondeterministic. It uses all the alternatives of the production defining I in Table 3.4, i.e., the same deterministic statements as blocks can be used, extended with nondeterministic assignment, nondeterministic choice, and *signals*.

blocks	::=	B_0 **,** $\ldots$ **,** B_n
env	::=	**environment** N {$vars_c$}
		(**in** $vars_{i_0}$ **,** $\ldots$ **,** **in** $vars_{i_m}$ **,** **out** $vars_{o_0}$ **,** $\ldots$ **,** **out** $vars_{o_n}$ **,**
		block $blocks_{b_0}$ **,** $\ldots$ **,** **block** $blocks_{b_p}$)
		is
		alias B_0 {$args_0$} **as** B'_0 **,** $\ldots$ **,** B_k {$args_k$} **as** B'_k
		static var $vars_{p_0}$ $\ldots$ **static var** $vars_{p_r}$
		var $vars_{t_0}$ $\ldots$ **var** $vars_{t_s}$
		I
		end environment

Table 3.7: Syntax of GRL environments

3.5.1 Data constraints

An environment interacts with a block either by reception (on input parameters) or by emission (on output parameters) of tuples of values; interactions are initiated by blocks. Each interaction being instantaneous (or synchronous), the parameters involved in the same interaction are grouped in *channels* of the form "**in** *vars*" or "**out** *vars*". Now, an environment may interact independently with several blocks, which trigger its execution in a nondeterministic way, according to the interleaving semantics. Hence, all the possible executions devoted to interaction on the different channels must be defined inside the environment. This requires additional communication primitives that guard the code (part) to be executed, whenever interactions on some channel occur. We introduce *data signals* as such communication primitives. A data signal, introduced by keyword **when**, is associated to each channel.

Example 3.8. Environment *Env_Storey* ensures that a ticket, given to a car references exactly one storey. Values carried by outputs *Cmd_P1* and *Cmd_P2* are determined in a nondeterministic way.

```
1  environment Env_Storey (out Cmd_P1, Cmd_P2: bool) is
2     when <Cmd_P1, Cmd_P2> -> Cmd_P1 := any bool; — select Storey1
3                              Cmd_P2 := any bool  — select Storey2
4                                        where not (Cmd_P1 and Cmd_P2)
5  end environment
```

□

Example 3.9. Environment *Cmd_Park* ensures that no request to enter the car park can be detected, if the entrance gate is already open. Details of the code will be explained in the sequel.

```
1  environment Cmd_Park (in Open: bool, out Cmd: bool) is
2     static var Pre_Open: bool := false  — gate status at the block last step
3     select
4       when ?Open -> Pre_Open := Open  — store gate status
5     []
6       when Cmd   -> if (not (Pre_Open)) then Cmd := any bool — allow request
7                     else Cmd := not (Pre_Open)               — disallow request
8                     end if
9     end select
10 end environment
```

□

The signal associated to each input channel "**in** $X_0 : T_0 , \ldots , X_n : T_n$" has the form "**when** ?<$X_0 , \ldots , X_n$> -> I_0". If $n = 0$, angle brackets are optional. The code I_0 guarded by the signal is *active* (meaning that I_0 can be executed), whenever a block connected to the channel produces its own outputs $X_0 , \ldots , X_n$. Then, the values of those variables, received on the channel, can be read only inside statement I_0. In Example 3.9, the signal "**when** ?*Open* ->" defined at line 4 is active each time the block connected to channel "**in** *Open*" finishes a step. In this case, the value of *Open* is read and assigned to *Pre_Open*, when returning from the environment. Signal "**when** *Cmd* ->" defined at line 6 is not active during this execution of the environment.

The signal associated to each output channel "**out** $Y_0 , \ldots , Y_m$" has the form "**when** <$Y_0 , \ldots , Y_m$> -> I_0". If $m = 0$, angle brackets are optional. The code I_0 guarded by the signal is active, whenever a block connected to the channel reads its own inputs $Y_0 , \ldots , Y_m$. This requires statement I_0 to assign values to those variables which are emitted on the channel. In Example 3.9, the signal "**when** *Cmd* ->" defined at line 6 is active each time the block connected to channel "**out** *Cmd*" starts a step. In this case, *Cmd* is assigned a value when returning from the environment. Signal "**when** ?*Open* ->" is not active during this execution of the environment.

Since interactions on a channel occur whenever requested by the connected block, there must be at least one reachable execution path in the environment containing the signal corresponding to the channel. So doing, environments do not prevent block executions.

In general, the code fragments guarded by the different signals are combined using nondeterministic choice, as illustrated in Example 3.9 (lines 3-9).

Besides, since exactly one signal is active during each environment execution, GRL semantics prohibit sequential composition of signals, loop statements containing signals, and nested signals. So doing, exactly one signal should be present in each execution path. Note, however, that the code associated to a given signal is not necessarily deterministic, which allows the environment to have a nondeterministic behaviour.

Static variables are particularly useful to keep track of past events, such as exchanged values or the history of block steps. This is illustrated in Example 3.9 where parameters *Cmd* and *Open* are intended to be connected to block *Entrance* input and output, respectively. The value carried by output *Cmd* depends on the last value that the input *Open* has carried. The information is stored in the static variable *Pre_Open*.

3.5.2 Activation constraints

Highest-level blocks evolve by default in arbitrary interleaving. Environments enable to control the level of asynchrony between block executions, by putting constraints on the activation of one or several blocks. The activation of blocks whose identifiers occur as activation parameters is intended to be constrained by the environment. Similarly to input and output channels, to each activation parameter of the form "B" is associated an *activation signal* of the form "**enable** B". The difference between activation signals and data signals is that the former are used only for synchronisation purposes and not for data exchange. An activation signal "**enable** B" implements the permission for a block (named B) to start a step. A block, whose activation is intended to be constrained by an environment, can execute only if there is at least one reachable execution path, containing its respective signal. Therefore, contrary to data signals, activation signals may be unreachable in certain execution contexts. In particular, if no signal is associated to a given activation parameter, the corresponding block is never activated. Hence, the unreachability of activation signals is equivalent to the "deactivation" of the corresponding block.

Example 3.10. Let B be a highest-level block, connected to environment *Disable*, defined below. If the environment is invoked with the default value of C, block B will never execute since its corresponding activation signal is never reached.

```
1  environment Disable {C: bool := true}(block B) is
2     if not C then enable B end if
3  end environment
```

□

Example 3.11. Consider a system composed of highest-level blocks *B1*, ..., *Bn*.

The default arbitrary interleaving between blocks is equivalent to the following activation strategy, where no constraint is put on block activations.

```
1  environment Default (block B1, ..., block Bn) is
2     select
3       enable B1 [] ... [] enable Bn
4     end select
5  end environment
```

□

Example 3.12. Environment *Quasi_Synch_2* implements an activation strategy for two blocks evolving with nearly the same period but without sharing clocks. This example illustrates how to express relations between the paces of different blocks.

```
1  environment Quasi_Synch_2 (block Comp_A, block Comp_B) is
2    -- Initially, Comp_A and Comp_B can be activated
3    static var ok_A, ok_B: bool := true
4    select -- activate Comp_A once
5      if (ok_A) then
6        enable Comp_A;
7        ok_A := false
8      end if
9     [] -- activate Comp_B once
10       -- ...
11    end select;
12    if (not (ok_A) and not (ok_B)) then
13       -- reinitialise
14       ok_A := true; ok_B := true
15    end if
16 end environment
```

□

In Example 3.12, activation signals are combined using **if-then-else** statements, to constrain the activation of connected blocks. The reachability of those signals depends on the internal state of the environment, i.e., its static variables, recording part of the history of block activations.

In general, activation constraints are a framework to abstract properties of real-time distributed systems in an asynchronous model. GRL enables to implement complex activation strategies involving priorities and arbitrary relations between the paces of synchronous components.

3.5.3 Combining data and activation constraints

Data and activation signals enable to constrain the behaviour of blocks at different levels of abstraction. The syntactic separation between both concepts makes the user intention clearer on how to fine-tune the system constraints. A data signal allows to constrain

data carried by block inputs and outputs and cannot handle block activations. It must be reachable whenever required by a connected block. An activation signal allows to constrain the activation of blocks at specific moments in time and cannot handle input and output data. The reachability of activation signals induces the activation strategy of blocks.

The two kinds of signals can be combined to describe complex situations. In particular, test scenarios can be described in an elegant and modular way.

Example 3.13. The following environment describes a block crash.

```
1  environment Crash (block B, out alarm: bool) is
2    -- when a failure is detected, block B halts and an alarm is triggered
3    static var failure: bool := false
4    select
5      if not (failure) then  -- no failure has been detected
6        enable B;  -- activate block B normally
7        failure := any bool  -- a failure may occur
8      end if
9    []
10     when alarm -> alarm := failure -- trigger an alarm in case of failure
11   end select
12 end environment
```

□

Example 3.14. The following GRL code specifies a test scenario for the car park application. A car enters the car park and is given a ticket. The car tries to access the second storey, contrarily to what is indicated on its ticket. The access to the storey is then denied and the car parks in the first storey. Finally, the car leaves the car park. The enumerated type *cases* defines the scenario steps.

```
1  type cases is
2     enum Car_Park, Car_P1, Car_P2, Car_Ex, None
3  end type
```

Environment *Scen_Act* defines the order in which blocks should be activated.

```
1  environment Scen_Act (block Entrance, Exit, Storey1, Storey2) is
2     static var action:cases := Car_Park
3     case action is
4       Car_Park -> action := Car_P2; enable Entrance
5     | Car_P2   -> action := Car_P1; enable Storey2
6     | Car_P1   -> action := Car_Ex; enable Storey1
7     | Car_Ex   -> action := None;   enable Exit
8     | any      -> action := None
9     end case
10 end environment
```

Environment *Scen_Data* defines the values that should be carried by the inputs of different blocks.

```
1  environment Scen_Data (out Cmd_Park          : bool, out Cmd_P11, Cmd_P12: bool,
2                         out Cmd_P21, Cmd_P22: bool, out Exit_P1, Exit_P2: bool)
3  is
4     select
5        -- a car enters the car park
6        when <Cmd_Park>          -> Cmd_Park := true  -- Entrance PLC data
7        -- the ticket given to the car indicates storey1
8        -- access to storey2 will be denied
9     [] when <Cmd_P11, Cmd_P12> -> Cmd_P11  := true; -- Storey2 PLC data
10                                  Cmd_P12  := false
11       -- the ticket given to the car indicates storey1
12       -- access to storey1 will be granted
13    [] when <Cmd_P21, Cmd_P22> -> Cmd_P21  := true; -- Storey1 PLC data
14                                  Cmd_P22  := false
15       -- a car parking in Storey1 asks to leave
16    [] when <Exit_P1, Exit_P2> -> Exit_P1  := true; -- Exit PLC data
17                                  Exit_P2  := false
18    end select
19 end environment
```

□

3.6 Mediums

Mediums are intended to implement the asynchronous communication between highest-level blocks. Their syntax is given by the grammar in Table 3.8. Medium specification is described similarly to environments, except that input and output channels are replaced by receive and send channels, and activation parameters are absent. A medium behaviour is defined by a nondeterministic statement, in which activation signals are not allowed.

med	::=	**medium** M {$vars_c$} [**receive** $vars_{r_0}$, ..., **receive** $vars_{r_m}$, **send** $vars_{s_0}$, ..., **send** $vars_{s_n}$] **is** **alias** B_0 {$args_0$} **as** B'_0, ..., B_k {$args_k$} **as** B'_k **static var** $vars_{p_0}$ ... **static var** $vars_{p_r}$ **var** $vars_{t_0}$ ... **var** $vars_{t_s}$ I **end medium**

Table 3.8: Syntax of GRL mediums

A medium interacts with highest-level blocks either by reception (on receive channel) or by emission (on send channel) of tuples of values, called *messages* in the sequel. To enable an asynchronous message transmission between a pair of blocks, a medium should interact (synchronously) with both blocks on separate channels. To each channel

is associated a data signal. The data signal must be reachable whenever required by a connected block, similarly to data signals inside environments. Nondeterministic choice is appropriate to combine data signals, since it does not lead to blocking situations. Static variables are particularly useful for message buffering.

Example 3.15. The following code implements a unidirectional one-place buffer. Channel *rec_msg* is devoted to receive messages. A received message is buffered using the static variable *buf_msg*, waiting to be emitted on channel *snd_msg*.

```
1  medium Sampling [receive rec_msg: Bool, send snd_msg: Bool] is
2    static var buf_msg: Bool := false — memory shared by connected blocks
3    select
4       when ?rec_msg -> buf_msg := rec_msg — message reception and buffering
5    [] when snd_msg  -> snd_msg := buf_msg — emission of the buffer content
6    end select
7  end medium
```

This model is used in *Loosely Time-Triggered Architectures* [BBC10], in which the Bus behaves as a shared memory between components. Messages are sustained by the bus and are periodically refreshed. Such communication, said *by sampling*, is non blocking. □

Example 3.16. The following code implements an unreliable medium, supporting message loss.

```
1  medium Lossy [receive rec_msg: t_msg, send snd_msg: t_msg] is
2    static var buf_msg: t_msg := empty_msg — Initially, no message is buffered
3    select
4    — message reception
5      when ?rec_msg  -> select
6                           buf_msg := rec_msg — message stored
7                        [] null               — message lost
8                        end select
9    [] — emission of the buffer content
10      when snd_msg  -> snd_msg := buf_msg
11    end select
12 end medium
```

□

Example 3.17. The following code implements a FIFO queue. The queue is encoded by using a static variable (line 2) of type *queue*, which is an array of messages. Initially the queue is empty. When a message is received on channel *rec_msg*, it is inserted in the queue by using a subblock *enqueue*, which returns the updated queue. Similarly, when a message has to be emitted on channel *snd_msg*, subblock *dequeue* returns the first message inserted and updates the queue.

```
1  medium FIFO [receive rec_msg:t_msg, send snd_msg:t_msg] is
2    static var queue : queue := queue (none)
3      select
4         when ?rec_msg -> enqueue (rec_msg, queue, ?queue)
5      [] when snd_msg  -> dequeue (queue,  ?queue, ?snd_msg)
6      end select
7  end medium
```

□

3.7 Systems

Blocks, environments, and mediums are composed inside systems. We first present the definition of GRL systems. Then, we discuss the expressiveness of GRL.

3.7.1 System definition

The syntax of systems is given by the grammar in Table 3.9. A system specification consists of formal parameters, temporary variables, component aliasing, and a behaviour described as a composition of components. Formal parameters are either constants, thus enabling parameterised specification, or without mode, to compose actual channels of components.

chan	::=	<$X_0, \ldots, X_n$>
	|	<_, ..., _>
	|	<**any** $T_0, \ldots,$ **any** T_n>
	|	?<$X_0, \ldots, X_n$>
	|	?<_, ..., _>
comp_alias	::=	**alias** B {*args*} **as** B'
	|	**alias** N {*args*} **as** N'
	|	**alias** M {*args*} **as** M'
block_invoc	::=	B' ($chan_0, \ldots, chan_m$)[$chan'_1, \ldots, chan'_n$]
env_invoc	::=	N' ($chan_0, \ldots, chan_m, B'_1, \ldots, B'_n$)
med_invoc	::=	M' [$chan_0, \ldots, chan_m$]
system	::=	**system** S {$vars_c$} ($vars_f$) **is** $comp\_alias_1, \ldots, comp\_alias_m$ **var** $vars_t$ **block list** $block\_invoc_0, \ldots, block\_invoc_n$ **environment list** $env\_invoc_0, \ldots, env\_invoc_p$ **medium list** $med\_invoc_0, \ldots, med\_invoc_q$ **end system**

Table 3.9: Syntax of GRL systems

Example 3.18. The following code describes the composition of the car park components. Details of the code will be given later.

```
1  system Main (— parameters observable by the outside world
2               Cmd_P11, Cmd_P12, Cmd_P21, Cmd_P22, Exit_P1, Exit_P2: bool,
3               — other parameters
4               ...
5              )
6  is
7    — component aliasing
8    alias Storey {true, false} as Storey1,
9          Storey {false, true} as Storey2,
10         Sampling as Exit_to_Storey1, Sampling as Exit_to_Storey2,
11         — other components
12         ...
13   — parameters non–observable by the outside world
14   var S_Out1, S_Out2: bool
15     — other parameters
16     ...
17   block list
18     Exit (<Exit_P1, Exit_P2>, ?Out_Open) [?S_Out1, ?S_Out2],
19     — other blocks
20     ...
21   environment list
22     Quasi_Synch_4 (Entrance, Exit, Storey1, Storey2),
23     Env_Storey    (?<Exit_P1, Exit_P2>)
24   medium list
25     Exit_to_Storey1 [S_Out1, ...],
26     Exit_to_Storey2 [S_Out2, ...],
27     — other mediums
28     ...
29 end system
```

□

Highest-level block instances are introduced by keywords **block list**. Actual parameters of highest-level blocks have the same form as those in subblock invocation inside components. Additional parameters, called *wildcards*, can be used as input and receive actual parameters. Wildcards have the form "**any** T" and match any value of type T. They are semantically equivalent to actual parameters that are declared as temporary variables, but not used for interactions with other components. Actual parameters are grouped to compose channels. In each channel, parameters should have the same form, i.e., all parameters are either variables (of the form "<X_0 , ... , X_n>" or "?<X_0 , ... , X_n>"), wildcards (of the form "<**any** T_0 , ... , **any** T_n>"), or unconnected (of the form "<_ , ... , _>" or "?<_ , ... , _>").

Environment and medium instances are introduced by keywords **environment list** and **medium list**, respectively. Their channels can be either tuples of variables or unconnected. If a channel is unconnected, the behaviour defined by its associated signal in the component definition is never executed. Actual activation parameters of environ-

ments should belong to the names of highest-level blocks. Note that if the activation of blocks in constrained, all blocks must have been already aliased. To prevent undesirable interferences that may occur when the activation of a block is constrained by several environments, actual activation parameters should be pairwise distinct in all environments.

Connections between components occur through channels. A channel may occur in exactly one pair of components. A block and an environment can be connected using a set of variables "X_0 , ... , X_n" by passing "<X_0 , ... , X_n>" as input channel to the block and "?<X_0 , ... , X_n>" as output channel to the environment, or conversely (Example 3.18, lines 18 and 23). If $n = 0$, angle brackets are optional. Connections between mediums and blocks on receive and send channels are carried out similarly (Example 3.18, lines 18, 25, and 26).

Alternatively, channels may occur in a single component. In this case, for input or receive channels, arbitrary values are assigned to parameters.

Channels whose parameters are declared as formal parameters of the system are observable by the system outside (Example 3.18, line 2), whereas channels whose parameters are declared as temporary variables are not (Example 3.18, line 14). Distinction between observable and non observable channels is a key device for abstraction, inspired by process algebra [Mil82] and is essential for verification.

Blocks cannot be directly connected to each other using channels. This ensures arbitrary interleaving between their activations. Environments and mediums cannot be connected to each other, neither. They are intended to be passive components that need to be triggered by blocks.

The behaviour of the system is defined as follows. A block can execute only if permitted by the environment constraining its activation. Such environment, if any, is unique according to GRL semantics. In this case, the block starts a step by triggering the components connected to its input and receive channels, to obtain values. After carrying out internal computations, the block finishes the step by triggering the components connected to its output and send channels, to deliver values. Following this execution model, a block interacts with a given component in at most two moments (i.e., causal events) during the same step. Accordingly, the data exchanged with each component at the same moment should be grouped in one single channel.

Consider Example 3.18. When block *Exit* starts a step, environment *Env_Storey* executes to provide the block inputs *Exit_P1* and *Exit_P2*. When the block finishes a step, both mediums *Exit_to_Storey1* and *Exit_to_Storey2* execute to consume the block outputs *S_Out1* and *S_Out2*, respectively. The combined execution of all components interacting during a block step is assumed to be instantaneous, thus preserving the zero-delay assumption of the block step.

3.7.2 Discussion and related work

This section summarises the GRL behavioural constructs and draws a comparison with some existing GALS approaches.

GRL blocks Depending on the call context, GRL blocks may behave either in a synchronous or in an asynchronous way. Inside components, subblocks behave as synchronous components that are composed synchronously with other subblocks. Inside systems, highest-level blocks behave as deterministic and atomic asynchronous components that interleave with other highest-level blocks. This allows a smooth integration of synchronous components inside an asynchronous context without requiring additional components to interface the synchronous world with the asynchronous one.

Contrarily to GRL blocks, the SAM-LNT approach [GT09] does not ensure the atomicity of LNT processes corresponding to synchronous components. Thus, inputs and outputs of different processes can interleave arbitrarily, leading to state explosion. In this respect, GRL enables a more concise representation than [GT09] (see Section 5.10 for a detailed comparison). In the Signal-Promela [DMK+06] approach, the atomicity of synchronous components is ensured by using the atomic construct of Promela. Both CRSM and SystemJ build upon the Esterel synchronous semantics, which make their synchronous part rich, compared to GRL. This is reasonable since they are design-oriented languages aiming to be used directly by users that are familiar with synchronous programming.

GRL mediums GRL mediums provide enough expressiveness to model general GALS systems. On the one hand, GRL does not fix any communication protocol, contrarily to some existing approaches. In the Signal-Promela approach [DMK+06], an LTTA protocol is used. Specific hardware communication buses are abstracted as Promela one-place FIFO channels. In the SAM-LNT approach [GT09], a TFTP protocol is used. FIFO and bag mediums with fixed size buffers and supporting message loss are used. On the other hand, GRL mediums support nondeterministic statements, which are absent in SystemJ [MGS12] and CRSM [Ram98]. Nondeterminism allows to describe useful situations such as message loss and reordering in a succinct way.

GRL environments Data constraints are similar to, but more general than *assertions* in synchronous languages. The latter are Boolean expressions (e.g., invariances and relations on inputs) that are assumed to always hold inside the synchronous program. In GRL, data constraints can express more complex behaviours, possibly combining inputs and outputs of several blocks, and depending on their history. Such constraints combined with nondeterminism allow the user to fine-tune the system description and reduce the size of generated state spaces. We have found no equivalent to GRL data constraints in the Signal-Promela approach nor in the SAM-LNT approach.

Activation constraints make the degree of asynchrony between concurrent highest-level blocks controllable. On the one hand, they enable an accurate and abstract description of realistic situations such as failure and activation strategies. On the other hand, they participate to reduce the size of the system state space. Again, no equivalent to GRL activation constraints has been mentioned either in SAM-LNT or in CRSM and SystemJ. In the SAM-LNT approach, this causes a maximal degree of asynchrony leading to state space explosion. In the Signal-Promela approach, similar constraints, called *clock constraints*, are automatically generated from Signal. When clock constraints are not met, a stuttering step is executed leading to no change in input and output signals.

GRL systems Factoring a GALS description into blocks, environments, and mediums enhances the modularity and reduces the complexity of modelling. At early design stages, when system specifications are evolving frequently, one might well want to verify the behaviour of blocks separately or a primary system behaviour with simple communication mediums and test case scenarios. Afterward, the system behaviour can be refined to meet detailed communication schemes and constraints. This helps the user to make explicit her intention on how to precisely design the system behaviour. Quoting Dijkstra [Dij82] about *the role of scientific thought*: "*The separation of concerns is yet the only available technique for effective ordering of one's thoughts. This is focusing one's attention upon some aspect.*".

Chapter 4

Formal Dynamic Semantics of GRL

GRL is endowed with formal semantics, giving a rigorous specification of how correct programs behave. This improves our understanding of the finest details of the language and helps in tool construction. The dynamic semantics of GRL are defined in an operational style. The meaning of program phrases is defined in terms of their computation steps during the program execution and the possible state transformation they perform. In this chapter, we first introduce the basic notions needed to specify the operational semantics of GRL. We then present the semantics of GRL constructs, stressing on the behavioural ones.

> It is not what you meant to say, but it is what your saying meant.
>
> Walter M. Miller Jr

4.1 Preliminaries

Programs are assumed to have successfully passed all static analysis phases, including parsing, binding analysis, typing analysis, and variable initialisation analysis. In particular, we make the following assumptions:

- Each variable is assigned a distinct name. This prevents *variable shadowing* to occur, i.e., a variable declared within a certain scope with the same name as a variable declared in an outer scope.
- Each component instance is assigned a distinct name. Thus, those names can be used when associating a separate internal state to each component instance. For simplicity, we assume that each component instance called in some context has been aliased earlier in the same call context.

In this section, we first present the main concepts used to define the formal semantics of GRL. These are *stores*, *stacks*, and *memories*. Second, we sketch how GRL programs can be interpreted in terms of LTSs (*Labelled Transition Systems*). Finally, we introduce auxiliary functions required to formally define those LTSs.

4.1.1 Stores

We define a *store*, written ρ, as a *partial* function from variables to values. For a store ρ mapping each variable X_i to the corresponding value e_i, where $i \in 1..n$, we write $\rho = [X_1 \leftarrow e_1, \ ..., \ X_n \leftarrow e_n]$ and $\rho\ (X_1) = e_1, \ \ldots, \ \rho\ (X_n) = e_n$.

We define the *domain* of store ρ, written $dom\ (\rho)$, as $\{X_1, \ \ldots, \ X_n\}$. In particular, we write "[ ]" for the empty store and $dom\ ([\ ]) = \{\ \}$ for its domain. For a subset $\{Y_1, \ldots, Y_p\} \subseteq dom\ (\rho)$, we write $\rho_{|\{Y_1, \ ..., \ Y_p\}}$ for the restriction of ρ to $\{Y_1, \ \ldots, \ Y_p\}$ defined by $[Y_1 \leftarrow \rho(Y_1), ..., Y_p \leftarrow \rho(Y_p)]$.

We define the *update* of a store ρ_1 with a store ρ_2, written $\rho_1 \oplus \rho_2$, as follows:

$$(\rho_1 \oplus \rho_2)(X) \quad = \quad \begin{cases} \rho_2\ (X) & \text{if } X \in dom(\rho_2) \\ \rho_1\ (X) & \text{if } X \notin dom(\rho_2) \text{ and } X \in dom(\rho_1) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The notation $\bigoplus_{i \in 1..n} \rho_i$ stands for the sum $\rho_1 \oplus \ldots \oplus \rho_n$ (note that $\oplus$ is an associative but not commutative operator).

Example 4.1. Here are some examples of store update.

$$\begin{array}{lcl} [X \leftarrow 0] \oplus [\ ] & = & [X \leftarrow 0] \\ [X \leftarrow 0] \oplus [Y \leftarrow 1] & = & [X \leftarrow 0, \ Y \leftarrow 1] \\ [X \leftarrow 0, \ Y \leftarrow 1] \oplus [X \leftarrow 1, \ Y \leftarrow 1] & = & [X \leftarrow 1, \ Y \leftarrow 1] \\ [X \leftarrow 0, \ Y \leftarrow 1] \oplus [Y \leftarrow 0, \ Z \leftarrow 1] & = & [X \leftarrow 0, \ Y \leftarrow 0, \ Z \leftarrow 1] \end{array}$$

□

For sets of stores, we define the *update* of a set S_1 with a set S_2, written $S_1 \oplus S_2$, as follows:

$$S_1 \oplus S_2 \ = \ \{\rho_1 \oplus \rho_2 \mid \rho_1 \in S_1 \ \wedge \ \rho_2 \in S_2\}$$

4.1.2 Stacks

We define a *stack*, written σ, as a sequence of component instance identifiers. A stack is defined recursively, either as the empty sequence ϵ or as a non empty sequence of the form "$\sigma'.id$" where σ' is a stack and id is the name of a component instance pushed on top of the stack. We write $\sigma_1.\sigma_2$ for the concatenation of stacks σ_1 and σ_2, defined as:

$$\begin{array}{lcl} \sigma_1.\epsilon & \triangleq & \sigma_1 \\ \sigma_1.(\sigma_2.\text{id}) & \triangleq & (\sigma_1.\sigma_2).\text{id} \end{array}$$

where symbol $\triangleq$ means *equal by definition*.

We define function *prefix* which indicates whether a stack σ_1 is a prefix of a stack σ_2 as follows:

$$\mathit{prefix}\ (\sigma_1,\ \sigma_2)\ \triangleq\ \exists\ \sigma',\ \sigma_1.\sigma' = \sigma_2$$

During program execution, a unique stack is associated to each component instance. This stack is the ordered sequence of the names of all its enclosing component instances, starting from the highest-level component (block, medium, or environment), down to the current component, transitively. GRL stacks are similar to call stacks in ordinary programming languages, except that they only contain component instance identifiers. Stacks are finite and statically bounded, since recursion is forbidden in GRL.

Example 4.2. Let "**alias** B **as** B'" be the aliasing of a highest-level block inside a system S. The stack of B' is $\epsilon.B'$. Now, let "**alias** B **as** B'" be a subblock aliasing inside a medium M and let "**alias** M **as** M'" be the medium aliasing inside the system S. The stack of M' is $\epsilon.M'$ and that of B' in this call context is "$\epsilon.M'.B'$". □
In the sequel, we omit the initial ϵ in non empty stacks. For example, we write "$M'.B'$" instead of "$\epsilon.M'.B'$".

4.1.3 Memories

We define a *memory*, written μ, as a partial function from stacks to stores. Memories implement the internal state of components. For a memory μ mapping stacks σ_i to stores ρ_i, where $i \in 1..n$, we write $\mu = [\sigma_1 \leftarrow \rho_1,\ \ldots,\ \sigma_n \leftarrow \rho_n]$ and $\mu\ (\sigma_1) = \rho_1,\ \ldots,\ \mu\ (\sigma_n) = \rho_n$. Relation $\sigma_i \leftarrow \rho_i$ means that ρ_i defines the internal state of the component instance whose stack is σ_i. We define the domain of memory μ, written $dom(\mu)$, as $\{\sigma_1, \ldots, \sigma_n\}$.

Similarly to store update, we define the *update* of a memory μ_1 with a memory μ_2, written $\mu_1 \oplus \mu_2$ as follows:

$$(\mu_1 \oplus \mu_2)(\sigma) \quad = \quad \begin{cases} \mu_2(\sigma) & \text{if } \sigma \in dom(\mu_2) \\ \mu_1(\sigma) & \text{if } \sigma \notin dom(\mu_2) \text{ and } \sigma \in dom(\mu_1) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The notation $\bigoplus_{i \in 1..n} \mu_i$ stands for the sum $\mu_1 \oplus \ldots \oplus \mu_n$ (note that $\oplus$ is an associative but not commutative operator).

We define a function *mem* which extracts from a memory μ the submemories corresponding to the component whose stack is σ and those of its subblocks.

$$\mathit{mem}\ (\mu,\ \sigma) = \bigoplus_{\substack{\sigma' \in dom(\mu) \\ \wedge\ \mathit{prefix}\ (\sigma,\ \sigma')}} [\sigma' \leftarrow \mu(\sigma')]$$

Example 4.3. Block *C_Shift* (Ex. 3.3, page 40) defines no static variables and no subblocks. The memories of its instances are the empty memory [ ].

Block *B_Edge* (Ex. 3.2, page 39) has one static variable *Pre_Signal*. The block is invoked twice inside Block *Exit* (Ex. 3.7, page 42). Let call *B_Edge_1* and *B_Edge_2* the names associated to the subblocks. The respective memories of the instances are initially:

$$\begin{array}{lcl} \mu_{B\_Edge\_1} & = & [Exit\ .\ B\_Edge\_1 \leftarrow [Pre\_Signal \leftarrow false]] \\ \mu_{B\_Edge\_2} & = & [Exit\ .\ B\_Edge\_2 \leftarrow [Pre\_Signal \leftarrow false]] \end{array}$$

Block *Exit* (Ex. 3.7, page 42) has no static variables. The memory of each of its instances is composed of the memories of its subblocks. In addition to subblocks *B_Edge_1* and *B_Edge_2*, block *Exit* invokes instances of logical blocks *B_And* and *B_Or*. Such logical blocks neither have internal state nor invoke subblocks. Hence, each instance of block *Exit* is associated to a memory $\mu_{B\_Edge\_1} \oplus \mu_{B\_Edge\_2}$. □

4.1.4 LTSs of GRL systems

The behaviour of a GRL system is formally defined in terms of an LTS. States contain the sum of all memories of highest-level components, the initial state being the empty memory. Transitions between states denote block steps. With respect to a state μ, a step of some highest-level block B' leads to a transition of the form:

$$\mu \xrightarrow{B'\ (ch_1,\ldots,ch_m)[ch'_1,\ldots,ch'_n]} \mu'$$

where:

- μ' updates μ with the values of variables composing the memories of both B' and the components connected to B'.
- $ch_1, \ldots, ch_m$ map the values of actual *input* and *output* parameters to their respective formal parameters. Each ch_i ($i \in 1..m$) is composed of a set of elements. Each element has the form "$X = e$", where X is an actual parameter and e its value in the current step, if X is observable. Otherwise, the element has the form "$\_$".
- $ch'_1, \ldots, ch'_n$ are defined similarly to $ch_1, \ldots, ch_m$, but for *receive* and *send* parameters.

For example, the following transition corresponds to a step of block *Exit*:

$$\mu_0 \xrightarrow{\text{Exit (Cmd\_P1 = } \textbf{true}\text{, Cmd\_P2 = } \textbf{false}\text{, Open = } \textbf{true}\text{) [Out\_P1 = } \textbf{true}\text{, Out\_P2 = } \textbf{false}\text{]} \mu_1$$

More concretely, we are concerned with the behaviour of blocks as an external observer would see it: which block is executing and what are the values carried by its inputs and outputs. Since block steps are atomic, each step results in exactly one transition in the LTS. From one block step to another, the values of static variables composing its internal state are stored in the system state. The system state contains also the values of static variables composing the internal states of environments and mediums triggered by the block step.

4.1.5 Structural Operational Semantics (SOS)

The LTSs of GRL are formally defined by using the method advocated by Plotkin [Plo81], in which evaluation and execution relations are specified by *transition rules* in a syntax-directed way. A transition rule has a set of zero (in which case the rule is called *axiom*) or more *premises* and a *conclusion*. It is commonly written as follows:

$$\frac{\text{premises}}{\text{conclusion}}$$

The validity of all the premises implies the validity of the conclusion. Transition rules will be applied in derivations, such that the facts below the solid line are derived from the ones above.

4.2 Expressions

We define the evaluation of an expression E in a store ρ as a relation of the form $\{E\}\ \rho\ \rightarrow_{\mathtt{e}}\ e$, where e is the resulting value. An excerpt of transition rules of GRL expressions are given in Table 4.1. A literal constant K is always evaluated with itself as value (rule R1). A variable X evaluates to the value recorded in the current store (rule R2). Both rules are axioms. For example, an application of rule R2 is:

$$\frac{}{\{X\}\ [X \leftarrow 1] \rightarrow_{\mathtt{e}} 1}$$

The evaluation result of the expression "$F(E_0, \ldots, E_n)$" is the returned value of calling the predefined function F with the values of $E_0, \ldots, E_n$ (rule R3). Record field access and array element access are considered as predefined functions. Predefined functions are standard; we do not provide here their formal semantics.

(R1)	$\dfrac{}{\{K\}\ \rho \rightarrow_{\mathtt{e}} K}$
(R2)	$\dfrac{}{\{X\}\ \rho \rightarrow_{\mathtt{e}} \rho\ (X)}$
(R3)	$\dfrac{(\forall\ i\ \in 0..n)\ \{E_i\}\ \rho \rightarrow_{\mathtt{e}} e_i}{\{F\ (E_0, \ldots, E_n)\}\ \rho \rightarrow_{\mathtt{e}} F(e_0, \ldots, e_n)}$

Table 4.1: Sos rules of GRL expressions

4.3 Statements

The semantics of statements is defined by means of couples (store, memory). Stores serve to read and update local variables and parameters of the current component. Memories serve to read and update the internal states of *subblocks*, whose invocation is part of statements. To access those memories, the stack of the *current component*, inside which the statement executes, is required. In the sequel, we call *current store* and *current memory* the store and memory in which a statement will execute.

We define the execution of statements as a relation of the form $\{I\}\ \sigma,\ \rho,\ \mu \xrightarrow{\ell}_{\mathsf{i}}\ \rho',\ \mu'$ where:

- I is the statement to execute
- σ is the stack of the current component instance inside which I executes
- ρ is the store defining parameters and variables of the current component
- μ is the memory defining the internal state of both the current component and its subblocks
- ρ' and μ' are the respective updates of ρ and μ with the computations performed by I
- ℓ is a label having one of the following forms:

label	meaning
ϵ	I terminates without encountering any signal
B_0	I terminates and has encountered an activation signal "**enable** B_0"
$?\langle X_1, \ldots, X_n\rangle$	I terminates and has encountered a data signal "**when ?<**X_1 **,** $\ldots$ **,** X_n**>**"
$\langle X_1, \ldots, X_n\rangle$	I terminates and has encountered a data signal "**when <**X_1 **,** $\ldots$ **,** X_n**>**"

In the sequel, we present some representative semantic rules of statement execution.

4.3.1 Basic statements

Table 4.2 gives the semantics rules of a subset of GRL statements.

(R4)	$\dfrac{\{E\}\ \rho \rightarrow_{\mathsf{e}} e}{\{X\ \texttt{:=}\ E\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho \oplus [X \leftarrow e], \mu}$
(R5)	$\dfrac{e \in T \qquad \{E\}\ \rho \oplus [X \leftarrow e] \rightarrow_{\mathsf{e}} \textbf{true}}{\{X\ \texttt{:=}\ \textbf{any}\ T\ \textbf{where}\ E\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\mathsf{i}} \rho \oplus [X \leftarrow e], \mu}$
(R6)	$\dfrac{\{I_1\}\ \sigma, \rho, \mu \xrightarrow{\ell_1}_{\mathsf{i}} \rho', \mu' \qquad \{I_2\}\ \sigma, \rho', \mu' \xrightarrow{\ell_2}_{\mathsf{i}} \rho'', \mu''}{\{I_1\,;I_2\}\ \sigma, \rho, \mu \xrightarrow{\ell_1+\ell_2}_{\mathsf{i}} \rho'', \mu''}$
(R7)	$\dfrac{(\exists k \in 1..n) \quad \{I_k\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}}\ \rho', \mu'}{\{\textbf{select}\ I_1\ \texttt{[]} \ldots \texttt{[]}\, I_n\ \textbf{end select}\}\ \sigma, \rho, \mu \xrightarrow{\ell}_{\mathsf{i}} \rho', \mu'}$

Table 4.2: Sos rules of statements (Excerpts)

Rule *R4* defines the semantics of deterministic assignment statement. An assignment statement terminates normally by assigning the value of expression E (right-hand side) in the current store to variable X (left-hand side). Note that this rule updates the store but not the memory even if X was defined as a static variable. Note also that the current memory μ is not used to evaluate the expression because store ρ is assumed to already contain a copy of the static variables that are local to the current component. Construction of store ρ and memory update are handled at the level of component invocation.

Rule *R5* defines the semantics of nondeterministic assignment statement. A nondeterministic assignment terminates normally after updating the store by assigning an arbitrary value of type T to variable X, provided condition E of the assignment evaluates to true in the updated store.

Rule *R6* defines the semantics of a sequential composition of two statements I_1 and I_2. The sequential composition starts by executing statement I_1 and updating the current store and memory. Then, I_2 is executed in the store and memory updated by I_1. Symbol + denotes label concatenation; ϵ is the identity element, i.e., $\epsilon + \ell = \ell + \epsilon = \ell$ for every label ℓ. At least one of the labels ℓ_1 and ℓ_2 must be equal to ϵ, since sequential composition between signal statements is forbidden (see Section 3.5.1, page 44).

Rule *R7* defines the semantics of a nondeterministic choice between $I_1, \ldots, I_n$. Nondeterministic choice terminates normally after behaving either as I_1, ..., or as I_n. In both *R6* and *R7*, note that memories μ' and μ'' may differ from memories μ and μ', respectively, if (and only if) statement I_1, I_2, or I_k invokes subblocks.

Example 4.4. To illustrate the derivation of transition rules, we consider the execution of the GRL statements below in a stack ϵ, store [ ], and memory [ ].

```
1  X := any bool; — statement 1
2  Y := not (X)   — statement 2
```

Inspecting rule *R6* (see Table 4.2), statement 1 should be executed before statement 2. The statement execution starts by assigning an arbitrary value of Boolean type to X (rule *R5*, Table 4.2). There are two possible rules, each corresponding to a value of the Boolean type. Only the rule corresponding to value *false* is presented here, for conciseness. The execution continues by assigning a value to Y (rule *R4*, Table 4.2), which depends on the evaluation of variable X. Here is the derivation of transition rules, where *ff* and *tt* are shorthands for Boolean values *false* and *true*.

$$\dfrac{\dfrac{}{ff \in bool} \quad \dfrac{\dfrac{}{\{X\}\ [X \leftarrow ff]\ \rightarrow_{\mathsf{e}}\ ff}}{\{\textbf{not}\ (X)\}\ [X \leftarrow ff]\ \rightarrow_{\mathsf{e}}\ tt}}{\{X\ \textbf{:=}\ \textbf{any bool}\}\ \epsilon,\ [],\ []\ \xrightarrow{\epsilon}_{\mathsf{i}}\ [X \leftarrow ff],\ []} \quad \dfrac{}{\{Y\ \textbf{:=}\ \textbf{not}\ (X)\}\ \epsilon,\ [X \leftarrow ff],\ []\ \xrightarrow{\epsilon}_{\mathsf{i}}\ [X \leftarrow ff,\ Y \leftarrow tt],\ []}}{\{\ X\ \textbf{:=}\ \textbf{any bool;}\ Y\ \textbf{:=}\ \textbf{not}\ (X)\ \}\ \epsilon,\ [],\ []\ \xrightarrow{\epsilon}_{\mathsf{i}}\ [X \leftarrow ff,\ Y \leftarrow tt],\ []}$$

□

4.3.2 Signals

The semantics of signals, given in Table 4.3, is inspired by the semantics of communication actions in process algebra.

(R8)	$\dfrac{\{I_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{\epsilon} \rho',\ \mu'}{\{\textbf{when}\ \texttt{<}X_1\texttt{,}\ \ldots\ \texttt{,}X_n\texttt{>}\ \texttt{->}\ I_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{\langle X_1,\ldots,X_n\rangle}_{\mathrm{i}} \rho',\ \mu'}$
(R9)	$\dfrac{\{I_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{\epsilon} \rho',\ \mu'}{\{\textbf{when}\ \texttt{?<}X_1\texttt{,}\ \ldots\ \texttt{,}X_n\texttt{>}\ \texttt{->}\ I_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{?\langle X_1,\ldots,X_n\rangle}_{\mathrm{i}} \rho',\ \mu'}$
(R10)	$\dfrac{}{\{\textbf{enable}\ B_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{B_0}_{\mathrm{i}} \rho, \mu}$

Table 4.3: Sos rules of signals

Rules *R8* and *R9* state that a signal, after executing the statement I_0 in the current store and memory, terminates normally by producing updated store and memory and by passing a label to the context. According to static semantics, I_0 does not contain a signal statement, since nested signals are forbidden.

More concretely, the data signal "**when** `?<`X_1`,` $\ldots$ `,`X_n`>`" is first prepared to receive values for variables $X_1, \ldots, X_n$ from a block. The execution of the signal is then contingent on whether a connected block sends those values over the channel corresponding to the signal. To express such contingency, we label the transition defining the signal execution by "`?<`X_1`,` $\ldots$ `,`X_n`>`" (rule *R9*). By construction, the values of variables $X_1, \ldots, X_n$ are available in the current store, when the signal executes.

Similarly, the semantics of the data signal "**when** `<`X_1`,` $\ldots$ `,`X_n`>`" is prepared to send values on variables $X_1, \ldots, X_n$ to a block. The values of variables $X_1, \ldots, X_n$ are assumed to be assigned inside I_0. The transition defining the signal execution is labelled by "`<`X_1`,` $\ldots$ `,`X_n`>`" (rule *R8*).

An activation signal "**enable** B_0" encodes the permission of a block to execute. The semantics of activation signals is defined by an axiom (rule *R10*). The execution of an activation signal yields a transition labelled by B_0, i.e., the activation formal parameter corresponding to the block meant to be activated. The current store and memory remain unchanged.

4.4 Store construction at component invocation

This section is meant for future reference when writing down the transition rules of component invocation. We first present some auxiliary functions. Then, we present the way the *global store*, containing all global constants, is constructed. Finally, we present the way the current store and memory are updated at components invocation.

4.4.1 Auxiliary functions

We define a set of auxiliary functions.

Variable list Given a variable declaration list *vars* or an actual channel *chan*, function *get_vars* returns the ordered list of variable identifiers. For unconnected channels, the symbol ϵ denotes the empty list.

$$
\begin{array}{lcl}
\text{get\_vars}\ (vars_1, \ldots, vars_n) & = & \text{get\_vars}\ (vars_1)\texttt{++}\ldots\texttt{++}\text{get\_vars}\ (vars_n) \\
\text{get\_vars}\ (X_1\texttt{,}\ldots\texttt{,}X_m\texttt{:}type) & = & \langle X_1, \ldots, X_m\rangle \\
\text{get\_vars}\ (X_1\texttt{,}\ldots\texttt{,}X_m\texttt{:}type\ \texttt{:=}\ E) & = & \langle X_1, \ldots, X_m\rangle \\
\text{get\_vars}\ (chan_1, \ldots, chan_n) & = & \text{get\_vars}\ (chan_1)\texttt{++}\ldots\texttt{++}\text{get\_vars}\ (chan_n) \\
\text{get\_vars}\ (\texttt{<}X_1\texttt{,}\ldots\texttt{,}X_m\texttt{>}) & = & \langle X_1, \ldots, X_m\rangle \\
\text{get\_vars}\ (\texttt{?<}X_1\texttt{,}\ldots\texttt{,}X_m\texttt{>}) & = & \langle X_1, \ldots, X_m\rangle \\
\text{get\_vars}\ (\texttt{<\_,}\ldots\texttt{,\_>}) & = & \epsilon \\
\text{get\_vars}\ (\texttt{?<\_,}\ldots\texttt{,\_>}) & = & \epsilon \\
\text{get\_vars}\ (\texttt{<any}\ type_1\texttt{,}\ldots\texttt{,any}\ type_m\texttt{>}) & = & \epsilon
\end{array}
$$

Type list Given a variable declaration list, function *get_types* returns the ordered list of types with which variables are declared.

$$
\begin{array}{lcl}
\text{get\_types}\ (vars_1, \ldots, vars_n) & = & \text{get\_types}\ (vars_1)\texttt{++}\ldots\texttt{++}\text{get\_types}\ (vars_n) \\
\text{get\_types}\ (X_1\texttt{,}\ldots\texttt{,}X_m\texttt{:}type) & = & \langle \underbrace{type, \ldots, type}_{m}\rangle \\
\text{get\_types}\ (X_1\texttt{,}\ldots\texttt{,}X_m\texttt{:}type\ \texttt{:=}\ E) & = & \langle \underbrace{type, \ldots, type}_{m}\rangle
\end{array}
$$

Initialisation Function *init* assigns to parameters (resp. variables) their default values (resp. initialisation values). Given a variable declaration list *vars* and a store ρ, function *init* returns a store assigning to each variable in *vars* the evaluation of its default expression in store ρ.

$$
\begin{array}{lcl}
\text{init}\ (\langle vars_1, \ldots, vars_n\rangle, \rho) & = & \text{init}\ (vars_1, \rho) \oplus \ldots \oplus \text{init}\ (vars_n, \rho) \\
\text{init}\ (X_1\texttt{,}\ldots\texttt{,}X_m\texttt{:}type, \rho) & = & [] \\
\text{init}\ (X_1\texttt{,}\ldots\texttt{,}X_m\texttt{:}type\ \texttt{:=}\ E, \rho) & = & [X_1 \leftarrow e, ..., X_m \leftarrow e] \text{ where } \{E\}\ \rho \rightarrow_{\texttt{e}} e
\end{array}
$$

Example 4.5. Consider the following GRL code excerpt.

```
1  const C: nat := 3 — global constant declaration
2  block ... is
3     var X1, X2: nat := C + 1  — variable declaration
4     ...
5  end block
```

By applying functions *get_vars*, *get_types*, and *init*, we have:

$\text{get\_vars}\ (C\,\texttt{:nat :=}\ 3) = \langle C \rangle$ $\quad$ $\text{get\_types}\ (C\,\texttt{:nat :=}\ 3) = \langle \textbf{nat} \rangle$
$\text{get\_vars}\ (X1\,\texttt{,}\,X2\,\texttt{:nat :=}\ C\ \texttt{+}\ 1) = \langle X1,\ X2 \rangle$ $\quad$ $\text{get\_types}\ (X1\,\texttt{,}\,X2\,\texttt{:nat :=}\ C\ \texttt{+}\ 1) = \langle \textbf{nat},\ \textbf{nat} \rangle$

$\text{init}\ (X1\,\texttt{,}\ X2\,\texttt{:nat :=}\ C\ \texttt{+}\ 1, [C \leftarrow 3]) = [X1 \leftarrow 4,\ X2 \leftarrow 4]$

□

In the sequel, we will define and use functions on lists of actual parameters (non-terminal *args* in Table 3.4, page 36) and on actual channels (non-terminal *chan* in Table 3.9, page 51). We introduce a non-terminal *acts* to denote either *args* or *chan*, when such a distinction is unnecessary.

Some functions concern either only constant, input, and receive parameters or only output and send parameters. We will write $acts_{in}$ as shorthand for constant, input, or receive actual parameters and $vars_{in}$ for the corresponding formal parameter list. Hence, $acts_{in}$ has one of the following forms:

- "arg_1 `,` ... `,` arg_n", where each parameter arg_i is either a variable or an underscore.
- "`<`arg_1 `,` ... `,` arg_n`>`", where parameters are either all variables, all underscores, or all wildcards.

Similarly, we will write $acts_{out}$ as shorthand for output and send actual parameters and $vars_{out}$ for the corresponding formal parameter list. Hence, $acts_{out}$ has one of the following forms:

- "arg_1 `,` ... `,` arg_n", where each parameter arg_i has either the form "`?`X" or "`?_`".
- "`?<`arg_1 `,` ... `,` arg_n`>`", where parameters are either all variables or all underscores.

Formal parameter assignment We define two functions *assign* and *init_assign*. Function *assign* copies the values of actual parameters into their respective formal parameters in the component definition. Only constant, input, and receive parameters are concerned. Actual parameters are assumed to be assigned values in a store ρ. Function *assign* returns a set of stores assigning to each formal parameter the evaluation of its corresponding actual parameter in ρ. When "**any** *type*" is used as actual parameter, the corresponding formal parameter is assigned an arbitrary value of type *type*. Because of this nondeterminism, there is no unique store. Note that the function is well-formed because operator $\oplus$ is defined on sets of stores (see page 57).

$$
\begin{array}{ll}
\text{assign}\ (\texttt{<}arg_1\,\texttt{,}\,\ldots\,\texttt{,}\,arg_n\texttt{>}, \langle X_1, \ldots, X_n \rangle, \rho) & = \text{assign}\ (arg_1, X_1, \rho) \oplus\ \ldots \oplus \text{assign}\ (arg_n, X_n, \rho) \\
\text{assign}\ (\langle arg_1, \ldots, arg_n \rangle, \langle X_1, \ldots, X_n \rangle, \rho) & = \text{assign}\ (arg_1, X_1, \rho) \oplus\ \ldots \oplus \text{assign}\ (arg_n, X_n, \rho) \\
\text{assign}\ (\_, X, \rho) & = \{[]\} \\
\text{assign}\ (E, X, \rho) & = \{[X \leftarrow e] \mid var(E) \subseteq dom(\rho)\ \wedge\ \{E\}\ \rho\ \rightarrow_{\mathtt{e}}\ e\} \\
\text{assign}\ (\textbf{any}\ type, X, \rho) & = \{[X \leftarrow e] \mid e \in T\}
\end{array}
$$

where function *var* returns the set of variables in an expression

Example 4.6. Here are some applications of function *assign*.

$$\begin{array}{lllll} \text{assign } (\_, & W, & [a \leftarrow 1, b \leftarrow 0]) & = & \{[]\} \\ \text{assign } (a, & X, & [a \leftarrow 1, b \leftarrow 0]) & = & \{[X \leftarrow 1]\} \\ \text{assign } (a\text{+1}, & Y, & [a \leftarrow 1, b \leftarrow 0]) & = & \{[Y \leftarrow 2]\} \\ \text{assign } (\textbf{any bool}, & Z, & [a \leftarrow 1, b \leftarrow 0]) & = & \{[Z \leftarrow false], [Z \leftarrow true]\} \end{array}$$

□

Function *init_assign* builds upon functions *init* and *assign* and is intended to prepare the store in which a component body will execute. It assigns values to formal parameters. First, to each formal parameter is assigned its default value in a store ρ, using function *init*. Then, the value of each formal parameter is updated with the value of its corresponding actual parameter in a store ρ', using function *assign*. According to function *assign*, the formal parameters having "_" as corresponding actual parameter are not updated.

Sometimes, input and receive actual parameters may be unavailable at invocation time, e.g., an environment triggered on an output signal. In such case, the function returns the empty store.

$$\text{init\_assign } (\underset{k\in 1..m}{\text{++}} acts_{in_k}, \underset{k\in 1..m}{\text{++}} vars_{in_k}, \rho, \rho') = \begin{cases} \bigoplus_{k\in 1..m} \text{init } (vars_{in_k}, \rho) \oplus \bigoplus_{k\in 1..m} \text{assign } (acts_{in_k}, \text{get\_vars}(vars_{in_k}), \rho') & \text{if } \underset{k\in 1..m}{\text{++}} acts_{in_k} \neq \epsilon \\ [] & \text{otherwise} \end{cases}$$

Example 4.7. Consider the following GRL code excerpt.

```
1  block isequal (in X1, X3: nat := 0, out Y: bool) is
2    Y := (X1 == X2)
3  end block
4
5  block ... is
6    a := 3;
7    isequal (_, a, ?b)
8  end block
```

By applying function *init_assign* in stores $\rho = []$ and $\rho' = [a \leftarrow 3]$, we have:

$$\begin{array}{l} \text{init\_assign } (\langle \_, a \rangle, \ X1\text{ , }X2\text{:}\textbf{nat}\text{ := }0, \ [], \ [a \leftarrow 3]) \\ \quad = \text{ init } (X1\text{ , }X2\text{:}\textbf{nat}\text{ := }0, \ []) \ \oplus \ \text{assign } (\langle \_, a \rangle, \ \langle X1\text{ , }X2 \rangle, \ [a \leftarrow 3]) \\ \quad = \ [X1 \leftarrow 0, \ X2 \leftarrow 0] \ \oplus \ [X2 \leftarrow 3] \\ \quad = \ [X1 \leftarrow 0, \ X2 \leftarrow 3] \end{array}$$

□

Actual parameter update Function *update* allows the values of output and send formal parameters to be copied back into the actual parameters at the end of component invocation. Formal parameters are assumed to be assigned values in a store ρ. Function *update* returns a store assigning to each actual parameter the evaluation of its corresponding formal parameter in store ρ.

Sometimes, output and send actual parameters may be unavailable at invocation time, e.g., an environment triggered on an input signal. In such case, the function returns the empty store.

$$\text{update}\ (\underset{k\in 1..n}{\texttt{++}}\ acts_{out_k},\ \underset{k\in 1..n}{\texttt{++}}\ vars_{out_k},\ \rho) \quad = \quad \begin{cases} \bigoplus_{k\in 1..n} \text{update}\ (acts_{out_k}, vars_{out_k}, \rho) & \text{if } \underset{k\in 1..n}{\texttt{++}}\ acts_{out_k} \neq \epsilon \\ [] & \text{otherwise} \end{cases}$$

$$\begin{array}{lcl} \text{update}\ (\texttt{?<}arg_1, \ldots, arg_n\texttt{>}, \langle X_1, \ldots, X_n\rangle, \rho) & = & \text{update}\ (\texttt{?}arg_1, X_1, \rho) \oplus \ \ldots \oplus \text{update}\ (\texttt{?}arg_n, X_n, \rho) \\ \text{update}\ (\langle arg_1, \ldots, arg_n\rangle, \langle X_1, \ldots, X_n\rangle, \rho) & = & \text{update}\ (arg_1, X_1, \rho) \ \ \oplus \ \ldots \oplus \text{update}\ (arg_n, X_n, \rho) \\ \text{update}\ (\texttt{?}\,Y, X, \rho) & = & \begin{cases} [Y \leftarrow \rho(X)] & \text{if } X \in dom(\rho) \\ [] & \text{otherwise} \end{cases} \\ \text{update}\ (\texttt{?\_}, X, \rho) & = & [] \end{array}$$

Example 4.8. Consider the GRL code excerpt of Example 4.7. Variable Y evaluates to *false* in store $[X1 \leftarrow 0,\ X2 \leftarrow 3]$. Hence, by applying function *update* we have:

$$\text{update}\ (\langle \texttt{?}b\rangle,\ Y\texttt{: bool},\ [Y \leftarrow false]) \ = \ [b \leftarrow false]$$

□

Memory access As advanced in Section 4.3, computations on local variables, including static ones, are performed in the current store. An intermediate store is then needed, assigning values to static variables of the current component. This is done by function *static*. The function returns a store assigning to each variable its corresponding value in the current memory; if not such value exists, the variable is assigned the value of its default expression.

$$\text{static}\ (vars,\ \sigma,\ \rho,\ \mu) \quad = \quad \begin{cases} \mu\ (\sigma) & \text{if } \sigma \in dom\ (\mu) \\ \text{init}\ (vars,\ \rho) & \text{otherwise} \end{cases}$$

Example 4.9. Consider the following GRL code excerpt.

```
1  block Mem ... is
2    static var X:nat := 0
3      ... — X not updated here
4      X := X + 1
5  end block
```

Assume the block is used as highest-level component ($\sigma = \epsilon$) with no other component. By applying function *static* in the first and second steps of block *Mem*, we have:

$$\begin{array}{lcl} \text{static } (X\text{:}\mathbf{nat} := 0,\ Mem,\ [],\ []) & = & [X \leftarrow 0] \\ \text{static } (X\text{:}\mathbf{nat} := 0,\ Mem,\ [],\ [Mem \leftarrow [X \leftarrow 1]]) & = & [X \leftarrow 1] \end{array}$$

□

4.4.2 Global store

Global constants can be used in the current module and its importing modules. Their values should be available in the stores in which components are invoked. For this purpose, a *global store* is constructed, assigning to each constant the value of its expression.

After binding analysis, global constants are assumed to be ordered according to their dependencies, cyclic dependencies being forbidden. Suppose $X_1, \ldots, X_n$ is the ordered set of global constants defined respectively with expressions $E_1, \ldots, E_n$, such that:

$$\begin{cases} \text{var } (E_1) = \emptyset \\ (\forall i \in 2..n) \quad \text{var } (E_i) \subseteq \{X_1,\ \ldots,\ X_{i-1}\} \end{cases}$$

In such case, E_1 is a literal constant. The global store, written ρ_{glob}, is constructed by assigning to each global constant X_i the value of its expression E_i in store ρ_{i-1}, as follows:

$$\begin{array}{lcl} \rho_0 & = & [] \\ \rho_{i+1} & = & \rho_i \oplus [X_{i+1} \leftarrow e_{i+1}] \text{ where } \{E_{i+1}\}\ \rho_i \rightarrow_{\mathsf{e}} e_{i+1} \quad (\forall i \in 0..n-1) \\ \rho_{glob} & = & \rho_n \end{array}$$

Example 4.10. Consider the following ordered list of constant declarations.

```
1   const X1: nat := 0
2   const X2: nat := X1 + 1
```

The corresponding global store is constructed as follows:

$$\begin{array}{lcl} \rho_0 & = & [] \\ \rho_1 & = & [X1 \leftarrow 0] \\ \rho_2 & = & [X1 \leftarrow 0,\ X2 \leftarrow 1] \\ \rho_{glob} & = & [X1 \leftarrow 0,\ X2 \leftarrow 1] \end{array}$$

□

4.4.3 Store and memory construction at component invocation

We are concerned with component invocations, regardless whether they are blocks, environments, or mediums. Hence, we will write $\mathcal{C}$ to denote a component and $\mathcal{C}' \rightarrow \mathcal{C}$ to denote that $\mathcal{C}'$ is an instance of $\mathcal{C}$.

Consider the invocation of $\mathcal{C}' \rightarrow \mathcal{C}$, where σ is the stack of the caller, ρ and μ are the current store and memory. For actual parameter lists or channels in $\mathcal{C}'$ and their corresponding formal parameter lists in $\mathcal{C}$, we write:

parameter	formal	actual
constant	$vars_c$	$args_c$
input / receive	$vars_{in_1}, \ldots, vars_{in_m}$	$acts_{in_1}, \ldots, acts_{in_m}$
output / send	$vars_{out_1}, \ldots, vars_{out_n}$	$acts_{out_1}, \ldots, acts_{out_n}$
blocks	$B_{b_1}, \ldots, B_{b_p}$	$B'_{b_1}, \ldots, B'_{b_p}$

For conciseness, we assume that temporary and static variable lists are unified into one list for each kind of variables. We can then write $vars_v$ and $vars_{sv}$ for the list of temporary and static variables, respectively.

To write down the transition rules of $\mathcal{C}'$ invocation, we define two functions:

- "body $(args_c, \mathop{\texttt{++}}_{k\in 1..m} acts_{in_k}, \sigma, \rho, \mu)$" returns a local store in which the body of $\mathcal{C}'$ will execute.
- "return $(\mathop{\texttt{++}}_{k\in 1..n} acts_{out_k}, \sigma.\mathcal{C}', \rho, \rho_{ret}, \mu_{ret})$" returns a couple (store, memory) updating the current store and memory. Store ρ_{ret} and memory μ_{ret} are assumed to be produced by the component invocation.

More precisely, during the execution of $\mathcal{C}'$, store "body $(args_c, \mathop{\texttt{++}}_{k\in 1..m} acts_{in_k}, \sigma, \rho, \mu)$" is constructed. It assigns values to constant, input, and receive parameters as well as to temporary and static variables. For this purpose, intermediate stores ρ_c, ρ_{in}, ρ_{sv}, and ρ_v, are constructed as follows:

- Store ρ_c assigns values to formal constant parameters, using function *init_assign*. Default expressions of constant parameters may depend on global constants, which are assigned in store ρ_{glob}.

$$\rho_c \quad \in \quad \text{init\_assign}\ (args_c,\ vars_c,\ \rho_{glob},\ \rho)$$

- Store ρ_{in} assigns values to formal input and receive parameters. Default expressions of input parameters may depend both on global constants and constant parameters of $\mathcal{C}$, which are assigned in store $\rho_{glob} \oplus \rho_c$.

$$\rho_{in} \quad \in \quad \text{init\_assign}\ (\mathop{\texttt{++}}_{k\in 1..m} acts_{in_k},\ \mathop{\texttt{++}}_{k\in 1..m} vars_{in_k},\ \rho_{glob} \oplus \rho_c,\ \rho)$$

– Store ρ_v assigns to temporary variables the evaluation of their respective initialisation expressions. Such expressions may depend on global constants as well as on constant and input parameters of $\mathcal{C}'$, all of them assigned in store $\rho_{glob} \oplus \rho_c \oplus \rho_{in}$.

$$\rho_v \quad = \quad \text{init } (vars_v,\ \rho_{glob} \oplus \rho_c \oplus \rho_{in})$$

– Store ρ_{sv} assigns values to static variables. Each variable is assigned the value it has in the end of the previous step of $\mathcal{C}'$ and available in the store $\mu\ (\sigma.\mathcal{C}')$. An exception is the first step, during which static variables are assigned the values of their default expressions. Default expressions may depend on global constants and the constant parameters of $\mathcal{C}'$, which are both assigned in store $\rho_{glob} \oplus \rho_c$.

$$\rho_{sv} \quad = \quad \text{static } (vars_{sv},\ \sigma,\ \rho_{glob} \oplus \rho_c,\ \mu)$$

Hence, the local store returned by function *body* can be defined as:

$$\text{body } (args_c,\ \mathop{\texttt{++}}_{k\in 1..m} acts_{in_k},\ \sigma,\ \rho,\ \mu) = \rho_{glob} \oplus \rho_c \oplus \rho_{in} \oplus \rho_{sv} \oplus \rho_v$$

The body of $\mathcal{C}'$ is executed in the *local* store and the *current* memory and terminates producing a store ρ_{ret} and a memory μ_{ret}. In this respect, the current store and memory are updated, using function *return*. The current store ρ is updated with the actual values of output parameters assigned in ρ_{ret}. The current memory μ is updated with the current values of static variables of $\mathcal{C}'$ assigned in ρ_{ret} together with those of subblocks assigned in μ_{ret}.

$$\text{return } (\mathop{\texttt{++}}_{k\in 1..n} acts_{out_k},\ \sigma.\mathcal{C}',\ \rho,\ \rho_{ret},\ \mu_{ret}) = \begin{pmatrix} \rho \oplus \text{update } (\mathop{\texttt{++}}_{k\in 1..n} acts_{out_k},\ \mathop{\texttt{++}}_{k\in 1..n} vars_{out_k},\ \rho_{ret}), \\ \mu_{ret} \oplus [\sigma.\mathcal{C}' \leftarrow \rho_{ret|vars_{sv}}] \end{pmatrix}$$

In the sequel, we present applications of these computations on blocks and environments.

4.5 Blocks

For a concise presentation of the semantics of blocks, we consider blocks with no receive and send formal parameters. Receive (resp. send) formal parameters and their corresponding actual parameters are used in computations exactly as their input (resp. output) counterparts. We write I_0 for the body of a block B and consider the following aliasing and invocation of B:

alias B {$args_c$} **as** B'
B' ($acts_{in_1}$, ..., $acts_{in_m}$, $acts_{out_1}$, ..., $acts_{out_n}$)

The semantics of such block invocation inside a component (subblock) and inside a system (highest-level blocks) can both be defined by rule R11 in Table 4.4.

$$\textbf{(R11)}\ \frac{\begin{array}{l}(P1)\ \rho_{exec} \in \text{body}\ (args_c,\ \underset{k\in 1..m}{\text{++}}\ acts_{in_k},\ \sigma,\ \rho,\ \mu)\\ (P2)\ \{I_0\}\ \sigma.B',\ \rho_{exec},\ \mu \xrightarrow{\epsilon}_{\text{i}} \rho_{ret},\ \mu_{ret}\\ (P3)\ (\rho',\ \mu') = \text{return}\ (\underset{k\in 1..n}{\text{++}}\ acts_{out_k},\ \sigma.B',\ \rho,\ \rho_{ret},\ \mu_{ret})\end{array}}{\{B'\ (acts_{in_1}, \ldots, acts_{in_m}, acts_{out_1}, \ldots, acts_{out_n})\}\ \sigma, \rho, \mu \xrightarrow{\epsilon}_{\text{i,s}} \rho',\ \mu'}$$

Table 4.4: Sos rule of blocks

The execution of the invoked block starts by constructing a local store ρ_{exec} (premise *P1*) in which the block body I_0 will execute (premise *P2*). All actual input parameters are required for block execution ($\underset{k\in 1..m}{\text{++}}\ acts_{in_k} \neq \epsilon$). The execution of the block invocation terminates by producing store ρ' and memory μ' (premise *P3*). All actual output parameters are updated when the block returns ($\underset{k\in 1..n}{\text{++}}\ acts_{out_k} \neq \epsilon$). The label of the transition is necessarily ϵ, since GRL static semantics prohibit the use of signals inside blocks.

Example 4.11. We illustrate the store and memory construction presented in Section 4.4.3 on block *B_Edge*. To this aim, we consider the aliasing and invocation of block *B_Edge* below (lines 13, 15) in the current store ρ and memory μ.

```
1   block B_Edge {Rising_Mode: bool := true, Falling_Mode: bool := false}
2                (in  Logic_Signal : bool := true,
3                 out Edge_Detected: bool) is
4     static var Pre_Signal: bool := false
5     var Rise, Fall       : bool
6     Rise          := Logic_Signal and not (Pre_Signal);
7     Fall          := not (Rise);
8     Edge_Detected := (Rising_Mode and Rise) or (Falling_Mode and Fall);
9     Pre_Signal    := Logic_Signal
10  end block
11
12  system ... is
13    alias B_Edge {_, _} as B_Edge
14    ...
15    B_Edge (Cmd_P1, ?Edge_Cmd_P1)
16  end system
```

Block *B_Edge* is invoked inside a system, meaning that $\sigma = \epsilon$. Assume that: (i) $\rho_{glob} = []$ and (ii) input *Cmd_P1* takes value *true* in the first step, meaning that $\rho(Cmd\_P1) = \text{true}$. The execution of *B_Edge* first step starts by constructing the following intermediates stores.

ρ_c	=	[Rising_Mode ← true, Falling_Mode ← false]
ρ_{in}	=	[Logic_Signal ← true]
ρ_v	=	[]
ρ_{sv}	=	[Pre_Signal ← false]

The sum of those stores, using function *body*, results in the following local store:

$$body\ (\langle \_,\ \_\rangle,\ \langle Cmd\_P1\rangle,\ \epsilon,\ \rho,\ \mu) = \begin{bmatrix}\text{Rising\_Mode} \leftarrow \text{true},\ \text{Falling\_Mode} \leftarrow \text{false},\\ \text{Logic\_Signal} \leftarrow \text{true},\ \text{Pre\_Signal} \leftarrow \text{false}\end{bmatrix}$$

The execution of *B_Edge* body terminates by producing a store ρ_{ret} and a memory μ_{ret} given in the table below. Store ρ_{ret} assigns values to temporary variables *Rise* and *Fall*, output *Edge_Detected*, and updates the value of the static variable *Pre_Signal*. Memory μ_{ret} is equal to μ, since the block does not invoke subblocks.

ρ_{ret}	=	$\left[\begin{array}{l}\text{Rising\_Mode} \leftarrow \text{true},\ \text{Falling\_Mode} \leftarrow \text{false},\ \text{Logic\_Signal} \leftarrow \text{true},\\ \text{Pre\_Signal} \leftarrow \text{true},\ \text{Rise} \leftarrow \text{true},\ \text{Fall} \leftarrow \text{false},\ \text{Edge\_Detected} \leftarrow \text{true}\end{array}\right]$
μ_{ret}	=	μ

The execution of *B_Edge* invocation terminates by producing the following store and memory.

$$return\ (\langle ?Edge\_Cmd\_P1\rangle,\ B\_Edge,\ \rho_{ret},\ \mu_{ret}) = \left(\begin{array}{l}\rho \oplus [\text{Edge\_Cmd\_P1} \leftarrow \text{true}]\\ \mu \oplus [\text{B\_Edge} \leftarrow [\text{Pre\_Signal} \leftarrow \text{true}]]\end{array}\right)$$

□

4.6 Environments and mediums

For conciseness, we present only the semantics of environments. The semantics of mediums is defined in the same way as environments, except that input (resp. output) channels are replaced by receive (resp. send) channels and there are no activation parameters. The semantics of environments are defined in Table 4.5.

The execution of environments is guarded by signals. An environment is triggered only to execute a data signal, if an interaction on a channel occurs, or to execute an activation signal, constraining the activation of a block. If no interaction occurs, the environment never executes.

Rule *R12* defines the semantics of an environment N' when a connected block requests to interact on one input channel $chan_{in_i}$ in $chan_{in_1}$, ..., $chan_{in_m}$. Hence, contrarily to block semantics, only the actual parameters composing $chan_{in_i}$ are assigned values in store ρ_{exec} (premise *P1*). An execution path containing the signal associated to $chan_{in_i}$ is selected by the body I_0 of N' (premise *P2*). Such a signal is assumed to be both defined and reachable inside I_0, according to GRL static semantics. Hence, label "?$get\_vars(vars_i)$" (premise *P2*), necessarily different from ϵ, indicates the signal

$$\textbf{(R12)}\ \frac{\begin{array}{ll}(P1) & i \in in_1..in_m \quad \rho_{exec} \in \text{body } (args_c,\ chan_i,\ \sigma,\ \rho,\ \mu) \\ (P2) & \{I_0\}\ \sigma.N',\ \rho_{exec},\ \mu \xrightarrow{?get\_vars(vars_i)}_{\mathrm{i}} \rho_{ret},\ \mu_{ret} \\ (P3) & (\rho',\ \mu') = \text{return } (\epsilon,\ \sigma.N',\ \rho,\ \rho_{ret},\ \mu_{ret})\end{array}}{\{N'\ (chan_{in_1}, \ldots, chan_{in_m}, chan_{out_1}, \ldots, chan_{out_n}, B'_{b_1}, \ldots, B'_{b_p})\}\ \sigma, \rho, \mu \xrightarrow{chan_i}_{\mathrm{i}} \rho',\ \mu'}$$

$$\textbf{(R13)}\ \frac{\begin{array}{ll}(P1) & i \in out_1..out_n \quad \rho_{exec} \in \text{body } (args_c,\ \epsilon,\ \sigma,\ \rho,\ \mu) \\ (P2) & \{I_0\}\ \sigma.N',\ \rho_{exec},\ \mu \xrightarrow{get\_vars(vars_i)}_{\mathrm{i}} \rho_{ret},\ \mu_{ret} \\ (P3) & (\rho',\ \mu') = \text{return } (chan_i,\ \sigma.N',\ \rho,\ \rho_{ret},\ \mu_{ret})\end{array}}{\{N'\ (chan_{in_1}, \ldots, chan_{in_m}, chan_{out_1}, \ldots, chan_{out_n}, B'_{b_1}, \ldots, B'_{b_p})\}\ \sigma, \rho, \mu \xrightarrow{chan_i}_{\mathrm{i}} \rho',\ \mu'}$$

$$\textbf{(R14)}\ \frac{\begin{array}{ll}(P1) & i \in b_1..b_p \quad \rho_{exec} \in \text{body } (args_c,\ \epsilon,\ \sigma,\ \rho,\ \mu) \\ (P2) & \{I_0\}\ \sigma.N',\ \rho_{exec},\ \mu \xrightarrow{B_i}_{\mathrm{i}} \rho_{ret},\ \mu_{ret} \\ (P3) & (\rho',\ \mu') = \text{return } (\epsilon,\ \sigma.N',\ \rho,\ \rho_{ret},\ \mu_{ret})\end{array}}{\{N'\ (chan_{in_1}, \ldots, chan_{in_m}, chan_{out_1}, \ldots, chan_{out_n}, B'_{b_1}, \ldots, B'_{b_p})\}\ \sigma, \rho, \mu \xrightarrow{B'_i}_{\mathrm{i}} \rho',\ \mu'}$$

Table 4.5: Sos rules of environments

execution. No output actual parameters are updated by the rule, as denoted by the symbol ϵ in function *return* (premise *P3*).

In a similar way, rule *R13* defines the semantics of environment execution when a connected block requests to interact on one output channel $chan_{out_i}$ in $chan_{out_1}, \ldots, chan_{out_n}$. No input actual parameters are available, as denoted by the symbol ϵ in function *body* (premise *P1*).

Rule *R14* defines the semantics of an environment N' when a block whose identifier B'_{b_i} is in $B'_{b_1}, \ldots, B'_{b_p}$ wishes to perform a step (premise *P1*). The body of N' tries to select an execution path containing the signal associated to B'_{b_i}. If such execution path is reachable, the body executes and the transition label indicates that the signal associated to B'_{b_i} has executed (premise *P2*); otherwise, N' does not execute. No input neither output actual parameters are involved in rule *R14*, as suggest by symbol ϵ in both functions *body* and *return* (premises *P1* and *P3*).

Example 4.12. To illustrate the semantics of environments, consider the environment *Disable* below.

```
1  environment Disable {b:bool := true} (block B) is
2    if not (b) then
3      enable B
4    end if
5  end environment
```

Environment *Disable* can be used either to allow an arbitrary activation of a block *B1* or to forever prohibit the activation of a block *B2*.

```
1  Disable {false}(B1) — always execute B1
2  Disable {true} (B2) — never execute B2
```

Let ρ and μ be the current store and memory and assume that $\rho_{glob} = []$. The invocation of *Disable* with value *false* starts by constructing the following intermediates stores.

ρ_c	=	[b ← false]
ρ_{in}	=	[]
ρ_v	=	[]
ρ_{sv}	=	[]

The sum of those stores, using function *body*, results in the following local store:

$$body\ (\langle false \rangle,\ \epsilon,\ \epsilon,\ \rho,\ \mu)\ =\ [\text{b} \leftarrow \text{false}]$$

The execution of the environment body terminates by producing the store [b ← false] and without updating the memory. The execution of *Disable* invocation terminates by producing the following store and memory.

$$return\ (\epsilon,\ Disable,\ \rho,\ [\text{b} \leftarrow \text{false}],\ \mu)\ =\ (\rho \oplus [\text{b} \leftarrow \text{false}],\ \mu)$$

The derivation of transition rules defining *Disable* invocation is:

$$\frac{\dfrac{}{\{\ b\ \}\ [b \leftarrow false] \rightarrow_{\mathsf{e}}\ false} \qquad \left\{\begin{array}{l}\textbf{if not } (b) \textbf{ then}\\ \quad \textbf{enable } B\\ \textbf{end if}\end{array}\right\}\ Disable,\ [b \leftarrow false],\ \mu \xrightarrow{B}_{\mathsf{i}}\ \rho \oplus [b \leftarrow false],\ \mu}{\{\ Disable\ \{\textbf{false}\}(B1)\ \}\ \epsilon,\ \rho,\ \mu\ \xrightarrow{B1}_{\mathsf{i}}\ \rho,\ \mu}$$

No derivation rule corresponds to the invocation of *Disable* with value *true*. Since signal "**enable** B" is never reachable (b=**true**), the environment never executes. □

Example 4.13. Consider the definition and invocation of environment *Signal* below.

```
1  environment Signal(out Cmd: bool) is
2    when Cmd -> Cmd := true
3  end environment
4  ...
5  Signal (?Cmd)
```

Let ρ and μ be the current store and memory and assume that $\rho_{glob} = []$. By applying functions *body* and *return*, we have:

$body\ (\epsilon,\ \epsilon,\ \epsilon,\ \rho,\ \mu)$	=	[]
$return\ (\langle ?Cmd \rangle,\ Signal,\ \rho,\ \rho \oplus [\text{Cmd} \leftarrow \text{true}],\ \mu_{ret})$	=	$(\rho \oplus [\text{Cmd} \leftarrow \text{true}],\ \mu)$

□

4.7 Systems

This section presents the semantics of systems. We first present some required sets and auxiliary functions. Then, we define the semantic rule of system execution. Finally, we discuss the semantic model of GRL with respect to related work.

4.7.1 Sets and auxiliary functions

Let S be a system. We write *block_invoc*, *env_invoc*, and *med_invoc* as shorthands for component invocations (see Table 3.9, page 51). Each component $\mathcal{C}'$ in S is associated to a unique index. We write B'_i (resp. N'_i, M'_i) for the name of the component whose invocation is $block\_invoc_i$ (resp. env_invoc_i, med_invoc_i). We write $indices(S, \mathsf{block})$, $indices(S, \mathsf{env})$, $indices(S, \mathsf{med})$ for the set of indices of respectively blocks, environments, and mediums inside S.

Let *mode* be a partial function mapping actual parameters and channels to the set {in, out}.

$mode\ (X_0)$	=	in	$mode\ (\texttt{<}X_0\texttt{,}\ldots\texttt{,}X_n\texttt{>})$	=	in
$mode\ (\texttt{?}X_0)$	=	out	$mode\ (\texttt{?<}X_0\texttt{,}\ldots\texttt{,}X_n\texttt{>})$	=	out
$mode\ (\_)$	=	in	$mode\ (\texttt{<\_,}\ldots\texttt{,\_>})$	=	in
$mode\ (\texttt{?\_})$	=	out	$mode\ (\texttt{?<\_,}\ldots\texttt{,\_>})$	=	out
$mode\ (\textbf{any}\ type_0)$	=	in	$mode\ (\textbf{any}\ type_0\texttt{,}\ldots\texttt{,}\textbf{any}\ type_n)$	=	in

Each channel used in S is associated to an index such that:

- Channels having the same non-empty set of variables are associated to the same index. In this case, we write $mode\ (\mathcal{C}',\ chan_k)$ for the mode of channel $chan_k$ in component $\mathcal{C}'$. For example, a channel $\texttt{<}X_0\texttt{>}$ of a block B'_i is associated to the same index as a channel $\texttt{?<}X_0\texttt{>}$ of an environment N'_j. We also have "$mode\ (B'_i,\ \texttt{<}X_0\texttt{>}) =$ in" and "$mode\ (N'_j,\ \texttt{?<}X_0\texttt{>}) =$ out".
- Each other channel is associated to a unique index.

We write $indices(\mathcal{C}', \mathsf{chan})$ for the set of channel indices used in $\mathcal{C}'$ invocation.

Example 4.14. Consider system *Main* below. The table on the right-hand side summarises component and channel indexation.

```
1   system Main (Y: bool) is
2     alias B_Edge  {_, _}  as B_Edge,
3           Disable {false} as Disable,
4           Signal          as Signal
5     var X :bool
6     block list
7       B_Edge (<X>, ?<Y>)
8     environment list
9       Disable (B_Edge),
10      Signal  (?<X>)
11  end system
```

component	index
B_Edge	1
Disable	2
Signal	3
channel	**index**
<X>	1
?<Y>	2
?<X>	1

By applying function *indices*, we have:

indices	(*Main*, block)	=	{1}
indices	(*Main*, env)	=	{2, 3}
indices	(*Main*, med)	=	{}

indices	(*B_Edge*, chan)	=	{1, 2}
indices	(*Disable*, chan)	=	{}
indices	(*Signal*, chan)	=	{1}

□

The execution of systems is guided by the execution of their active components, i.e., blocks. Let B'_i be a block such that $i \in indices(S, \mathsf{block})$. The components connected to B'_i are identified by their indices. We define sets *In*, *Out*, *Rec*, and *Snd* containing the indices of components connected to respectively input, output, receive, and send channels of B'_i. Such sets are possibly empty; but if not empty, they are not necessarily singletons. Formally, they are defined as follows:

$$\begin{array}{lcl}
In\ (B'_i) & = & \{j \in indices(S, \mathsf{env}) \mid \exists \mathsf{k},\ \mathsf{k} \in (indices(B'_i, \mathsf{chan}) \cap indices(N'_j, \mathsf{chan})) \wedge mode(B'_i, chan_k) = \mathsf{in}\} \\
Out\ (B'_i) & = & \{j \in indices(S, \mathsf{env}) \mid \exists \mathsf{k},\ \mathsf{k} \in (indices(B'_i, \mathsf{chan}) \cap indices(N'_j, \mathsf{chan})) \wedge mode(B'_i, chan_k) = \mathsf{out}\} \\
Rec\ (B'_i) & = & \{j \in indices(S, \mathsf{med}) \mid \exists \mathsf{k},\ \mathsf{k} \in (indices(B'_i, \mathsf{chan}) \cap indices(M'_j, \mathsf{chan})) \wedge mode(B'_i, chan_k) = \mathsf{in}\} \\
Snd\ (B'_i) & = & \{j \in indices(S, \mathsf{med}) \mid \exists \mathsf{k},\ \mathsf{k} \in (indices(B'_i, \mathsf{chan}) \cap indices(M'_j, \mathsf{chan})) \wedge mode(B'_i, chan_k) = \mathsf{out}\}
\end{array}$$

We also need to identify the environment constraining B'_i activation, if any. Let $Act\ (B'_i)$ be a set containing the index of such an environment. The set is either singleton or empty, since at most one environment can constrain a block activation, according to static semantics.

Still, to the input and receive channels that are not connected to other components, arbitrary values should be assigned. Let $Any\ (B'_i)$ be a set containing the indices of B'_i channels of the form $\texttt{<}X_1\texttt{,}\ \ldots\texttt{,}\ X_n\texttt{>}$ and not connected to other components. Let *assign_any* be a function assigning arbitrary values to variables. The function returns a set of stores, yielding nondeterminism.

$$\begin{array}{lcl}
\text{assign_any}\ (\texttt{<}X_1\texttt{,}\ \ldots\texttt{,}X_n\texttt{>}, \langle T_1, \ldots, T_n\rangle) & = & \text{assign_any}\ (X_1, T_1) \oplus\ \ldots \oplus \text{assign_any}\ (X_n, T_n) \\
\text{assign_any}\ (X, T) & = & \{[X \leftarrow e] \mid e \in T\}
\end{array}$$

Example 4.15. Consider system *Main* defined in Example 4.14. For block *B_Edge*, we have:

$In\ (B\_Edge)$	=	$\{3\}$	$Snd\ (B\_Edge)$	=	$\{\}$
$Out\ (B\_Edge)$	=	$\{\}$	$Act\ (B\_Edge)$	=	$\{2\}$
$Rec\ (B\_Edge)$	=	$\{\}$	$Any\ (B\_Edge)$	=	$\{\}$

□

Example 4.16. Consider system *Main_Edge* below.

```
1  system Main_Edge (X, Y: bool) is
2    alias B_Edge {_, _} as B_Edge
3    block list
4      B_Edge (<X>, ?<Y>)
5  end system
```

By applying function *assign*, we have:

$$\text{assign\_any}\ (\texttt{<}X\texttt{>}, \mathbf{bool}) = \{[X \leftarrow \mathit{false}], [X \leftarrow \mathit{true}]\}$$

□

To specify the labels of transitions, we define two functions *channel* and *transition*. Function *channel* defines the labels of transition rules corresponding to component executions. Given a block B'_i, a component $\mathcal{C}'_j$, and an element in $\{\mathsf{in}, \mathsf{out}\}$, function

channel returns the channel of mode m in B'_i that is connected to $\mathcal{C}'_j$. Note that GRL static semantics ensure that there is at most one such channel.

$$channel\ (B'_i,\ \mathcal{C}'_j,\ m) = \{chan_k \mid k \in (indices(B'_i,\ \mathsf{chan}) \cap indices(\mathcal{C}'_j,\ \mathsf{chan}))\ \wedge\ mode(B'_i, chan_k) = m\}$$

Example 4.17. Consider system *Main* defined in Example 4.14. By applying function *channel*, we have:

$$channel\ (B\_Edge,\ Signal,\ \mathsf{in}) = \texttt{<}X\texttt{>}$$

□

Function *transition* defines the label of transition rule corresponding to the system execution. The label is built upon the actual channels of blocks.

$$\begin{array}{lcl}
\text{transition } (\texttt{<}X_1, \ldots, X_n\texttt{>},\ \rho) & = & \text{transition } (X_1,\ \rho)\texttt{,}\ \ldots\texttt{,}\ \text{transition } (X_n,\ \rho) \\
\text{transition } (\texttt{?<}X_1, \ldots, X_n\texttt{>},\ \rho) & = & \text{transition } (X_1,\ \rho)\texttt{,}\ \ldots\texttt{,}\ \text{transition } (X_n,\ \rho) \\
\text{transition } (\texttt{<}\textbf{any}\ type_1, \ldots, \textbf{any}\ type_n\texttt{>},\ \rho) & = & \texttt{_,} \cdots \texttt{,_} \\
\text{transition } (\texttt{<_,} \ldots, \texttt{_>},\ \rho) & = & \texttt{_,} \cdots \texttt{,_} \\
\text{transition } (\texttt{?<_,} \ldots, \texttt{_>},\ \rho) & = & \texttt{_,} \cdots \texttt{,_} \\
\\
\text{transition } (X_0,\ \rho) & = & \begin{cases} X_0 = \rho(X_0) & \text{if } X_0 \in visible\ (S) \\ \texttt{_} & \text{otherwise} \end{cases}
\end{array}$$

where *visible* (S) denotes the set of the visible parameters in S

4.7.2 Semantics of systems

The semantics of systems is given in Table 4.6. Rule *R15* defines the execution of a block step together with its connected components.

Before the execution of a block B'_i, its activation should be granted and values should be assigned to all input and receive actual parameters. To this aim, the environment N'_j, where $j \in Act\ (B'_i)$, is executed in the empty store and in its own memory. This memory is extracted from the current memory μ, by using function *mem* (premise *P1*). It produces a store $\rho'_{\mathcal{A}_j}$ and a memory $\mu'_{\mathcal{A}_j}$. Store $\rho'_{\mathcal{A}_j}$ is the empty store, since no output channel is involved in the environment execution.

Similarly, all environments and mediums whose indices are in $In\ (B'_i) \cup Rec\ (B'_i)$ are executed in the empty store and in their own memories (premises *P2* and *P3*). A particular case is when an environment index is in $In\ (B'_i) \cap Act\ (B'_i)$, meaning that the environment not only constrains B'_i activation but also is connected to an input channel of B'_i. In such case, the environment memory is the one produced by its previous execution during the current step of B'_i, which is captured by the definition of $\mu_{\mathcal{I}_l}$ (row (c)).

Hence, a local store ρ_i, in which the block will execute, is constructed (row (b)). The store assigns to input and receive parameters of B'_i the values produced by the preceding

$$
\textbf{(R15)}\quad
\begin{array}{llllll}
\text{(P1)} & \forall j \in Act\ (B'_i) & \{env\_invoc_j\} & \epsilon,\ [],\ mem\ (\mu,\ N'_j) & \xrightarrow{B'_i}_{\mathsf{i}} & \rho'_{\mathcal{A}_j},\ \mu'_{\mathcal{A}_j} \\
\text{(P2)} & \forall l \in In\ (B'_i) & \{env\_invoc_l\} & \epsilon,\ [],\ \mu_{\mathcal{I}_l} & \xrightarrow{channel(B'_i,\ ,N'_l,\ \mathsf{in})}_{\mathsf{i}} & \rho'_{\mathcal{I}_l},\ \mu'_{\mathcal{I}_l} \\
\text{(P3)} & \forall k \in Rec\ (B'_i) & \{med\_invoc_k\} & \epsilon,\ [],\ \ mem\ (\mu,\ M'_k) & \xrightarrow{channel(B'_i,\ M'_k,\ \mathsf{in})}_{\mathsf{i}} & \rho'_{\mathcal{R}_k},\ \mu'_{\mathcal{R}_k} \\
\text{(P4)} & & \{block\_invoc_i\} & \epsilon,\ \rho_i,\ mem\ (\mu,\ B'_i) & \xrightarrow{\epsilon}_{\mathsf{i}} & \rho'_i,\ \mu'_i \\
\text{(P5)} & \forall m \in Out\ (B'_i) & \{env\_invoc_m\} & \epsilon,\ \rho'_i,\ \mu_{\mathcal{O}_m} & \xrightarrow{channel(B'_i,\ N'_m,\ \mathsf{out})}_{\mathsf{i}} & \rho'_{\mathcal{O}_m},\ \mu'_{\mathcal{O}_m} \\
\text{(P6)} & \forall n \in Snd\ (B'_i) & \{med\_invoc_n\} & \epsilon,\ \rho'_i,\ \mu_{\mathcal{S}_n} & \xrightarrow{channel(B'_i,\ M'_n,\ \mathsf{out})}_{\mathsf{i}} & \rho'_{\mathcal{S}_n},\ \mu'_{\mathcal{S}_n}
\end{array}
$$

$$
\mu \xrightarrow{\ell} \mu \oplus \bigoplus_{j \in Act\ (B'_i)} \mu'_{\mathcal{A}} \oplus \bigoplus_{l \in In\ (B'_i)} \mu'_{\mathcal{I}} \oplus \bigoplus_{k \in Rec\ (B'_i)} \mu'_{\mathcal{R}} \oplus \bigoplus_{m \in Out\ (B'_i)} \mu'_{\mathcal{O}} \oplus \bigoplus_{n \in Snd\ (B'_i)} \mu'_{\mathcal{S}}
$$

ρ_{any}	$\triangleq$	$\bigoplus_{p \in Any\ (B'_i)} assign\_any\ (args_p,\ types(vars_p))$	(a)
ρ_i	$\triangleq$	$\rho_{any} \oplus \bigoplus_{j \in Act\ (B'_i)} \rho'_{\mathcal{A}_j} \oplus \bigoplus_{l \in In\ (B'_i)} \rho'_{\mathcal{I}} \oplus \bigoplus_{k \in Rec\ (B'_i)} \rho'_{\mathcal{R}}$	(b)
$\mu_{\mathcal{I}_l}$	$\triangleq$	$mem\ (\mu,\ N'_l) \oplus \bigoplus_{j \in Act\ (B'_i)} mem\ (\mu'_{\mathcal{A}_j},\ N'_l)$	(c)
$\mu_{\mathcal{O}_m}$	$\triangleq$	$mem\ (\mu,\ N'_m) \oplus \bigoplus_{j \in Act\ (B'_i)} mem\ (\mu'_{\mathcal{A}_j},\ N'_m) \oplus \bigoplus_{l \in In\ (B'_i)} mem\ (\mu'_{\mathcal{I}_l},\ N'_m)$	(d)
$\mu_{\mathcal{S}_n}$	$\triangleq$	$mem\ (\mu,\ M'_n) \oplus \bigoplus_{k \in Rec\ (B'_i)} mem\ (\mu'_{\mathcal{R}_k},\ M'_n)$	(e)
ℓ	$=$	$B'_i\ (\text{transition}(chan_1,\ \rho'_i), \ldots, \text{transition}(chan_m,\ \rho'_i))$ $[\text{transition}(chan'_1,\ \rho'_i), \ldots, \text{transition}(chan'_n,\ \rho'_i)]$	(f)

Table 4.6: Sos rule of systems

components. Part of those parameters are available in stores $\rho'_{\mathcal{R}_k}$ ($k \in Rec\ (B'_i)$) and $\rho'_{\mathcal{I}_l}$ ($l \in In\ (B'_i)$). For channels whose indices are in $Any\ (B'_i)$, a store ρ_{any} is constructed, assigning arbitrary values to its variables (row (a)). The execution of block B'_i in store ρ_i and its own memory, producing a store ρ'_i and a memory μ'_i (premise *P4*).

Last, all environments and mediums whose indices are in $Out\ (B'_i) \cup Snd\ (B'_i)$ are executed in store ρ'_i and their own memories (premises *P5* and *P6*, rows (d) and (e)). In particular, $\rho_{\mathcal{O}_m} = \rho'_i$ ($\forall m \in Out\ (B'_i)$) and $\rho_{\mathcal{S}_n} = \rho'_i$ ($\forall n \in Snd\ (B'_i)$), because the values produced by B'_i execution are needed in component executions.

The execution of the system defines a transition updating the current memory μ with all the memories produced by the executed components. Such transition denotes a multiway synchronisation between a specific block and its connected environments and mediums. The transition label indicates which block is executing and which values the block channels have carried (row (f)). The whole system LTS is constructed by instantiating the semantic rule for any block. This leads to an interleaving of block executions.

Example 4.18. Consider system *Main* defined in Example 4.14. See Examples 4.11, 4.12, and 4.13 for details about the execution of components *B_Edge*, *Disable*,

and *Signal*, respectively. The transition label is obtained by applying function *transition*, as follows:

$$\begin{array}{lll} \text{transition } (\texttt{<}X\texttt{>}, & [\text{X} \leftarrow \text{true},\ \text{Y} \leftarrow \text{true}]) & = \ \_ \\ \text{transition } (\texttt{?<}Y\texttt{>}, & [\text{X} \leftarrow \text{true},\ \text{Y} \leftarrow \text{true}]) & = \ Y \texttt{ = } \mathit{true} \end{array}$$

The transition rule defining the first execution of system *Main* in the empty memory is:

$$\frac{\begin{array}{llll} \{\ \mathit{Disable}\ \texttt{(}B\_Edge\texttt{)}\ \} & \epsilon,\ [], & [] \xrightarrow{B\_Edge}_{\text{i}} & [], \quad [] \\ \{\ \mathit{Signal}\ \texttt{(?<}X\texttt{>)}\ \} & \epsilon,\ [], & [] \xrightarrow{?X}_{\text{i}} & [\text{X} \leftarrow \text{true}], \quad [] \\ \{\ B\_Edge\ \texttt{(<}X\texttt{>, ?<}Y\texttt{>)}\ \} & \epsilon,\ [\text{X} \leftarrow \text{true}], & [] \xrightarrow{\epsilon}_{\text{i}} & \begin{bmatrix} \text{X} \leftarrow \text{true}, \\ \text{Y} \leftarrow \text{true} \end{bmatrix}, \quad [\text{B\_Edge} \leftarrow [\text{Pre\_Signal} \leftarrow \text{true}]] \end{array}}{[] \xrightarrow{B\_Edge\ (\_, Y = true)} [\text{B\_Edge} \leftarrow [\text{Pre\_Signal} \leftarrow \text{true}]]}$$

□

Remark 4.1. One might well aim to verify the behaviour of synchronous components before constructing the GALS system. The LTS corresponding to a block can be obtained by invoking the block inside a system with no other component. The following table summarises the sizes of the LTSs corresponding to a logical block *B_And*, *B_Edge* (Example 3.2, page 39), and *Exit* (Example 3.7, page 42).

Block	states	transitions	labels
B_And	1	4	4
B_Edge	2	4	4
Exit	4	16	10

In particular, the internal state of block *B_And* is the empty memory. Thus, its LTS contains one state with several outgoing transitions. The number of the LTS transitions corresponds to the possible values taken by its two inputs, both of Boolean type. Block *B_Edge* defines one static variable of Boolean type. Thus, its LTS contains two states, each corresponding to a Boolean value. ■

4.7.3 Relation with existing work

We discuss the relation of GRL semantics to some existing work issued from the synchronous and asynchronous communities.

As regards synchronous semantics, the idea of associating one LTS transition to a synchronous component step is not new. It has been adopted by Esterel [BG92], whose operational semantics are defined by means of LTSs. Transition labels in Esterel, like in GRL, are specified in terms of component inputs and outputs. Moreover, both GRL

and Esterel enjoy process algebra traits such as bisimulation reduction techniques.

As regards asynchronous semantics, the dichotomy of GRL components into active (blocks) and passive (environments and mediums) deviates from the classical process algebraic view. Process algebra usually abstract from the composition of a system into a set of components. The only relevant information (i.e., visible on LTSs) is the different actions performed by components and their composition. In GRL, contrarily, the system composition plays a key role. Only blocks are of interest. Block identity, input and output values, and interleaving between blocks are all visible on LTSs.

Related to the dichotomy into active and passive components is communication asymmetry in GRL. Blocks do not define signals; they are self-activated. Their activation triggers, through signals, medium and environment executions, which always accept to interact. In the Sos rules, asymmetry is expressed by labels ϵ in block execution (see Table 4.4, page 71) and labels $\ell \neq \epsilon$ in medium and environment execution (see Table 4.5, page 73). In process algebra, all components should define communication actions; and communication is performed only if all participant components are ready.

Chapter 5

Translation from GRL into LNT

This chapter presents a syntax-directed translation from GRL into the process language LNT. We first give insights into the translation scheme. Then, for each GRL construct, we give an informal description, the formal translation functions, and some examples. For an informal presentation of the translation, the reader can omit the formal definition of translation functions. Afterwards, we briefly present the GRL2LNT translator implementing the proposed translation. Finally, we compare the LTSs generated by the translation to the LTSs of GRL semantics and to related work.

5.1 Overview of the translation

We translate a GALS-specific language into a full-fledged process language for asynchronous processes. GRL types, expressions, and statements are inspired by LNT. Their translation is straightforward and is presented in Section 5.2. Global constants are translated to LNT functions. Their translation is presented in Section 5.3.

In GRL, interaction between (synchronous) subblocks inside components and between (asynchronous) components inside systems occurs through common variables. In LNT, however, communication between asynchronous processes occurs through gates. Moreover, GRL actual parameters and channels can be unconnected whereas no similar notion is present in LNT. The translation of variable declaration as well as actual parameters and channels is given in Sections 5.4.2, 5.4.3, and 5.4.4.

GRL blocks are translated to LNT functions, whose execution is deterministic and atomic. We propose an encoding of the mutable internal state in LNT (Section 5.4.5), where no such notion exists. For GRL subblocks, the corresponding LNT functions are encapsulated in other LNT functions, which implement one block step. For GRL highest-level blocks, the corresponding LNT functions are encapsulated in LNT *wrapper* processes, which implement the (implicit) synchronous loop of GRL blocks. Each wrapper process interacts with other asynchronous components using gate communica-

tions, producing a transition sequence in the resulting LTS. To preserve the atomicity of those transition sequences, we propose a locking mechanism. The translation of blocks is presented in Section 5.5.

GRL environments and mediums are naturally translated to LNT processes. The translation of signals involves gate communications. In particular, the translation of activation signals cooperates with the locking mechanism in constraining the execution of wrapper processes. We present the translation of environments and mediums in Section 5.6.

GRL systems are translated to processes, called *root processes*. Inside a root process, the processes corresponding to GRL highest-level blocks, environments, and mediums are composed asynchronously. The translation of GRL systems is presented in Section 5.7.

5.2 Translation of variables, types, expressions, and statements

Each GRL variable X is translated to an LNT variable with the same name X[1]. Expressions and statements have a direct, one-to-one, correspondence with their LNT counterparts; the only exception concerns signals. Signals are translated to behaviours involving LNT communication actions. The imperative style of both GRL and LNT makes rather straightforward such a translation.

GRL data types are translated with no difficulty into LNT, owing to the ability of LNT to handle complex data types. GRL types **bool**, **char**, **string** and the user-defined ones are translated to LNT types with the same name as the corresponding GRL types. The translation of GRL numerical types is summarised in the following table:

GRL type	LNT type	Definition
nat	*Nat8*	**type** *Nat8* **is range** 0..255 **of** *Nat* **end type**
nat16	*Nat16*	**type** *Nat16* **is range** 0..65535 **of** *Nat* **end type**
nat32	**Nat**	Predefined in LNT
int	*Int8*	**type** *Int8* **is range** −128..127 **of** *Int* **end type**
int16	*Int16*	**type** *Int16* **is range** −32768..−32767 **of** *Int* **end type**
int32	**Int**	Predefined in LNT

The number of bits on which LNT numerical types **Nat** and **Int** are represented is set by default to 8. To enable the description of GRL numerical types represented on 16 and 32 bits, we set indeed such a number to 32, using the following pragmas:
!nat_bits *32*
!int_bits *32*
Types are defined in one LNT module, named "*GRL_V1*", which is systematically imported by each generated LNT module. There is a special enumerated type, named *block*, introduced by the translation. Type *block* contains the names of all highest-level

[1] In practice, the translation must ensure that the GRL name is not an LNT keyword. This is handled by the translator but we skip such low-level details in this presentation, for conciseness.

blocks encapsulated inside the root process under translation. It will be used to translate activation parameters.

We write *t2t* and *e2v* for the translation function of GRL types and expressions to their LNT counterpart. We write *i2s* for the translation function of GRL statements to LNT statements and behaviours. Excerpts of function *i2s* will be given when translating subblock invocation (Section 5.5.2) and signals (Section 5.6.1).

5.3 Translation of global constants

We consider the following global constants.

$$\textbf{const}\ X_1 : type_1\ \texttt{:=}\ E_1\ , \ldots , X_n : type_n\ \texttt{:=}\ E_n$$

The translation function, named *c2f*, of global constants is given in Table 5.1. The definition of each global constant is translated to an LNT function. The function name corresponds to the GRL constant name. The function returns the value expression assigned to the GRL constant.

$c2f\left(\ \textbf{const}\ X_1 : type_1\ \texttt{:=}\ E_1\ , \ldots , X_n : type_n\ \texttt{:=}\ E_n\ \right) =$	**function** $X_1 : t2t\ (type_1)$ **is**
	  return $e2v\ (E_1)$
	end function
	...
	function $X_n : t2t\ (type_n)$ **is**
	  return $e2v\ (E_n)$
	end function

Table 5.1: Translation function of global constants

Example 5.1. Consider the following GRL constants:

```
1  -- GRL code
2  const nb_max_cars: nat      := 4
3  const empty_queue: t_queue := t_queue (no_message)
```

These constants are translated to LNT as follows:

```
1  -- LNT code
2  function nb_max_cars: Nat8 is
3     return 4 of Nat8
4  end function
5
6  function empty_queue: t_queue is
7     return t_queue (no_message)
8  end function
```

□

5.4 Translation of variable declarations, parameters, and internal states

This section is structured as follows. We first introduce some sets and functions that will be used in the translation functions. Afterwards, we present the translation of GRL variable declarations, actual parameters, and actual channels into LNT constructs. Finally, we present the translation of the internal state notion.

5.4.1 Preliminaries

We use the following notations. We write $\mathcal{C}' \rightarrow \mathcal{C}$ to denote that $\mathcal{C}'$ is an instance of component $\mathcal{C}$, as in Chapter 4. We write $\mathcal{C}' \subset \mathcal{C}_1$ to denote that $\mathcal{C}'$ is a component instance used inside component $\mathcal{C}_1$. We write $sub(\mathcal{C})$ for the set of subblocks B'_k such that $B'_k \subset \mathcal{C}$.

Let $\mathcal{C}'$ be a component instance. $\mathcal{C}'$ may be invoked either with actual parameter lists "$args_1$, ..., $args_n$" (inside another component) or with actual channels "$chan_1$, ..., $chan_n$" (inside a system). Each actual parameter (resp. actual channel) in $\mathcal{C}'$ is associated to a unique index; and its corresponding formal parameter (resp. formal channel) is associated to the same index. We write $indices(\mathcal{C}', \mathsf{arg})$ (resp. $indices(\mathcal{C}', \mathsf{chan})$, similarly to Section 4.7.1) for the set of indices of actual parameters in $\mathcal{C}'$ invocation, namely {1, ..., n}, where n is the number of parameters (resp. channels) in $\mathcal{C}'$.

Let *connexion* be a partial function mapping actual parameters and channels to the set {connected, unconnected, wildcard}.

$connexion\ (X_0)$	= connected	$connexion\ (\texttt{<}X_1\texttt{ , ... , }X_n\texttt{>})$	= connected
$connexion\ (\texttt{?}X_0)$	= connected	$connexion\ (\texttt{?<}X_1\texttt{ , ... , }X_n\texttt{>})$	= connected
$connexion\ (\texttt{\_})$	= unconnected	$connexion\ (\texttt{<\_ , ... , \_>})$	= unconnected
$connexion\ (\texttt{?\_})$	= unconnected	$connexion\ (\texttt{?<\_ , ... , \_>})$	= unconnected
$connexion\ (\textbf{any}\ type_0)$	= wildcard	$connexion\ (\textbf{any}\ type_1\texttt{ , ... , }\textbf{any}\ type_n)$	= wildcard

We will also use function *mode* (see Section 4.7.1, page 75), mapping actual parameters and channels to the set {in, out}. Functions *connexion* and *mode* allow us to define the following sets on actual parameters and channels:

– The set $indices(\mathcal{C}', \mathsf{arg}, \mathsf{in}, \mathsf{unconnected})$ contains the indices of unconnected input an receive actual parameters.

$$indices(\mathcal{C}', \mathsf{arg}, \mathsf{in}, \mathsf{unconnected}) = \{k \in indices(\mathcal{C}', \mathsf{arg}) \mid mode(arg_k) = \mathsf{in} \wedge connexion(arg_k) = \mathsf{unconnected}\}$$

– The set $indices(\mathcal{C}', \mathsf{arg}, \mathsf{out}, \mathsf{unconnected})$ contains the indices of unconnected output and send actual parameters.

$$indices(\mathcal{C}', \mathsf{arg}, \mathsf{out}, \mathsf{unconnected}) = \{k \in indices(\mathcal{C}', \mathsf{arg}) \mid mode(arg_k) = \mathsf{out} \wedge connexion(arg_k) = \mathsf{unconnected}\}$$

– The set $indices(\mathcal{C}', \mathsf{chan}, \mathsf{connected})$ contains the indices of connected actual channels.
$indices(\mathcal{C}', \mathsf{chan}, \mathsf{connected}) = \{k \in indices(\mathcal{C}', \mathsf{chan}) \mid connexion(chan_k) = \mathsf{connected}\}$

Example 5.2. Consider the following block invocation, either inside another component or inside a system.

```
1   B' (X, _, ?Y, ?_)
```

We have the following sets, where the last set stipulates that B' is invoked inside a system.

$$\begin{array}{lcl} indices(B', \mathsf{arg}, \mathsf{in}, \mathsf{unconnected}) & = & \{2\} \\ indices(B', \mathsf{arg}, \mathsf{out}, \mathsf{unconnected}) & = & \{4\} \\ indices(B', \mathsf{chan}, \mathsf{connected}) & = & \{1, 3\} \end{array}$$

□

We will also use the following functions:

Function	Purpose
$type(\mathcal{C}', k)$	returns the type of the parameter whose index is k in component instance $\mathcal{C}'$
$default(\mathcal{C}, k)$	returns the default value of the formal parameter whose index is k in component $\mathcal{C}$
$variable(\mathcal{C}', k)$	builds a unique variable name built upon the actual parameter whose index is k in component instance $\mathcal{C}'$
$gate(X_1, \ldots, X_n)$	builds a unique gate name, namely $Gate\_X_1\_...\_X_n$, upon a variable identifier list. The function is an injection from variable lists to gate names
$channel(type_1, \ldots, type_n)$	builds a unique channel name, namely $Chan\_type_1\_...\_type_n$, upon a type identifier list. The function is an injection from type lists to channel names

Example 5.3. Consider the following GRL code excerpt.

```
1   block B (in X: nat := 0, out Y: bool) is
2     ...
3   end block
4   ...
5   alias B as B'
```

By applying functions *type*, *default*, and *variable*, we have:

$$\begin{array}{ll} type\ (B',\ 1) = \mathbf{nat} & default\ (B,\ 1) = 0 \\ type\ (B',\ 2) = \mathbf{bool} & variable\ (B',\ 2) = B'\_Y \end{array}$$

□

5.4.2 Translation of variable declarations and activation parameters

Variable declarations may occur in GRL programs either as formal parameters, static variables, or temporary variables. Depending on its usage in the GRL program, a variable declaration list will be translated to several LNT constructs. We also consider activation parameters, which are untyped and do not take default values.

Unlike GRL, LNT separates variable declaration and assignment. Thus, GRL variable declarations are first translated to LNT variable declarations. In a second step, default values of parameters (resp. initialisation values of local variables) are assigned to the declared variables.

Additionnally, we will translate GRL variable declarations occurring as formal parameters in highest-level components to LNT gates and channels, in order to enable asynchronous communication in LNT.

Translation to variable declarations Let *dl2var* be a function translating a GRL variable declaration or activation parameter list to an LNT variable declaration list. Activation parameters are translated to LNT parameters of type *block* (see Section 5.2).

$$
\begin{array}{lcl}
dl2var\ (var_1, \ldots, var_n) & = & dl2var\ (var_1), \ldots, dl2var\ (var_n) \\
dl2var\ (X_1, \ldots, X_m : type := E_0) & = & X_1, \ldots, X_m : t2t(type) \\
dl2var\ (X_1, \ldots, X_m : type) & = & X_1, \ldots, X_m : t2t(type) \\
dl2var\ (B_1, \ldots, B_m) & = & B_1, \ldots, B_m : \text{block}
\end{array}
$$

Translation to variable assignments Let *dl2s* be a function translating a GRL variable declaration list to a sequence of LNT assignments.

$$
\begin{array}{lcl}
dl2s\ (var_1, \ldots, var_n) & = & dl2s\ (var_1);\ \ldots;\ dl2s\ (var_n) \\
dl2s\ (X_1, \ldots, X_m : type := E_0) & = & X_1 := e2v\ (E_0);\ \ldots;\ X_m := e2v\ (E_0) \\
dl2s\ (X_1, \ldots, X_m : type) & = & \textbf{null}
\end{array}
$$

Example 5.4. By applying functions *dl2var* and *dl2s*, the left-hand GRL program translates to the right-hand LNT program.

```
1   var X: nat := 0
```

```
1   var X: Nat8 in
2     X := 0
```

□

Sometimes, variable declarations and activation parameters need to be translated to actual parameters, e.g., to encode the internal state or to translate subblock aliasing, where actual input and output parameters are not yet available.

Translation into actual parameters Let *dl2ap* be a function translating either a GRL formal parameter list, a static variable list, or an activation parameter list to an LNT actual parameter list according to its kind in {in, out, block, static}. GRL input and activation parameters are translated to LNT input parameters. GRL output parameters are translated to LNT output parameters. GRL static variable declarations are translated to LNT "**in out**" parameters. Note that *dl2ap* is not defined for receive and send parameters, since it is not used to translate highest-level components.

$$
\begin{array}{lcl}
dl2ap\ (var_1,\ \ldots,\ var_n,\ kind) & = & dl2ap\ (var_1,\ kind),\ \ldots,\ dl2ap\ (var_n,\ kind) \\
dl2ap\ (X_1, \ldots, X_m : type := E_0,\ \mathsf{in}) & = & X_1,\ \ldots,\ X_m \\
dl2ap\ (X_1, \ldots, X_m : type,\ \mathsf{in}) & = & X_1,\ \ldots,\ X_m \\
dl2ap\ (X_1, \ldots, X_m : type,\ \mathsf{out}) & = & \mathtt{?}X_1,\ \ldots,\ \mathtt{?}X_m \\
dl2ap\ (B_1, \ldots, B_m,\ \mathsf{block}) & = & B_1, \ldots, B_m \\
dl2ap\ (X_1, \ldots, X_m : type := E_0,\ \mathsf{static}) & = & \mathtt{!?}X_1,\ \ldots,\ \mathtt{!?}X_m
\end{array}
$$

Translation into gate declaration Let *dl2gate_dl* be a function translating a GRL variable declaration list into gate declaration. The declared gates are typed by channels, which are assumed to be declared in the current LNT module. Channel names build upon the GRL types of formal parameters, corresponding to GRL actual channels. The construction of LNT channels will be presented in Section 5.7.

$$dl2gate\_dl\ (vars)\ =\ gate(get\_vars\ (vars))\mathtt{:}\ channel(get\_types\ (vars))$$

where functions *get_vars* and *get_types* (see Section 4.4.1, page 64) return respectively the ordered list of variable identifiers and type identifiers in the variable declaration *vars*.

Example 5.5. Consider the following GRL code excerpt.

```
1  block (in X: nat := 0, out Y1, Y2: bool) is
2     ...
3  end block
```

By applying function *dl2gate_dl*, we have:

$$
\begin{array}{lcl}
dl2gate\_dl\ (\mathbf{in}\ X\mathtt{:}\ \mathbf{nat}\ \mathtt{:=}\ 0) & = & Gate\_X\mathtt{:}\ Chan\_Nat \\
dl2gate\_dl\ (\mathbf{out}\ Y1, Y2\mathtt{:}\ \mathbf{bool}) & = & Gate\_Y1\_Y2\mathtt{:}\ Chan\_Bool\_Bool
\end{array}
$$

□

5.4.3 Translation of actual parameters

GRL actual parameters serve to describe synchronous interactions between subblocks. They will be translated to LNT actual parameters. For unconnected output parameters, additional "dummy" variables should be declared.

Translation into variable declaration Let *unconnected2var* be a function declaring LNT variables for GRL unconnected output parameters in a component instance $\mathcal{C}'$. These parameters are identified by the set $indices(\mathcal{C}', \mathsf{arg}, \mathsf{out}, \mathsf{unconnected})$. For each parameter, an LNT variable is created. The variable name is built using function *variable*. The variable type is fetched in the component definition, using function *type*.

$$
\begin{array}{lcl}
unconnected2var\ (\mathcal{C}') & = & \mathop{\texttt{++}}_{k \in indices(\mathcal{C}', \mathsf{arg}, \mathsf{out}, \mathsf{unconnected})} unconnected2var\ (\mathcal{C}',\ k) \\
unconnected2var\ (\mathcal{C}',\ k) & = & variable\ (\mathcal{C}',\ k) \texttt{:} t2t\ (type\ (\mathcal{C}',\ k)) \text{ where } \mathcal{C}' \rightarrow \mathcal{C}
\end{array}
$$

Translation into actual parameters Let *arg2ap* be a function translating a GRL actual parameter to an LNT one. The translation of parameters of the form "E_0" and "?X_0" is straightforward. For each parameter of the form "_", the default value of the corresponding formal parameter is fetched in the block definition, using function *default*. For each parameter of the form "?_", a variable is assumed to be declared earlier in the caller body, using function *unconnected2var*. Similarly, for parameters of the form "**any** *type*", a variable is assumed to be declared and assigned a value earlier in the caller body.

$$
\begin{array}{lcl}
arg2ap\ (\mathcal{C}',\ arg_1,\ \ldots,\ arg_n) & = & arg2ap\ (\mathcal{C}',\ arg_1,\ 1)\texttt{, } \ldots \texttt{, } arg2ap\ (\mathcal{C}',\ arg_n,\ n) \\
arg2ap\ (\mathcal{C}',\ E_0,\ k) & = & e2v\ (E_0) \\
arg2ap\ (\mathcal{C}',\ \texttt{?}X_0,\ k) & = & \texttt{?}e2v\ (X_0) \\
arg2ap\ (\mathcal{C}',\ \texttt{\_},\ k) & = & default\ (\mathcal{C}, k) \text{ where } \mathcal{C}' \rightarrow \mathcal{C} \\
arg2ap\ (\mathcal{C}',\ \texttt{?\_},\ k) & = & \texttt{?}variable\ (\mathcal{C}', k) \\
arg2ap\ (\mathcal{C}',\ \textbf{any}\ type,\ k) & = & variable\ (\mathcal{C}', k)
\end{array}
$$

Example 5.6. Consider the following GRL code excerpt.

```
1  block Sub (in  X: nat := 0,
2             out Y: bool)
3  is
4    ...
5  end block
```

```
1  Block High ... is
2    alias Sub as Sub1, Sub as Sub2
3    Sub1 (a, ?_);
4    Sub2 (_, ?b)
5  end block
```

By applying function *unconnected2var*, we have:

$$
\begin{array}{l}
unconnected2var\ (Sub1) = Sub1\_Y\texttt{: } \textbf{Bool} \\
\begin{array}{lcllcl}
arg2ap\ (Sub1,\ a, 1) & = & a & arg2ap\ (Sub2,\ \_, 1) & = & 0 \\
arg2ap\ (Sub1,\ \texttt{?\_}, 2) & = & \texttt{?}Sub1\_Y & arg2ap\ (Sub2,\ \texttt{?}b, 2) & = & \texttt{?}b
\end{array}
\end{array}
$$

□

5.4.4 Translation of actual channels

GRL actual channels serve to describe asynchronous communication between highest-level blocks. They will be translated to LNT gate declaration, thus enabling commu-

nication between LNT asynchronous processes. Sometimes, we do not need to declare gates for all the actual channels of GRL components, but only for connected ones. Additionally, GRL actual channels will be translated to variable declarations, to describe the data exchanged on gates. Finally, they will translated to behaviours, including gate instantiations.

Translation into variable declaration Let *chan2var* be a function translating a GRL actual parameter or channel of component instance $\mathcal{C}'$ to an LNT declaration list. The function is defined for all actual parameters, except parameters of the form "_", whose indices are in the set $indices(\mathcal{C}', \mathsf{arg}, \mathsf{in}, \mathsf{unconnected})$. For each GRL actual parameter, the type is fetched in the component definition $\mathcal{C}$, which serves to declare the corresponding LNT variable. The LNT variable has the same name as the GRL one, if any; otherwise, a variable name is created.

$$
\begin{array}{lcl}
chan2var\ (\mathcal{C}',\ \texttt{<}arg_1,\ \ldots,\ arg_n\texttt{>}) & = & chan2var\ (\mathcal{C}',\ arg_1,\ \ldots,\ arg_n) \\
chan2var\ (\mathcal{C}',\ \texttt{?<}arg_1,\ \ldots,\ arg_n\texttt{>}) & = & chan2var\ (\mathcal{C}',\ arg_1,\ \ldots,\ arg_n) \\
chan2var\ (\mathcal{C}',\ arg_1,\ \ldots,\ arg_n) & = & \underset{\substack{k \notin indices(\mathcal{C}', \mathsf{arg}, \mathsf{in}, \mathsf{unconnected}) \\ \wedge\ k \in 1..n}}{\texttt{++}} chan2var\ (\mathcal{C}',\ arg_k,\ k) \\
chan2var\ (\mathcal{C}',\ X,\ k) & = & X\texttt{:}t2t\ (type\ (\mathcal{C}',\ k)) \\
chan2var\ (\mathcal{C}',\ \texttt{?}X,\ k) & = & X\texttt{:}t2t\ (type\ (\mathcal{C}',\ k)) \\
chan2var\ (\mathcal{C}',\ \texttt{?\_},\ k) & = & variable\ (\mathcal{C}',\ k)\texttt{:}t2t\ (type\ (\mathcal{C}',\ k)) \\
chan2var\ (\mathcal{C}',\ \textbf{any}\ type,\ k) & = & variable\ (\mathcal{C}',\ k)\texttt{:}t2t\ (type)
\end{array}
$$

Example 5.7. Consider the following GRL code excerpt.

```
1  block Sub (in X: nat := 0, out Y: bool) is
2    ...
3  end block
4
5  system Main ... is
6    alias Sub as Sub1, Sub as Sub2, Sub as Sub3
7    block list
8      Sub1 (a, ?_),
9      Sub2 (_, ?b),
10     Sub3 (any nat, ?c)
11 end system
```

By applying function *chan2var*, we have:

$$
\begin{array}{lcl}
chan2var\ (Sub1,\ a, 1) & = & a\texttt{:}\ Nat8 \\
chan2var\ (Sub1,\ \texttt{?\_}, 2) & = & Sub1\_Y\texttt{:}\ Bool
\end{array}
\qquad
\begin{array}{lcl}
chan2var\ (Sub2,\ \_, 1) & = & undefined \\
chan2var\ (Sub2,\ \texttt{?}b, 2) & = & b\texttt{:}\ Bool
\end{array}
$$

$$
\begin{array}{lcl}
chan2var\ (Sub3,\ \textbf{any nat}, 1) & = & Sub3\_X\texttt{:}\ Nat8 \\
chan2var\ (Sub3,\ \texttt{?}c, 2) & = & c\texttt{:}\ Bool
\end{array}
$$

□

Translation into gate declaration Let *chan2gate* be a function building an LNT gate name from a GRL actual channel. For connected channels, whose indices are in $indices(\mathcal{C}', \mathsf{chan}, \mathsf{connected})$, the gate name builds upon the names of the variables composing the channel. For other channels, the gate name builds upon the names of the formal parameters corresponding to the GRL actual channel.

$$chan2gate\ (\mathcal{C}',\ chan_k) = \begin{cases} gate(get\_vars(chan_k)) & \text{if } k \in indices(\mathcal{C}', \mathsf{chan}, \mathsf{connected}) \\ gate(get\_vars(vars_k)) & \text{otherwise} \end{cases}$$

where $vars_k$ is the formal parameter list corresponding to $chan_k$

Let *chan2gate_dl* be a function declaring typed LNT gates for GRL actual channels of component instance $\mathcal{C}'$. Those gates are typed by channels, which are assumed to be declared in the current LNT module[2].

$chan2gate\_dl\ (\mathcal{C}',\ chan_k) = chan2gate\ (\mathcal{C}',\ chan_k)$: $channel(get\_types(vars_k))$
where $vars_k$ is the formal parameter list corresponding to $chan_k$

Example 5.8. Consider the GRL code excerpt given in Example 5.7. By applying *chan2gate_dl*, we have:

chan2gate_dl (*Sub1*, *a*)	=	*Gate_a*: *Chan_Nat8*
chan2gate_dl (*Sub1*, ?_)	=	*Gate_Sub1_Y*: *Chan_Bool*
chan2gate_dl (*Sub2*, _)	=	*Gate_Sub2_X*: *Chan_Nat8*
chan2gate_dl (*Sub2*, ?*b*)	=	*Gate_b*: *Chan_Bool*
chan2gate_dl (*Sub3*, **any nat**)	=	*Gate_Sub3_X*: *Chan_Nat8*
chan2gate_dl (*Sub3*, ?*c*)	=	*Gate_c*: *Chan_Bool*

where the LNT channels *Chan_Nat8* and *Chan_Bool* are defined as follows:

channel *Chan_Nat8* **is** (*Nat8*) **end channel**
channel *Chan_Bool* **is** (*Bool*) **end channel**

□

Let *connected2gate* and *connected2gate_dl* be the variants of functions *chan2gate* and *chan2gate_dl* for only connected channels. These channels are identified by the set $indices(\mathcal{C}', \mathsf{chan}, \mathsf{connected})$.

$$connected2gate\ (\mathcal{C}') = \mathop{\text{++}}_{k \in indices(\mathcal{C}', \mathsf{chan}, \mathsf{connected})} chan2gate\ (\mathcal{C}',\ chan_k)$$
$$connected2gate\_dl\ (\mathcal{C}') = \mathop{\text{++}}_{k \in indices(\mathcal{C}', \mathsf{chan}, \mathsf{connected})} chan2gate\_dl\ (\mathcal{C}',\ chan_k)$$

[2] A preprocessing phase collects the ordered type lists used for parameter declaration lists of GRL components in the current module and all imported modules. These type lists serve to build all the channels, defining gate profiles, that will be used in the generated LNT code. Gate profiles must be pairwise distinct, thus ensuring a unique channel for each gate profile.

where $vars_k$ is the formal parameter list corresponding to $chan_k$.

Example 5.9. Consider the GRL code excerpt given in Example 5.7. By applying *connected2gate_dl*, we have:

$$\begin{array}{lcl} \textit{connected2gate\_dl}\ (\textit{Sub1}) & = & \textit{Gate\_a}\texttt{:}\ \textit{Chan\_Nat8} \\ \textit{connected2gate\_dl}\ (\textit{Sub2},\ \texttt{?}b) & = & \textit{Gate\_b}\texttt{:}\ \textit{Chan\_Bool} \\ \textit{connected2gate\_dl}\ (\textit{Sub3},\ \texttt{?}c) & = & \textit{Gate\_c}\texttt{:}\ \textit{Chan\_Bool} \end{array}$$

□

Translation into behaviour Let *chan2b* be a function translating a GRL actual channel into an LNT behaviour. For each GRL connected channel `<`X_1`,` ... `,`X_n`>` or `?<`X_1`,` ... `,`X_n`>`, an LNT gate is assumed to be declared in the caller body, using function *chan2gate_dl*; and variables X_1, ..., X_n are assumed to be declared using function *chan2var*. In such case, function *chan2b* returns an LNT gate instantiation using variables X_1, ..., X_n. For each wildcard channel, variables are assumed to be declared in the caller body, using function *chan2var*. In such case, function *chan2b* assigns a nondeterministically chosen value to each variable. For unconnected channels, the function returns the **null** statement.

$$\begin{array}{lcl} \textit{chan2b}\ (\mathcal{C}',\ \texttt{<}X_1\texttt{,}\ \ldots\ \texttt{,}X_n\texttt{>}) & = & \textit{gate}(X_1, \ldots, X_n)\ \texttt{(!}X_1\texttt{,}\ \ldots\ \texttt{,!}X_n\texttt{)} \\ \textit{chan2b}\ (\mathcal{C}',\ \texttt{?<}X_1\texttt{,}\ \ldots\ \texttt{,}X_n\texttt{>}) & = & \textit{gate}(X_1, \ldots, X_n)\ \texttt{(?}X_1\texttt{,}\ \ldots\ \texttt{,?}X_n\texttt{)} \\ \textit{chan2b}\ (\mathcal{C}',\ \texttt{<}\textbf{any}\ \textit{type}_1\texttt{,}\ \ldots\ \texttt{,}\textbf{any}\ \textit{type}_n\texttt{>}) & = & \textit{variable}\ (\mathcal{C}',\ 1)\ \texttt{:=}\ \textbf{any}\ \textit{t2t}\ (\textit{type}_1)\texttt{;} \\ & & \ldots\texttt{;} \\ & & \textit{variable}\ (\mathcal{C}',\ n)\ \texttt{:=}\ \textbf{any}\ \textit{t2t}\ (\textit{type}_n) \\ \textit{chan2b}\ (\mathcal{C}',\ \texttt{?<\_,}\ \ldots\ \texttt{,\_>}) & = & \textbf{null} \\ \textit{chan2b}\ (\mathcal{C}',\ \texttt{<\_,}\ \ldots\ \texttt{,\_>}) & = & \textbf{null} \end{array}$$

Example 5.10. Consider the GRL code excerpt given in Example 5.7. By applying *chan2b*, we have:

$$\begin{array}{lcllcl} \textit{chan2b}\ (\textit{Sub1},\ a) & = & \textit{Gate\_a}\ \texttt{(}a\texttt{)} & \textit{chan2b}\ (\textit{Sub2},\ \texttt{?}b) & = & \textit{Gate\_b}\ \texttt{(}b\texttt{)} \\ \textit{chan2b}\ (\textit{Sub1},\ \texttt{?\_}) & = & \textbf{null} & \textit{chan2b}\ (\textit{Sub3},\ \textbf{any nat}) & = & \textit{Sub3\_X}\ \texttt{:=}\ \textbf{any}\ \textit{Nat8} \\ \textit{chan2b}\ (\textit{Sub2},\ \texttt{\_}) & = & \textbf{null} & \textit{chan2b}\ (\textit{Sub3},\ \texttt{?}c) & = & \textit{Gate\_c}\ \texttt{(}c\texttt{)} \end{array}$$

□

5.4.5 Construction of the internal state

To illustrate how the internal state of GRL components is built, we consider the following running example:

```
1  block Sub ... is
2    static var X: bool := false
3    ...
4  end block
```

```
1  block High ... is
2    alias Sub as Sub
3    static var X: bool := false
4    ...
5  end block
```

The state of a GRL system is composed of the internal states of its highest-level components, static variables being syntactically prohibited in systems. The internal state of a component $\mathcal{C}$ is defined by the component static variables concatenated with those of its subblocks, transitively. Each instance $\mathcal{C}'$ of $\mathcal{C}$ has its own copy of the component internal state.

Intuitively, we implement the internal state in LNT by means of local variables, which we call *state variables*. These variables are declared inside the LNT processes corresponding to the GRL highest-level components. To allow state variables to be read and updated by LNT functions corresponding to GRL subblocks, they are propagated through **in out** parameters to those functions transitively.

Formally, to define the translation of the internal state, we need the following functions:

– Function *static* concatenates the declaration lists of a component static variables. For example, if a component $\mathcal{C}$ defines the declaration lists: "**static var** $vars_1$ **,** ... **,** **static var** $vars_n$" , then $static\ (\mathcal{C}) = vars_1$ **,** ... **,** $vars_n$. Contrarily, if $\mathcal{C}$ defines no static variables, then $static\ (\mathcal{C}) = \epsilon$. For blocks *Sub* and *High*, we have:

$$\begin{array}{lcl} static\ (Sub) & = & X\text{: } \textbf{bool := false} \\ static\ (High) & = & X\text{: } \textbf{bool := false} \end{array}$$

– Function *build_state* builds the internal state of a component instance $\mathcal{C}'$. Concretely, the function creates a copy of the internal state of $\mathcal{C}$, where $\mathcal{C}' \rightarrow \mathcal{C}$, in which each variable is given a new unique name. This prevents name clashes, e.g., occurring when several components, used in the same context, define static variables with identical names.

$$\begin{array}{ll} build\_state\ (\mathcal{C}',\ vars_1,\ \ldots,\ vars_n) & = build\_state\ (\mathcal{C}',\ vars_1)\textbf{ , } \ldots \textbf{ , } build\_state\ (\mathcal{C}',\ vars_n) \\ build\_state\ (\mathcal{C}',\ var_1,\ \ldots,\ var_n) & = build\_state\ (\mathcal{C}',\ var_1)\textbf{ , } \ldots \textbf{ , } build\_state\ (\mathcal{C}',\ var_n) \\ build\_state\ (\mathcal{C}',\ X_1,\ \ldots,\ X_n\text{:}type\ \textbf{:=}\ E) & = variable(\mathcal{C}',\ X_1)\textbf{, } \ldots \textbf{ , } variable(\mathcal{C}',\ X_m)\text{:}type\ \textbf{:=}\ E \end{array}$$

For subblock *Sub*, we have: $build\_state\ (Sub, X\text{: } \textbf{bool := false}) = Sub\_X\text{: } \textbf{bool := false}$

The function will be used only for subblocks, since at highest-level components, all subblock variables have already been renamed, transitively.

– Function *get_state* builds the internal state of a component $\mathcal{C}$ upon the internal states of its subblocks. To this aim, a recursive descent is done through the component subblocks and their static variable lists are synthesised in a bottom-up

way[3].

$$get\_state\ (\mathcal{C}) \ = \ static\ (\mathcal{C})\ \texttt{++} \underset{\substack{B' \in sub(\mathcal{C}) \\ \wedge\ B' \rightarrow B}}{\texttt{++}} build\_state\ (B',\ get\_state\ (B))$$

For blocks *Sub* and *High*:

get_state (*Sub*) = X: **bool := false**
get_state (*High*) = X: **bool := false**, *Sub_X*: **bool := false**

In the sequel, we present the translation functions from GRL behavioural constructs to LNT ones. For a concise presentation of functions, we will consider that each component has one formal parameter of each mode accepted by its syntax, one static (resp. temporary) variable list, and one subblock. The generalisation into 0 or n ($n > 0$) parameter lists, variable lists, and subblocks is straightforward.

5.5 Translation of blocks

The section is organised as follows. We start by presenting the translation of GRL block definitions into LNT functions. Afterwards, we give the translation of subblocks and highest-level blocks.

5.5.1 Block definition

The translation function, named *b2f*, from block definitions to LNT functions is given in Table 5.2. It uses functions *dl2var* and *dl2ap* (see Section 5.4.2), function *get_state* (see Section 5.4.5), and function *i2s*.

Blocks defined by the user They are translated to several LNT functions (see (a) in Table 5.2):

- an LNT function having the same name as the block, called *definition function*.
- and an LNT function for each subblock aliasing inside the block. This is done using function *a2f*, whose definition will be given in Section 5.5.2.

The translation of the block body is straightforward since GRL deterministic statements are inspired by LNT ones. Hence, the LNT statement implements one block step, computing outputs from inputs.

Each GRL constant, input, and receive parameter is translated to an LNT input parameter, using function *dl2var*. Similarly, each GRL output and send parameter is translated to an LNT output parameter. Default values of GRL formal parameters do not appear in

[3]This can be performed statically, since the number of GRL components is finite and known. GRL and LNT compilers forbid the dynamic creation of components to enable enumerative verification.

$b2f\left(\begin{array}{l}\textbf{block } B \texttt{ \{}vars_c\texttt{\}} \\ \quad\quad \textbf{(in } vars_i\textbf{,} \\ \quad\quad \textbf{ out } vars_o\textbf{)} \\ \quad\quad \texttt{[}\textbf{receive } vars_r\textbf{,} \\ \quad\quad \textbf{ send } vars_s\texttt{]} \\ \textbf{is} \\ \quad \textbf{alias } B_0 \texttt{ \{}args_0\texttt{\}} \textbf{ as } B_0' \\ \quad \textbf{static var } vars_{sv}\textbf{,} \\ \quad \textbf{var } vars_v \\ \quad\quad\quad I \\ \textbf{end block}\end{array}\right)$	$=$	$\begin{array}{l}\textbf{function } B \textbf{ (in } \quad dl2var\ (vars_c)\textbf{,} \\ \quad\quad\quad\quad\quad\quad \textbf{in } \quad dl2var\ (vars_i)\textbf{,} \\ \quad\quad\quad\quad\quad\quad \textbf{out } dl2var\ (vars_o)\textbf{,} \\ \quad\quad\quad\quad\quad\quad \textbf{in } \quad dl2var\ (vars_r)\textbf{,} \\ \quad\quad\quad\quad\quad\quad \textbf{out } dl2var\ (vars_s)\textbf{,} \\ \quad\quad\quad\quad\quad\quad \textbf{in out } dl2var\ (get\_state(B))\textbf{)} \\ \textbf{is} \\ \quad \textbf{var } dl2var\ (vars_v) \textbf{ in} \\ \quad\quad dl2s\ (vars_v)\texttt{;} \\ \quad\quad i2s\ (I) \\ \quad \textbf{end var} \\ \textbf{end function} \\ \\ a2f(B_0\texttt{[}args_0\texttt{]} \textbf{ as } B_0',\ B)\end{array}$	(a)
$b2f\left(\begin{array}{l}\textbf{block } B \texttt{ \{}vars_c\texttt{\}} \\ \quad\quad \textbf{(in } vars_i\textbf{, out } vars_o\textbf{)} \\ \textbf{is} \\ \quad \texttt{!}\textbf{c}\ \texttt{string} \\ \textbf{end block}\end{array}\right)$	$=$	$\begin{array}{l}\textbf{function } B \textbf{ (in } \quad dl2var\ (vars_c)\textbf{,} \\ \quad\quad\quad\quad\quad\quad \textbf{in } \quad dl2var\ (vars_i)\textbf{,} \\ \quad\quad\quad\quad\quad\quad \textbf{out } dl2var\ (vars_o)\textbf{)} \\ \textbf{is} \\ \quad \texttt{!}\textbf{implementedby } \texttt{"string\%}i\texttt{"} \\ \quad \texttt{!}\textbf{external} \\ \quad \textbf{null} \\ \textbf{end function}\end{array}$	(b)
$b2f\left(\begin{array}{l}\textbf{block } B \texttt{ \{}vars_c\texttt{\}} \\ \quad\quad \textbf{(in } vars_i\textbf{, out } vars_o\textbf{)} \\ \textbf{is} \\ \quad \texttt{!}\textbf{lnt}\ \texttt{string} \\ \textbf{end block}\end{array}\right)$	$=$	$\begin{array}{l}(*\ LNT\ file\ *) \\ \textbf{function } B \textbf{ (in } \quad dl2var\ (vars_c)\textbf{,} \\ \quad\quad\quad\quad\quad\quad \textbf{in } \quad dl2var\ (vars_i)\textbf{,} \\ \quad\quad\quad\quad\quad\quad \textbf{out } dl2var\ (vars_o)\textbf{)} \\ \textbf{is} \\ \quad \texttt{string (}dl2ap\ (vars_c,\ \texttt{in})\texttt{,} \\ \quad\quad\quad\quad dl2ap\ (vars_i,\ \texttt{in})\texttt{,} \\ \quad\quad\quad\quad dl2ap\ (vars_o,\ \texttt{out})\texttt{)} \\ \textbf{end function}\end{array}$	(c)

Table 5.2: Translation functions of block definition
(a) blocks defined by the user (b) blocks defined by external C code (c) blocks defined by external LNT code

the LNT function. They are deferred to the translation of block aliasing and invocation, where those values are useful.

GRL temporary variables are translated to LNT local variables. Such LNT variables are first declared, using function *dl2var*, then assigned to their initialisation values, using function *dl2s*. GRL static variables cannot be translated similarly, since local variables loose their values between subsequent executions of the function. Rather, the static variables of B, and the internal state of subblock B_0', are first captured using function *get_state*; then, they are translated to LNT **in out** parameters, using functions *dl2var*. This way, the internal state of B is stored by the caller, giving function B the ability of read and update.

Example 5.11. In the following, blocks *Foot* and *Dummy* translate to the LNT functions *Foot* and *Dummy*.

GRL blocks	LNT functions

`block Foot {C: nat16 := 1}`
`  (in I: nat16 := 0,`
`  out O: nat16)`
`is`
`  O := I + C`
`end block`
	`function Foot (in C: Nat16,`
	`  in I: Nat16,`
	`  out O: Nat16)`
	`is`
	`  O := I + C`
	`end function`

`block Dummy (in I: nat) is`
`  static var X: nat := 0`
`  var Y: nat := 0`
`  X := X + I;`
`  Y := Y + I`
`end block`
	`function Dummy (in I: Nat8,`
	`  in out X: Nat8)`
	`is`
	`  var Y: Nat8 in`
	`  Y := 0;`
	`  X := X + I;`
	`  Y := Y + I`
	`  end var`
	`end function`

□

Blocks defined by external C code Their translation relies on the capability of LNT to import external C code. In compliance with the reference manual of the Lnt2Lotos compiler [CCG+16], a block defined by external C code is translated to:

- an LNT function having the same name as the block (see (b) in Table 5.2). Its formal parameters are obtained as explained in the previous paragraph. The LNT function B encapsulates the generated C code, by using pragmas **!implementedby** and **!external**. The **!external** pragma indicates the use of external C functions in the generated LNT module, in which case the function body is necessarily **null**. The **!implementedby** pragma gives the precise name of the function to be used by the back-end compilers.
- an interface C function for each output of the GRL block returning the output value. The name of each function is determined by expanding $\%i$ with the output position in function B. The interface C functions are produced in a file with suffix ".fnt". We do not give a formal definition of this translation.

Example 5.12. Consider the following block *C_Shift* defined by the external C function *Shift*.

```
-- GRL file importing the C file
block C_Shift (in num :int16, out left, right :int16)
 is
   !c "Shift"
end block
```

```
// C file
void Shift (GRL_Int16 num, GRL_Int16* left, GRL_Int16* right)
{// convert types to the C domain
```

```
4    unsigned char arg_number = GRL_Int16_To_Signed_Char (num);
5   // compute outputs and reconvert types to the GRL domain
6    *left  = GRL_Signed_Char_To_Int16 (arg_number << arg_bits);
7    *right = GRL_Signed_Char_To_Int16 (arg_number >> arg_bits);
8   }
```

The translation generates the LNT function *C_Shift* and two interface C functions *Shift1* and *Shift2*, returning the respective values of outputs *left* and *right*.

```
1   -- LNT file
2   function C_Shift (in  num   : Int16,
3                     out left  : Int16,
4                     out right : Int16)
5   is
6      !implementedby "Shift%i"
7      !external
8      null
9   end function
```

```
1   // FNT file
2   GRL_Int16 Shift1 (GRL_INT16 num)
3   {
4      GRL_Int16 left; GRL_Int16 right;
5      Shift (num, &left, &right);
6      return left;
7   }
8   GRL_Int16 Shift2 (GRL_INT16 num)
9   {
10     GRL_Int16 left; GRL_Int16 right;
11     Shift (num, &left, &right);
12     return right;
13  }
```

□

Blocks defined by external LNT code Their translation is straightforward (see (c) in Table 5.2). A block defined by external LNT code is translated to an LNT function having the same name as the block. The function body consists of a call to the *external* LNT function.

Example 5.13. The following GRL block *LNT_Foot*, defined by an external LNT function named *Foot*, translates to the following LNT function.

```
1   -- GRL code
2   block LNT_Foot (in  C: nat16,
3                   in  I: nat16,
4                   out O: nat16)
5   is
6      !lnt "Foot"
7   end block
```

```
1   -- LNT code
2   function LNT_Foot (in  C: Nat16,
3                      in  I: Nat16,
4                      out O: Nat16)
5   is
6      Foot (C, I, ?O)
7   end function
```

□

5.5.2 Subblock aliasing and invocation

We consider the translation of the following subblock aliasing and invocation, where $args_i$ and $args_o$ denote actual input and output parameter lists, respectively:

alias B {$args_c$} **as** B'
B' ($args_i$, $args_o$)

We write $vars_i$ and $vars_o$ for the formal parameter lists corresponding to $args_i$ and $args_o$ in the definition of block B.

The translation functions of subblock aliasing and invocation are given in Table 5.3. They use functions *dl2var* and *dl2ap* (see Section 5.4.2), functions *build_state* and *get_state* (see Section 5.4.5), and function *unconnected2var* (see Section 5.4.3).

$a2f\big(\ B\{args_c\}$ **as** $B'\ \big) =$	**function** B' **(in** *dl2var* $(vars_i)$**,** **out** *dl2var* $(vars_o)$**,** **in out** *dl2var* $(build\_state(B',\ get\_state(B)))$ **is** **eval** B **(***arg2ap* $(B', args_c)$**,** *dl2ap* $(vars_i,$ in**),** *dl2ap* $(vars_o,$ out**),** *dl2ap* $(build\_state(B',\ get\_state(B)),$ static**))** **end function**
$i2s\big(\ B'$ **(***args*$_i$**,** *args*$_o$**)** $\big) =$	**var** *unconnected2var* (B') **in** **eval** B' **(***arg2ap* $(B',\ args_i)$**,** *arg2ap* $(B',\ args_o)$ *dl2ap* $(build\_state(B',\ get\_state(B)),$ static**))** **end var**

Table 5.3: Translation functions of subblock aliasing ($a2f$) and invocation ($i2s$)

Subblock aliasing is translated to an LNT function, called *aliasing function*, using the translation function *a2f*. The aliasing function has the same name[4] as the GRL subblock and declares:

- input and output parameters corresponding to those of block B. These parameters will be used later to interact with functions corresponding to other subblocks.
- **in out** parameters, which are the state variables implementing the subblock internal state. This way, state variables can be synthesised in a bottom-up manner up to the highest-level component.

Inside the aliasing function, the definition function named B is called with the declared state variables and with the actual constant parameters of the subblock. In particular, actual constant parameters are available at aliasing time and not used for interactions. Hence, they are irrelevant when calling the aliasing function.

Example 5.14. Consider the following aliasing of block *Foot*.

```
1   alias Foot {2} as Large, Foot {_} as Small
```

[4]To simplify the presentation of the translation functions, we consider that LNT functions and processes corresponding to GRL component instances have the same names as their GRL components. In practice, a unique name is given to each generated LNT component. This prevents name clashes, e.g., occurring when a subblock is aliased with the same name in different components.

The translation generates the following functions:

```
1  function Large (in  I: nat16,
2                  out O: nat16)
3  is
4     Foot (2, I, ?O)
5  end function
```

```
1  function Small (in  I: Nat16,
2                  out O: Nat16)
3  is
4     -- default value of the constant
5     -- parameter is passed
6     Foot (1, I, ?O)
7  end function
```

□

A subblock invocation is translated to a call to the LNT aliasing function[5], using function *i2s* (see Table 5.3). Unconnected output parameters are translated to auxiliary variables, which are declared, using function *unconnected2var*, and passed to the function call, using function *dl2ap*. See Section 5.4.3 for details about the translation of actual parameters.

Example 5.15. The left-hand GRL block invocations below translate to the right-hand LNT code. In particular, subblock *Foot* is invoked at line 1 without being aliased. Consequently, an LNT function named *Foot_165* is generated by the translation.

```
1  -- GRL code
2  Foot (_, ?X); -- not aliased before
3  Small (X, ?Not_X);
4  Large (Not_X, ?_)
```

```
1  -- LNT code
2  eval Foot_165 (0, ?X);
3  eval Small (X, ?Not_X);
4  var Dummy_O: Bool in
5    eval Large (Not_X, ?Dummy_O)
6  end var
```

□

Remark 5.1. The synchronous assumptions are granted for free in the translation of a single GRL block. LNT functions are deterministic and execute atomically without producing transitions. This coincides with the assumption that computations and data processing are instantaneous in synchronous components. ■

5.5.3 Highest-level block aliasing and invocation

We consider the translation of the following highest-level block aliasing and invocation, where $chan_i$, $chan_o$, $chan_r$, $chan_s$ denote respectively actual input, output, receive and send channels:

alias B {$args_c$} **as** B'
B' ($chan_i$, $chan_o$)[$chan_r$, $chan_s$]

[5]For subblocks invoked without being aliased, the translation automatically generates aliasing functions, whose names are not always user-friendly.

The translation functions, given in Table 5.4, use functions *dl2var*, *dl2s*, *dl2ap* (see Section 5.4.2), functions *chan2var*, *arg2ap*, *connected2gate*, *connected2gate_dl*, *chan2b* (see Section 5.4.3), and functions *build_state* and *get_state* (see Section 5.4.5).

$b2p($ **alias** B {$args_c$} **as** B' $) =$

process B' [*connected2gate_dl* (B'),
Start:Block, Finish:**none**]
is
var *chan2var* (B', $args_c$),
chan2var (B', $chan_i$),
chan2var (B', $chan_o$),
chan2var (B', $chan_r$),
chan2var (B', $chan_s$),
dl2var (*build_state*(B', *get_state*(B)))
in
dl2s (*build_state*(B', *get_state*(B)));
loop
Start (B');
chan2b (B', $chan_r$);
chan2b (B', $chan_i$);
eval B (*arg2ap* (B', $args_c$),
arg2ap (B', $chan_i$),
arg2ap (B', $chan_o$),
arg2ap (B', $chan_r$),
arg2ap (B', $chan_s$),
dl2ap (*build_state*(B', *get_state*(B)), static));
chan2b (B', $chan_o$);
chan2b (B', $chan_s$);
Finish
end loop
end var
end process

$b2b($ B' ($chan_i$, $chan_o$)[$chan_r$, $chan_s$] $) =$ B' [*connected2gate* (B'), Start, Finish]

Table 5.4: Translation functions of highest-level block aliasing and invocation

Highest-level block aliasing

The aliasing of B' is translated to an LNT wrapper process, using the translation function *b2p*. The process encapsulates the definition function named B to interface it with other processes. It receives values from processes, invokes function B with those values, and emits the values returned by the function to processes. This requires to translate GRL actual channels into both:

– gate communication to enable value exchange with other processes. Gates build upon the GRL actual channels of blocks, available at invocation time, and not upon formal parameters as for subblocks. This enables to translate actual channels depending on their form, using function *connected2gate_dl*. Since GRL unconnected and wildcard channels are unused in communications, the code is optimised by not generating useless transitions. Additional gates *Start* and *Finish* are declared, the usage of which will be given later.

– local variables, using function *chan2var*, to represent the values exchanged on gates. An exception is unconnected input and receive channels, whose values are fetched in the block definition and passed to the LNT function call (see function *arg2ap*, Section 5.4.3).

The wrapper process defines an infinite loop, which implements the implicit synchronous loop of GRL highest-level blocks. Each loop iteration defines a step of the block. The execution of the loop starts by computing input and receive values for the current step, using function *chan2b*. Only for connected channels, a gate is instantiated to receive values from other processes which are stored in dedicated local variables. Variables corresponding to wildcard parameters are assigned to nondeterministically chosen values. Then, those variables are passed as actual input parameters to function B. The function provides actual output parameters, among which only parameters corresponding to GRL connected channels are emitted through subsequent gates.

Locking mechanism A loop iteration of a wrapper process instantiates sequentially several gate communications. This corresponds to a sequence of transitions in the generated LTS, each gate corresponding to a transition. Such sequences should be atomic, i.e., individual sequences of transitions corresponding to different blocks should not interleave, thus preserving the atomicity of block steps. For this purpose, we introduce a locking mechanism. Additional gate communications, *Start* and *Finish*, are added at the beginning and end of each process loop iteration, respectively. These gates enable the process to synchronise with an additional process *Mutex*, defined as follows:

```
process Mutex [Start: Block, Finish: none] is
  loop
    Start (?any block);– Only the process named "block" can execute
    Finish
  end loop
end process

channel Block is – introduced by the translation
  (block) – type enumerating the names of highest-level blocks, including B'
end channel
```

This way, gate *Start* starts the gate communication sequence in process B' by acquiring the lock and gate *Finish* finishes it by releasing the lock, without interleaving with gate communications of other processes in between. More details about synchronisations between LNT processes will be given in Section 5.7.

Example 5.16. Consider the aliasing of block *Foot* in the left-hand side of the code below. Since the translation of block aliasing relies also on the block invocation, we present the invocation of block *Foot*. The translation generates the LNT process in the right-hand side.

```
system S2 ... is
  -- aliasing
   alias Foot {2} as Large
   ...
  -- invocation
   Large (I, ?O)
end system
```

```
process S2_Large [Gate_I: Chan_nat16,
                  Gate_O: Chan_nat16,
                  Start : Block,
                  Finish: None] is
  var I: Nat16, O: Nat16 in
    loop
      Start  (Large);
      Gate_I (?I);
      eval Foot (2, I, ?O);
      Gate_O (O);
      Finish
    end loop
  end var
end process
```

□

Example 5.17. The following aliasing of block *Foot* (left-hand side) translates to the following LNT process (right-hand side). No gate in the LNT process is associated to the GRL unconnected input.

```
system S3 ... is
  -- aliasing
   alias Foot {2} as Large
  -- invocation
   Large (_, ?O)
end system
```

```
process S3_Large [Gate_O: Chan_nat16,
                  Start : Block,
                  Finish: None]
is
  var O: Nat16 in
    loop
      Start  (Large);
      -- value 0 fetched in block definition
      eval Foot (2, 0, ?O);
      Gate_O (O);
      Finish
    end loop
  end var
end process
```

□

Translation of the internal state Contrarily to the translation of subblocks, the internal state of highest-level blocks is implemented using LNT local variables (as anticipated in Section 5.4.5). These *state variables* are declared, using function *dl2var*, and initialised, using function *dl2s*, before starting the synchronous loop. State variables are passed as **in out** parameters to the encapsulated function, thus propagated to functions corresponding to subblocks, transitively. This way, each loop iteration of a wrapper process starts by reading the values of state variables stored in the previous iteration of the loop and finishes by updating those values.

Remark 5.2. Static variables could not have been translated in a modular way,

i.e., independently of the call context of blocks. The corresponding LNT state variables should be defined outside the synchronous loop of highest-level blocks. However, it is worth noticing that the support of **in out** parameters by LNT enables an elegant and controllable implementation of the state notion while keeping a functional flavour. ■

Example 5.18. Consider the following aliasing and invocation of block *Dummy* inside a system *S1*.

```
1  system S1 ... is
2    alias Dummy as Dummy
3    ...
4    Dummy (I)
5  end system
```

The translation generates the following wrapper process:

```
1  process S1_Dummy [Gate_I: Chan_nat, Start: Block, Finish: none] is
2    var Dummy_X: Nat8,              -- internal state declaration
3        I      : Nat8
4    in
5      Dummy_X := 0;                 -- internal state initialisation
6      loop
7        Start (Dummy);
8        GATE_I (?I);
9        eval Dummy (I, !?Dummy_X); -- internal state read and update
10       Finish
11     end loop
12   end var
13 end process
```

□

Highest-level block invocation

Finally, each highest-level block invocation is translated to the invocation of the corresponding wrapper process. This is done by using function *b2b* (see Table 5.4).

Example 5.19. The following GRL block invocations translate to the following LNT process invocations.

```
1  -- GRL code          1  -- LNT code
2  Dummy (I)            2  S1_Dummy [Gate_I, Start, Finish]
3  Large (I, O)         3  S2_Large [Gate_I, Gate_O, Start, Finish]
4  Large (_, O)         4  S3_Large [Gate_O, Start, Finish]
```

□

5.6 Translation of environments and mediums

This section is organised as follows. We first present the translation of signals. Then, we present the translation of environments and mediums.

5.6.1 Signals

Signals are translated using function *i2s* as given in Table 5.5. It uses function *gate* (see Section 5.4.1).

$i2s($ **when** ?<X_0 , ... , X_n> -> I_0 $)$ =	$gate(X_0, \ldots, X_n)$ (?X_0 , ... , ?X_n); $i2s(I_0)$	reception data signal
$i2s($ **when** <X_0 , ... , X_n> -> I_0 $)$ =	$i2s(I_0)$; $gate(X_0, \ldots, X_n)$ (!X_0 , ... , !X_n)	emission data signal
$i2s($ **enable** B_0 $)$ =	Start (B_0)	activation signal

Table 5.5: Translation of signals

Since GRL data signals are communication primitives enabling environments and mediums to exchange values with blocks, their translation involves value-passing synchronisations. The reception data signal "**when** ?<X_0 , ... , X_n> -> I_0" is translated to a gate waiting for value reception on variables $X_0, \ldots, X_n$, followed by behaviour $i2s(I_0)$. The emission data signal "**when** <X_0 , ... , X_n> -> I_0" is translated to behaviour $i2s(I_0)$ followed by a gate emitting values on variables $X_0, \ldots, X_n$.

Since GRL activation signals aim to constrain highest-level block activation, their translation should exploit gate *Start*, introduced by the locking mechanism. Hence, an activation signal "**enable** B_0" is translated to a gate communication "Start (B_0)". This translation enables three-party synchronisation on gate *Start* between:

1. the wrapper process named B_0
2. process *Mutex*
3. the process containing "Start (B_0)"

Therefore, a process of a block can acquire the Mutex only if (i) the Mutex is acquired by no other process and (ii) a process corresponding to a GRL environment proposes a synchronisation on gate *Start* with the block name. Since synchronisations in LNT are blocking, process B_0 will wait for other processes to be ready on gate *Start*. If the gate is unreachable in some process, then process B_0 will not execute, which is in accordance with GRL semantics.

5.6.2 Environments

The translation functions of environment definition, aliasing, and invocation are given in Table 5.6. They use functions *dl2var*, *dl2s*, *dl2ap* (see Section 5.4.2), functions *chan2var*, *arg2ap*, *chan2gate*, *dl2gate_dl*, *chan2b* (see Section 5.4.3), and functions *build_state* and *get_state* (see Section 5.4.5).

Environment definition An environment is translated to an LNT process, called *definition process*, using function *n2p*. Because GRL environments support nondeter-

$$n2p\left(\begin{array}{l}\textbf{environment } N\\ \quad \{vars_c\}\\ \quad (\textbf{in } vars_i,\\ \quad\ \textbf{out } vars_o,\\ \quad\ \textbf{block } blocks)\\ \textbf{is}\\ \quad \textbf{alias } B_0\ \{args'_c\}\ \textbf{as } B'_0\\ \quad \textbf{static var } vars_{sv}\\ \quad \textbf{var } vars_v\\ \quad\ I\\ \textbf{end environment}\end{array}\right) = \begin{array}{l}a2f\ (\textbf{alias } B_0\ \{arg'_c\}\ \textbf{as } B'_0)\\ \\ \textbf{process } N\ [dl2gate\_dl(vars_i),\\ \qquad\qquad dl2gate\_dl(vars_o),\\ \qquad\qquad \text{Start: Block}]\\ \qquad\qquad (\textbf{in } dl2var\ (vars_c),\\ \qquad\qquad\ \textbf{in } dl2var\ (blocks),\\ \qquad\qquad\ \textbf{in out } dl2var\ (get\_state(N)))\\ \textbf{is}\\ \quad \textbf{var } dl2var\ (vars_v)\ \textbf{in}\\ \quad\ dl2s\ (vars_v);\\ \quad\ i2s\ (I)\\ \quad \textbf{end var}\\ \textbf{end process}\end{array}$$

$$a2p(\ \textbf{alias } N\ \{args_c\}\ \textbf{as } N'\ ) = \begin{array}{l}\textbf{process } N'\ [chan2gate\_dl\ (N', chan_i),\\ \qquad\qquad chan2gate\_dl\ (N', chan_o),\\ \qquad\qquad \text{Start: Block}]\\ \textbf{is}\\ \quad \textbf{var}\\ \quad\ chan2var\ (N', args_c),\\ \quad\ chan2var\ (N', args_b),\\ \quad\ dl2var\ (build\_state(N',\ get\_state(N)))\\ \quad \textbf{in}\\ \quad\ dl2s\ (build\_state(N',\ get\_state(N)));\\ \quad\ \textbf{loop}\\ \quad\quad N\ [gate\ (chan_i),\ gate\ (chan_o),\ \text{Start}]\\ \quad\quad\ (arg2ap\ (N', args_c),\\ \quad\quad\ arg2ap\ (N', args_b),\\ \quad\quad\ dl2ap\ (build\_state(N',\ get\_state(N)),\ \mathsf{static}))\\ \quad\ \textbf{end loop}\\ \quad \textbf{end var}\\ \textbf{end process}\end{array}$$

$$n2b(\ N'\ (chan_i,\ chan_o,\ args_b)\ ) = N'\ [chan2gate\ (chan_i),\ chan2gate\ (chan_o),\ \text{Start}]$$

Table 5.6: Translation functions of environment definition (*n2p*), aliasing (*a2p*), and invocation (*n2b*)

ministic behaviours and signals, their definition could not be described by LNT functions, as for blocks.

Each GRL constant parameter is translated to an LNT input parameter, using function *dl2var*. Each GRL input and output channel is translated to an LNT typed gate, using function *dl2gate*. Each GRL activation parameter is translated to an LNT input parameter of type block, using function *dl2var*. When the GRL environment defines activation parameters, the corresponding LNT process should also declare gate *Start*. The internal state is translated similarly to blocks.

Example 5.20. The following GRL environments (left-hand side) translate to the following LNT processes (right-hand side).

	GRL environments		LNT processes
1	**environment** Default	1	**process** Default [Start: Block]
2	(**block** B1, B2)	2	(**in** B1, B2: block)
3	**is**	3	**is**
4	**select**	4	**select**
5	**enable** B1	5	Start (B1)
6	**[] enable** B2	6	**[]** Start (B2)
7	**end select**	7	**end select**
8	**end environment**	8	**end process**
		1	**process** Disable [Gate_Cmd: Chan_bool]
1	**environment** Disable	2	**is**
2	(**out** Cmd: **bool**)	3	**var** Cmd: Bool **in**
3	**is**	4	Cmd := **false**;
4	**when** Cmd -> Cmd := **false**	5	Gate_Cmd (!Cmd)
5	**end environment**	6	**end var**
		7	**end process**

□

Environment aliasing and invocation We consider the translation of the following environment aliasing and invocation, where $chan_i$, $chan_o$, $args_b$ denote respectively actual input channel, output channel, and activation parameters:

alias N {$args_c$} **as** N'
N' ($chan_i$, $chan_o$, $args_b$)

Environment aliasing is translated to an LNT wrapper process, using function *a2p* (see Table 5.6). The wrapper process N' encapsulates the definition process named N, similarly to the translation of highest-level blocks. It defines the same set of gates as N, but no formal parameters. Contrarily to the translation of blocks, gates build upon the formal channels of the GRL environment. This is required to call the definition process with appropriate number of actual gates. The execution of the process starts by defining and initialising state variables. Then, it invokes process N inside an infinite loop with:

- input parameters corresponding to GRL actual constant parameters, using function *arg2ap*
- input parameters corresponding to GRL actual activation parameters, using function *dl2ap*
- **in out** parameters corresponding to the internal state of N', using function *dl2ap*

Example 5.21. Consider the following aliasing and invocation of environments *Default* and *Disable* inside a system *S4*.

```
1  system S4 ... is
2    alias Default as Default, Disable as Disable
```

```
3      ...
4      Default (B1, B2),
5      Disable (Cmd)
6    end system
```

The LNT wrapper processes corresponding to those environments are the following.

```
1   process S4_Default [Start: Block]
2   is
3     loop
4       Default [Start] (B1, B2)
5     end loop
6   end process
```

```
1   process S4_Disable [Gate_Cmd: Chan_bool]
2   is
3     loop
4       Disable [GATE_Cmd]
5     end loop
6   end process
```

□

Finally, each environment invocation is translated to the invocation of the corresponding wrapper process. This is done by function *n2b* (see Table 5.6). GRL actual channels are translated to LNT gates, using function *chan2gate*. GRL actual activation parameters are translated to gate *Start*.

Example 5.22. The following GRL environment invocations (left-hand side) translate to the following LNT process invocations (right-hand side).

```
1   -- GRL code
2   Default (B1, B2)
3   Disable (?Cmd)
```

```
1   -- LNT code
2   S4_Default [Start]
3   S4_Disable [Gate_Cmd]
```

□

5.6.3 Mediums

Similarly to environments, each medium definition is translated to an LNT definition process, using a function *m2p*. Each medium aliasing is translated to a wrapper process, encapsulating the definition process, using function *a2p* (see Table 5.6). Each medium invocation is translated to a call to the wrapper process, using a function *m2b*. Functions *m2p* and *m2b* are identical to functions *n2p* and *n2b*, except that input (resp. output) parameters are replaced by receive (resp. send) parameters and translation of activation parameters is omitted.

5.7 Translation of systems

This sections presents the translation of systems. We first present some required sets and auxiliary functions. Then, we define the translation function of systems.

5.7.1 Sets and auxiliary functions

Let S be a system. As in Section 4.7.2, all the system components are assumed to be indexed such that each component $\mathcal{C}' \subset S$ is associated to a unique index. We write

block_invoc, *env_invoc*, and *med_invoc* (resp. *block_alias*, *env_alias*, and *med_alias*) as shorthands for component invocation (resp. aliasing) inside S. We write B'_j (resp. N'_j, M'_j) for the component instance whose aliasing is $block\_alias_j$ and invocation is $block\_invoc_j$ (resp. env_alias_j and env_invoc_j, med_alias_j and med_invoc_j).

We write $indices(S, \mathsf{block})$, $indices(S, \mathsf{env})$, $indices(S, \mathsf{med})$ for the sets of indices of respectively blocks, environments, and mediums, inside S. We write $indices(S, \mathsf{activ})$ for the set of indices of blocks whose activation is constrained by environments; thus, we have the invariant $indices(S, \mathsf{activ}) \subseteq indices(S, \mathsf{block})$.

Each channel used inside S is associated to a unique index. We write $indices(\mathcal{C}', \mathsf{chan})$ for the set of indices of channels used inside $\mathcal{C}'$ invocation.

We define the following sets on channels:

– The set $indices(S, \mathsf{unconnected})$ contains the indices of unconnected channels of environments and mediums, respectively[6].

$$\begin{array}{lcl} indices(S, \mathsf{unconnected}) & = & \bigcup\limits_{k \in indices(S,\mathsf{env}) \cup indices(S,\mathsf{med})} indices(\mathcal{C}'_k, \mathsf{unconnected}) \\ indices(\mathcal{C}'_k, \mathsf{unconnected}) & = & \{j \in indices(\mathcal{C}'_k, \mathsf{chan}) \mid connexion(chan_j) = \mathsf{unconnected}\} \end{array}$$

– The set $indices(S, \mathsf{linked})$ contains the indices of channels that are common between components.

$$\begin{array}{lcl} indices(S, \mathsf{linked}) & = & \bigcup\limits_{k \in indices(S,\mathsf{block})} indices(S, B'_k, \mathsf{linked}) \\ indices(S, B'_k, \mathsf{linked}) & = & \{j \in indices(B'_k, \mathsf{chan}) \mid \\ & & \quad (\exists p \in indices(S, \mathsf{env}) \wedge \mathsf{j} \in indices(N'_p, \mathsf{chan}) \\ & & \quad \vee (\exists q \in indices(S, \mathsf{med}) \wedge \mathsf{j} \in indices(M'_q, \mathsf{chan}) \end{array}$$

– The set $indices(S, \mathsf{unlinked})$ contains the indices of connected channels that are used in exactly one environment or medium. The set uses function *connexion* defined in Section 5.4.1 (page 84).

$$\begin{array}{lcl} indices(S, \mathsf{unlinked}) & = & \bigcup\limits_{k \in indices(S,\mathsf{env}) \cup indices(S,\mathsf{med})} indices(S, \mathcal{C}'_k, \mathsf{unlinked}) \\ indices(S, \mathcal{C}'_k, \mathsf{unlinked}) & = & \{j \in indices(\mathcal{C}'_k, \mathsf{chan}) \mid \\ & & \quad connexion(chan_j) = \mathsf{connected} \ \wedge \ j \notin indices(S, \mathsf{linked})\} \end{array}$$

Additionally, we write $indices(S, \mathsf{visible})$ for the set of channels in S whose variables are visible from the outside[7].

[6] Unconnected channels of blocks are useless here since they have no corresponding LNT gates.

[7] In the formal definition of GRL, visible and hidden variables can compose the same actual channel. Here, for simplicity, we stipulate that the variables composing a GRL channel should be either all visible or all hidden.

5.7.2 Translation function

The translation function, named *s2p*, of systems is given in Table 5.7. It uses functions *a2f* (see Table 5.3), *a2p* (see Table 5.6), *b2b* (see Table 5.4), *n2p* (see Table 5.6), *m2b*, *chan2gate_dl* and *connected2gate_dl* (see Section 5.4.4). A GRL system is translated to:

- the enumerated type *Block*, introduced previously.
- wrapper processes corresponding to blocks, environments, and mediums. These processes will be called block-, environment-, and medium processes, hereafter.
- additional processes generated by the translation, which are process *Mutex*, introduced previously, and a process named *Activation*.
- a root process, encapsulating all the aforementioned processes.

Gates Synchronisation between the encapsulated processes takes place on gates that are declared inside the root process. LNT gates build upon the GRL actual channels used in component invocations, rather than on variable declaration lists of the system. This is because variables are declared individually but grouped to form actual channels only at invocation time.

A gate is either visible or hidden. Visible gates are declared using function *visible* (see (a) in Table 5.7) and correspond to:

- GRL channels of blocks, environments, and mediums whose variables are declared as formal parameters in the GRL system
- gate *Start* to visualise the activation policy of highest-level blocks

Hidden gates are declared using function *hidden* (see (b) in Table 5.7) and correspond to:

- GRL channels whose variables are declared as temporary variables in the GRL system
- gates corresponding to unconnected channels of environments and mediums
- gate *Finish* since it is used only to release the *Mutex* and contains no information about block execution

Parallel composition Inside the root process, wrapper processes corresponding to GRL components are composed using the parallel composition operator **par**. Block processes are composed in pure interleaving. This way, they cannot synchronise with each other, even on their common gates *Start* and *Finish*. Here is an excerpt of the translation function:

par – purely interleaved block processes
  b2b (*block_invoc*$_1$) || ... || *b2b* (*block_invoc*$_p$)
end par

This parallel composition is itself encapsulated inside a higher-level parallel composition to synchronise with process *Mutex* on gates *Start* and *Finish*. This enables individual synchronisation between process *Mutex* and block processes. Here is an excerpt of the translation function:

```
par Start, Finish in   – locking mechanism
  par   – purely interleaved block processes
    b2b (block_invoc_1) || ... || b2b (block_invoc_p)
  end par
  ||
  Mutex [Start, Finish]
end par
```

Similarly to the translation of blocks, environments and mediums processes are composed in pure interleaving. Here is an excerpt of the translation function:

```
par    – purely interleaved environment and medium processes
      n2b (env_invoc_1)   || ... || n2b (env_invoc_q)    – environment processes
  || m2b (med_invoc_1) || ... || m2b (med_invoc_r)   – medium processes
end par
```

Finally, all aforementioned parallel compositions are encapsulated inside a *main* parallel composition. The synchronisation set, defined by function *synch*, contains:

- Gate *Start* to enable synchronisations between: environment processes, block processes, and process *Mutex*.
- Gates corresponding to GRL channels that are common to blocks and environments/mediums. These are gates whose indices are in $indices(S, \mathsf{linked})$.
- Gates corresponding to GRL channels of environments and mediums that are either unconnected or occur in exactly one environment or medium process. These are gates whose indices are in $indices(S, \mathsf{unconnected}) \cup indices(S, \mathsf{unlinked})$. By putting these gates in the synchronisation set, the respective processes will wait infinitely for communication. As a result, the execution paths guarded by the gates are unfeasible, which complies with the semantics of GRL. Note however that the translation causes no blocking situations. Because each execution path of environment and medium processes contains at most one gate, the infinite waiting on a gate has no impact on other gates. Note also that while such gates are declared and used, no additional transition in the corresponding LTS is generated.

Here is an excerpt of the translation function, showing the main parallel composition:

```
par synch(S) in    – main parallel composition
  par Start, Finish in   – purely interleaved block processes with the locking mechanism
  ...
  end par
  ||
  par    – purely interleaved environment and medium processes
  ...
  end par
end par
```

Still, processes corresponding to blocks whose activation is not constrained cannot execute, since gate *Start* occurs in the synchronisation set of the main parallel composition. Such processes will block, waiting infinitely to synchronise on gate *Start* with an environment process. To prevent such undesirable situations, we introduce an additional process, named *Activation*, in parallel with processes corresponding to environments and mediums. Process *Activation* proposes permanently synchronisations on gate *Start* for blocks whose activation is not constrained. This is done by function *activate* (see (c) in Table 5.7).

$$visible\ (S) = \underset{\substack{j \in\ indices(S, \mathsf{block}) \\ \cap indices(S, \mathsf{visible})}}{\texttt{++}} connected2gate\_dl(B'_j), \underset{\substack{j \in\ indices(S, \mathsf{env}) \\ \cap indices(S, \mathsf{visible})}}{\texttt{++}} chan2gate\_dl(N'_j), \underset{\substack{j \in\ indices(S, \mathsf{med}) \\ \cap indices(S, \mathsf{visible})}}{\texttt{++}} chan2gate\_dl(M'_j) \quad \text{(a)}$$

$$hidden\ (S) = \underset{\substack{j \in\ indices(S, \mathsf{block}) \\ \setminus indices(S, \mathsf{visible})}}{\texttt{++}} connected2gate\_dl(B'_j), \underset{\substack{j \in\ indices(S, \mathsf{env}) \\ \setminus indices(S, \mathsf{visible})}}{\texttt{++}} chan2gate\_dl(N'_j), \underset{\substack{j \in\ indices(S, \mathsf{med}) \\ \setminus indices(S, \mathsf{visible})}}{\texttt{++}} chan2gate\_dl(M'_j) \quad \text{(b)}$$

$$activate\ (B'_0, \ldots, B'_n) = \textbf{select}\ \texttt{Start (}B'_0\texttt{)}\ \texttt{[]}\ldots\texttt{[]}\ \texttt{Start (}B'_n\texttt{)}\ \textbf{end select} \quad \text{(c)}$$

$s2p$ (

system S (*vars*$_1$, ... , *vars*$_m$) **is**
  block_alias$_1$, ... , *block_alias*$_p$
  env_alias$_1$, ... , *env_alias*$_q$
  med_alias$_1$, ... , *med_alias*$_r$
  var *vars*$'_1$, ... , *vars*$'_n$
    block list
      block_invoc$_1$,
      , ... ,
      block_invoc$_p$
    environment list
      env_invoc$_1$,
      , ... ,
      env_invoc$_q$,
    medium list
      med_invoc$_1$,
      , ... ,
      med_invoc$_r$
end system

) =

type Block **is** (*B*$'_1$, ... , *B*$'_p$) **end type**

a2f	(*block_alias*$_1$)	...	*a2f*	(*block_alias*$_p$)
a2p	(*env_alias*$_1$)	...	*a2p*	(*env_alias*$_q$)
a2p	(*med_alias*$_1$)	...	*a2p*	(*med_alias*$_r$)

process Mutex ... **end process**
process *Activation* [Start:Block] **is**
  loop
    activate(++ *B*$'_k$)
      $k \in 1..p \wedge$
      $k \notin indices(S, \text{activ})$
  end loop
end process

process S [*visible*(S), Start:Block] **is**
  hide *hidden*(S), Finish:**none in**
    par – main parallel composition
      synch(S)
    in
      par Start, Finish **in**
        par
          b2b (*block_invoc*$_1$)
        || ...
        || *b2b* (*block_invoc*$_p$)
        end par
      ||
        Mutex [Start, Finish]
      end par
    ||
      par
          n2b (*env_invoc*$_1$)
        || ...
        || *n2b* (*env_invoc*$_q$)
        || *m2b* (*med_invoc*$_1$)
        || ...
        || *m2b* (*med_invoc*$_r$)
        || *Activation* [Start]
      end par
    end par
  end hide
end process

Table 5.7: Translation of systems

Example 5.23. Below, the root LNT processes (right-hand side) correspond to GRL systems (left-hand side).

GRL systems

```
1  -- Sen and Rec communicate
2  -- through medium Buf
3  system S5 (Mr, Ms: bool)
4  is
5   block list
6    Sen [?Ms],
7    Rec [Mr]
8   medium list
9    Buf [Ms, ?Mr]
10 end system
```

LNT root processes

```
1  process S5 [Gate_Ms, Gate_Mr: Chan_bool,
2              Start: Block] is
3   hide Finish: None in
4    par
5     par Start, Finish in
6      Mutex [Start, Finish]
7      || par
8           S5_Sen [Gate_Ms, Start, Finish]
9        || S5_Rec [Gate_Mr, Start, Finish]
10       end par
11    end par
12   ||
13    S5_Buf [Gate_Ms, Gate_Mr]
14  end par
15 end hide
16 end process
```

GRL systems

```
1  -- The activation of both
2  -- B1 and B2 is constrained
3  system S6 (O1, O2: bool)
4  is
5   block list
6    B1 (?O1),
7    B2 (?O2)
8   environment list
9    Ctrl (B1, B2)
10 end system
```

LNT root processes

```
1  process S6 [Gate_O1, Gate_O2: Chan_bool,
2              Start: Block] is
3   hide Finish: None in
4    par
5     par Start, Finish in
6      Mutex [Start, Finish]
7      || par
8           S6_B1 [Gate_O1, Start, Finish]
9        || S6_B2 [Gate_O2, Start, Finish]
10    end par
11   end par
12   ||
13    Ctrl [Start]
14   end par
15  end hide
16 end process
```

GRL systems

```
1  -- Only the activation of B1
2  -- is constrained
3  system S7 (O1, O2: bool)
4  is
5    block list
6      B1 (?O1),
7      B2 (?O2)
8    environment list
9      Ctrl (B1)
10 end system
```

LNT root processes

```
1  process S7 [Gate_O1, Gate_O2: Chan_bool,
2              Start: Block] is
3   hide Finish: None in
4    par
5     par Start, Finish in
6      Mutex [Start, Finish]
7      || par
8           S7_B1 [Gate_O1, Start, Finish]
9        || S7_B2 [Gate_O2, Start, Finish]
10    end par
11   end par
12   ||
13      Ctrl [Start]
14   || Activation [Start]
15   end par
16  end hide
17 end process
```

□

Deadlock freedom A deadlock in a GRL program leads to a deadlock in the corresponding LNT program. Indeed, deadlocks in GRL occur if: (i) the activation of all blocks is constrained by environments and (ii) no activation signal is reachable. Because GRL statements (but signals) are the same as in LNT ones, the reachability of a GRL activation signal implies the reachability of the corresponding LNT gate *Start*. If no gate *Start* is reachable, no process can execute, which causes a deadlock in the LNT program.

The translation algorithm does not introduce spurious deadlocks. As regards separate processes, medium and environment (without activation constraints) processes are by construction deadlock-free, and so are processes *Activation* and *Mutex*. In block processes, each synchronisation on gate *Start* is eventually followed by a synchronisation on gate *Finish*; all synchronisations in between are value-exchange communications with medium and environment processes, which always accept to communicate. As regards deadlocks produced by blocking communication, all communications on gates occurring in the synchronisation set of root processes and involving *several* processes are feasible. Still, the only source of deadlocks is unreachable *Start* gates in environment processes. These are equivalent to GRL deadlocks.

5.8 Tool support

Our translation algorithm is implemented in a tool named GRL2LNT, developed mainly by Éric Léo, a software engineer from 2012 to 2016 in the Convecs project, in which this thesis has taken place. GRL2LNT is developed by using the Syntax/Traian Lotos NT technology for compiler construction [GLM02]. It consists of about 30,000 lines of code and translates GRL specifications into LNT. A second tool named GRL.OPEN has been developed. GRL.OPEN encapsulates GRL2LNT and calls LNT.OPEN, thus connecting GRL to the on-the-fly verification tools of CADP.

GRL2LNT and GRL.OPEN are validated on more than 120 GRL specifications, corresponding to about 7,000 lines of GRL, which generates 18,000 lines of LNT. The increase in the number of generated lines is mainly caused by the translation of GRL constructs into several LNT constructs. This shows that GRL is closer to the GALS user's view, compared to LNT.

A part of the benchmark is dedicated to *unit testing* of GRL constructs. At least two examples are written for each GRL syntactic and static semantic rule. The first example violates the rule to check that GRL2LNT captures the error. The second example, which is a corrected version of the first one, checks that no error is raised by GRL2LNT. Another part of the benchmark consists of more elaborated examples covering different aspects of the language. First, the generated LNT programs are

analysed manually to check their conformance with the translation algorithm. Then, the LTSs generated by using CADP, are checked either by visual checking (for small LTSs) or by interactive simulation and model checking (for large LTSs). Our industrial partner generates GRL code automatically from their synchronous programming software, and check the resulting LTSs, which is another validation form for the whole toolchain. Moreover, for each GRL specification, a set of *correct-by-construction* properties can be verified using model checking. Examples are:

- the atomicity of block steps
- the occurrence of inputs before outputs in each step
- the eventual release of the lock after being acquired
- the absence of transitions corresponding to unconnected channels

5.9 LTSs of the translation vs. LTSs of GRL semantics

We address the relationship between the LTSs of GRL semantics (Chapter 4) and the LTSs of root processes generated by the translation (Section 5.7). We note "LTS_{GRL}" and "LTS_{LNT}", respectively, those LTSs.

For a transition $\mu_0 \xrightarrow{B\ (ch_1,...,ch_m)[ch'_1,...,ch'_n]} \mu_1$ in LTS_{GRL}, our translation generates the following LNT transition sequence:

$$\begin{array}{rl} & S_0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad S_{m+n+2} \\ \textit{Start}\ !B \downarrow & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \uparrow \textit{Finish} \\ & S_1 \xrightarrow{G(ch_1)} S_2 \rightarrow ... \rightarrow S_m \xrightarrow{G(ch_m)} S_{m+1} \xrightarrow{G(ch'_1)} S_{m+2} \rightarrow ... \rightarrow S_{m+n} \xrightarrow{G(ch'_n)} S_{m+n+1} \end{array}$$

where G is a label transformation function defined as follows:

$$\begin{array}{lcl} G\ (X_1 = e_1, \ldots, X_p = e_p) & = & \textit{Gate\_}X_1 \ldots \_X_p !\ e_1 !\ ...\ !\ e_p \\ G\ (\_, \ldots, \_) & = & i \end{array}$$

The expansion caused by the translation is linear in the number of transitions, owing to the locking mechanism. In general, consider a GRL system S composed of n blocks B_1, ..., B_n having m_1, ..., m_n channels, respectively. We assume that B_1, ..., B_n have respectively p_1, ..., p_n transitions in LTS_{GRL}. The table below summarises the number of transitions in LTS_{GRL} and LTS_{LNT}. Note that the number of transitions in LTS_{GRL} is independent from the number of channels in blocks, contrarily to LTS_{LNT}.

LTS	LTS_{GRL}	LTS_{LNT}
number of transitions	$\sum_{i=1}^{n} p_i$	$\sum_{i=1}^{n} (2 + m_i \times p_i)$

LTS_{GRL} can be recovered from LTS_{LNT}. Because the transformation from LTS_{GRL} to LTS_{LNT} is *bijective*, there is a one-to-one correspondence between GRL transitions and LNT transition sequences. The labels of LTS_{GRL} transitions could be reconstructed from those of the corresponding LNT transition sequences in LTS_{LNT}, by using function G^{-1}.

Then, in each transition sequence in LTS_{LNT}, one transition can be renamed into the reconstructed label while hiding all other transitions. Finally, by applying a compression (e.g., τ-compression and τ-confluence) to remove hidden transitions, we obtain LTS_{GRL}. This algorithm is implemented in GRL.OPEN, by using an option named *merge*.

5.10 Comparison with related work

We compare our encoding of GALS systems in LNT to the earlier work proposed in [GT09, Thi11], which also use LNT. Similarly to [GT09, Thi11], we encode synchronous components in LNT functions, which we encapsulate inside processes. However, our encoding outperforms [GT09, Thi11] in several aspects:

- Our approach is more general since is it not confined to Mealy machines to describe synchronous components. Not all compilers of synchronous languages are able to synthesise Mealy machines, in which case automatic translators should be developed, as argued by the authors in [GT09, Thi11].
- Our approach is more modular since it allows a wrapper process to define several reception and emission gates, through which the process could interact with several processes. In [GT09, Thi11], a wrapper process defines exactly one reception and exactly one emission gates; this limits the modularity of the approach.
- Our approach leads to smaller LTS since the locking mechanism ensures the atomicity of synchronous components. This impacts the size of the corresponding LTS, as will be explained below.

We consider a GALS system composed of two synchronous components. Each component i ($i \in 1..2$) is represented by an LNT process with one input gate INi, one output gate OUTi, and no internal state. Gates are without value-exchange, for conciseness. In such a system, the only difference between our approach and [GT09, Thi11] is the locking mechanism. Figure 5.1 gives the LTS_{LNT} of the GALS system and the LTS (noted LTS_{GT}) obtained by applying the approach [GT09, Thi11]. It shows that the locking mechanism removes sequences in which the input reading of some component i is followed by the input reading of another component j ($j \neq i$) before the output writing of component j. In our opinion, such situations are irrelevant for GALS systems; even in [GT09, Thi11], removing such sequences seems to have no effect on the truth values of temporal logic formulas.

Furthermore, we increase the number of the concurrent synchonous components in the GALS system. Table 5.8 shows how the number of transitions in LTS_{GT} grows exponentially with the number of components while the number of transitions in LTS_{LNT} grows linearly.

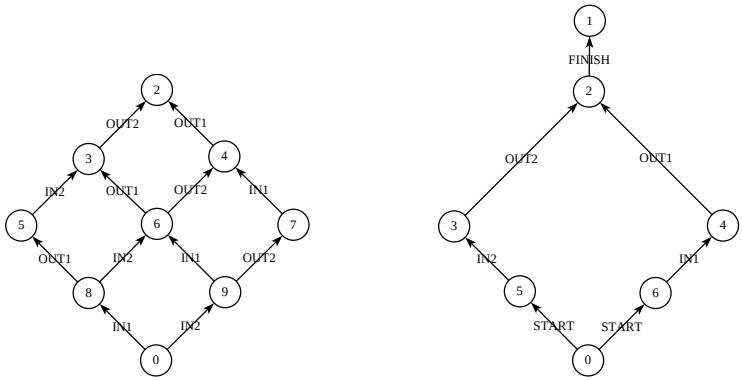


Figure 5.1: LTSs describing LTS_{GT} (left-hand side) and LTS_{LNT} (right-hand side)

Component number	2	3	4	5	6	10
LTS_{LNT}	7	10	13	16	19	31
LTS_{GT}	12	54	217	811	2917	393,661

Table 5.8: The number of transitions in LTS_{LNT} and LTS_{GT} depending on the number of components

5.11 Conclusion

In this chapter, we have proposed a translation from GRL to LNT. However, the principles of our translation would be transferable to other process algebras as well, provided they allow type definitions, function definitions with multiple output parameters, and are equipped with an n-ary parallel composition operator enabling data exchange.

Although both GRL and LNT have formal semantics, we have not proven formally the correctness of the translation. This would be a long task due to the size of GRL, which is far from being a toy language. Nonetheless, we gave hints to the correction of the translation. These hints have been complemented by tests using the GRL and CADP tools.

Chapter 6

The muGRL Language for GALS Property Specification

Temporal logics are tailored to capture properties of concurrent systems, but they require expertise. For example, a property for the car park is: *whenever a car leaves, the green light eventually turns on.* Its formulation in MCL would be:

```
1  [ true* . {Gate_OPEN !true}]
2    mu X.(<true> true and [ not ({Gate_GREEN_YELLOW_RED !true ?any ?any}) ] X)
```

Although the property is simple, its formulation is not easy to read and understand. This chapter proposes muGRL, a property specification language tailored to capture properties of GALS systems. muGRL is based on a system of patterns [DAC98, DAC99]. Patterns are high-level and parameterisable properties, aiming at reducing the complexity of using temporal logics. The semantics of muGRL are defined by a translation into MCL. The chapter is structured as follows. We first provide an overview of muGRL. Then, we present the frequently encountered properties in the scope of concurrent and GALS systems. In particular, we propose a definition of deadlocks, livelocks, and instability for GALS systems, as well as some discrete real-time properties.

6.1 Overview of muGRL

The language syntax is presented in Tables 6.1 (page 120) to 6.4 (page 122). The generic terminal symbols and non-terminal symbols are summarised in the table below. Action formulas enable action-based properties that involve data values to be specified. Regular formulas build upon action formulas and enable complex assertions over action sequences to be specified. Property patterns build upon regular formulas and enable both LTS states to be specified and LTS branching structure to be explored.

	Symbol	Description
Non-terminal symbols	P	*property pattern*
	R	*regular formula*
	A	*action formula*
	O	*offer formula*
	E	*expression*
Terminal symbols	X	*action variable*
	x	*local variable*
	T	*type*

Action formulas, regular formulas, and property patterns are interpreted over LTS_{LNT} (see Section 5.9, page 114), as shown by the flow depicted in Figure 6.1. In particular, the GRL2LNT tool is enhanced with an option "-relabel" which enables to rename actions of the form "Start !B" into "B" and actions of the form "Gate_X_1..._X_p !e_1 ! ... !e_p" into "X_1 = e_1,...,X_p = e_p". Hence, muGRL actions have one of the following forms:

Action	Form	Meaning
Activation action	S	S denotes a block name
Data action	X_1 = e_1, ..., X_n = e_n	X_1, ..., X_n are variables composing a block actual channel and e_1, ..., e_n are their respective values
Invisible action	i	

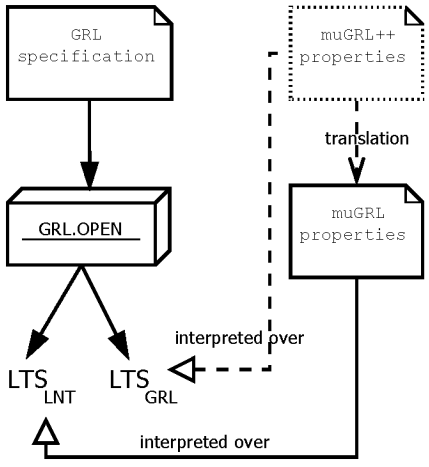


Figure 6.1: Interpretation model of muGRL

Remark 6.1. One would expect the property language to be interpreted over LTS_{GRL}, as depicted by dashed box and arrows in Figure 6.1. We use LTS_{LNT} as interpretation model for the sake of simplicity. It is cost effective, since one can use directly the CADP tools and techniques (hiding, minimisation, etc.) without needing to introduce

additional tools. Of course it is possible to write another property language (muGRL++, Figure 6.1) with LTS_{GRL} as interpretation model, and define a translation from that language to muGRL. This is left for future work. ■

Example 6.1. Throughout the current chapter, properties will be illustrated mainly on the car park application (see Section 3.1, page 29). We will consider the following subset of LTS_{LNT} actions (after renaming), where e_0, e_1, ... are Boolean values:

- activation actions "Entrance", "Exit", "Storey1", and "Storey2" denote PLC execution.
- data action "Cmd_P1 = e_1 , Cmd_P2 = e_2" denotes a request of a car, parking in the first or the second storey, to leave the car park.
- data action "Open_Park = e_0" denotes the opening of the entrance gate on a car request.
- data action "Open = e_0" denotes the opening of the exit gate on a car request.
- data actions "Out_P1 = e_1" and "Out_P2 = e_2" denote the leaving of a car, parking in the first or the second storey, respectively.
- data action "Green = e_1 , Yellow = e_2 , Red = e_3" denotes the exterior lights mounted at the car park entrance.
- data action "S_Out1 = e_0" denotes a message sent by the exit PLC to the first storey PLC indicating whether a car has left.
- data action "R_Out1 = e_0" denotes a message sent by the first storey PLC to the entrance PLC indicating whether a car has left.

□

6.2 Offer formulas

Offer formulas, whose syntax is given in Table 6.1, characterise a couple "X = e" present on a data action. More precisely, an offer formula allows the value carried by an action variable to be matched and stored in a variable local to the formula:

- Offer formula "X_0 = e_0" matches an action variable carrying a value identical to e_0.
- Offer formula "X_0 =? $x_0 : T_0$" matches an action variable carrying any value of type T_0 and stores it in variable x_0. We call *local* the variables used to store the values of action variables in formulas.
- Offer formula "X_0 =? **any**" matches any value of any type.

Example 6.2. Offer formula "*Open* = **false**" matches "Open = false" but does not match "Open = true". Offer formula "*Open* =? **any**" matches both "Open = false" and "Open = true". □

O	::=	X_0 = e_0	value matching
	|	X_0 =? x_0 : T_0	value extraction
	|	X_0 =? **any**	wildcard

Table 6.1: Syntax of offer formulas

6.3 Action formulas

Action formulas, whose syntax is given in Table 6.2, consist of *action predicates* combined using standard Boolean operators. Action predicate {S}, where S is a string denoting a component name[1], matches the activation action S. Action predicate {O_1 , ... , O_n [**where** E]} matches the data action "X_1 = e_1 , ..., X_n = e_n" if: (i) each offer O_i matches X_i = e_i, and (ii) the Boolean expression E, possibly using local variables occurring in O_1 , ... , O_n, evaluates to true.

For action predicates with value extraction, local variables can be exported outside the action predicates. Hence, they can be used in the surrounding formula. This is a generality of the action formulas first proposed in ACTL [DFGR93] in a dataless context.

A	::=	{S}	activation action
	|	{O_1 , ... , O_n [**where** E]}	data action
	|	**i**	invisible action
	|	**false**	unexisting action
	|	**true**	every action
	|	(A)	parenthesised action
	|	**not** A	negation
	|	A_1 **or** A_2	disjunction
	|	A_1 **and** A_2	conjunction

Table 6.2: Syntax of action formulas

Example 6.3. In the car park application, the action formula {$Exit$} matches the activation action Exit. The action formula {Cmd_P1 = **true**, Cmd_P2 =? **any**} matches the data actions "Cmd_P1 = true, Cmd_P2 = true" and "Cmd_P1 = true, Cmd_P2 = false" but does not match the data action "Cmd_P1 = false, Cmd_P2 = true". □

6.4 Regular formulas

Regular formulas, whose syntax is given in Table 6.3, characterise regular execution sequences. A regular formula is built from action formulas using standard regular expression operators. In particular, action predicates with value extraction and matching

[1] In muGRL, the term component is used to denote a synchronous component.

enable to specify the propagation of values on execution sequences in the LTS.

R	::=	A	one-step sequence
	|	NIL	empty sequence
	|	(R)	parenthesised formula
	|	R_1 . R_2	concatenation
	|	$R_1 \mid R_2$	choice
	|	R^*	iteration 0 or more times
	|	R^+	iteration 1 or more times
	|	R {m}	iteration m times
	|	R {m...n}	iteration m to n times

Table 6.3: Syntax of regular formulas

Example 6.4. In the car park application, the following regular formula specifies a step of block *Exit* in which a request for leaving is detected.

{*Exit*} . ({*Cmd_P1* = **true**, *Cmd_P2* = **false**} **or** {*Cmd_P1* = **false**, *Cmd_P2* = **true**}) .
true . {*Out_P1* =? *out1*:**bool**} . {*Out_P2* =? *out2*:**bool where** *out1*<>*out2*}

The regular formula matches the following action sequence:

$$s_0 \xrightarrow{Exit} s_1 \xrightarrow{Cmd\_P1=true,\ Cmd\_P2=false} s_2 \xrightarrow{Open=true} s_3 \xrightarrow{Out\_P1=true} s_4 \xrightarrow{Out\_P2=false} s_5$$

□

6.5 General property patterns

We consider the classification, first suggested by Lamport [Lam77], partitioning properties into *safety* and *liveness* ones. A safety property asserts that something (bad) *will not* happen. A liveness property asserts that something (good) *must* happen. Additionally, we consider the class of *fairness* properties. Fairness is concerned with resolving nondeterminism and is often required to ensure liveness. For each of those classes, we present the frequently encountered patterns in the literature. A final section presents a translation of the presented patterns into MCL.

6.5.1 Patterns for safety properties

To express a safety property in the action-based setting, we proceed as follows. First, the undesirable execution sequences are characterised in terms of regular formulas R. Then, the occurrences of R in the LTS are forbidden by using the following pattern:

Never (R)

P	::=	**true**
	|	**false**
	|	**not** P
	|	P_1 **or** P_2
	|	P_1 **and** P_2
	|	P_1 **implies** P_2
	|	Never (R)
	|	Not_To_Unless (A_1, A_2, A_3)
	|	Some (R)
	|	Always_Some (R)
	|	After_Some (R_1, R_2)
	|	Some_Never (R_1, R_2)
	|	After_Inev (R_1, R_2)
	|	Looping (R)
	|	Saturation (R)
	|	After_Looping (R_1, R_2)
	|	Some_Looping (R_1, R_2)
	|	After_Saturation (R_1, R_2)
	|	Some_Saturation (R_1, R_2)
	|	Deadlock (S)
	|	Alive (S)
	|	Some_Alive ($\overline{S}$)
	|	All_Alive ($\overline{S}$)
	|	Deadlock ($\overline{S}$)
	|	Progress ($\overline{S}$)
	|	Idle ($\overline{X}$)
	|	Some_Idle ($\overline{\overline{X}}$)
	|	All_Idle ($\overline{\overline{X}}$)
	|	Idle (S)
	|	Idle ($\overline{S}$)
	|	Progress ($\overline{X}$)
	|	Some_Progress ($\overline{\overline{X}}$)
	|	All_Progress ($\overline{\overline{X}}$)
	|	Progress (S)
	|	Progress ($\overline{S}$)
	|	Starvation_Freedom (S)
	|	Starvation_Freedom (S, S')
	|	Starvation_Freedom ($\overline{S}$, $\overline{S'}$)
	|	Out_Consistent ($\overline{X}$, $\overline{Y}$)
	|	Some_Consistent ($\overline{\overline{X}}$, $\overline{Y}$)
	|	All_Consistent ($\overline{\overline{X}}$, $\overline{Y}$)
	|	All_Consistent ($\overline{\overline{X}}$, $\overline{\overline{Y}}$)
	|	Stability ($\overline{\overline{X}}$, $\overline{Y}$)
	|	Stability ($\overline{\overline{X}}$, $\overline{\overline{Y}}$)
	|	Deadline (R, A_1, A_2, n)
	|	Sustain (R, A_1, A_2, n)
	|	From_To_Least (A_1, A_2, A_3, m)
	|	From_To_Most (A_1, A_2, A_3, m)

Table 6.4: Syntax of property patterns

Example 6.5. The informal specification of the car park application (Section 3.1, page 29) states that: *once a car asks for leaving the car park, an exit request is detected by the exit PLC, which opens the gate immediately.* An undesirable execution sequence is then an exit request that is not followed by the gate opening. Such a situation is captured and forbidden by the safety property:

$$\mathsf{Never}\left(\begin{array}{l}\mathbf{true}^*\ .\\ \{Cmd\_P1\ \texttt{=?}\ cmd1\texttt{:}\mathbf{bool},\ Cmd\_P2\ \texttt{=?}\ cmd2\texttt{:}\mathbf{bool}\ \mathbf{where}\ cmd1\texttt{<>}cmd2\}\ .\\ \{Open\ \texttt{=}\ \mathbf{false}\}\end{array}\right)$$

□

Further safety patterns can be obtained by specialising the regular formula R in pattern Never. In practice, a frequently encountered pattern forbids the execution of an action A_2 after an action A_1 without the occurrence of an action A_3 in between:

$$\mathsf{Not\_To\_Unless}\ (A_1, A_2, A_3) = \mathsf{Never}\ \ (\mathbf{true}^*\ .\ A_1\ .\ (\mathbf{not}\ A_3\ )^*\ .\ A_2)$$

Example 6.6. In the car park application, a valid safety property is: *the exit gate cannot open unless an exit request is detected.* Such a situation is captured and forbidden by the safety property:

$$\mathsf{Not\_To\_Unless}\left(\begin{array}{l}\{Open\ \texttt{=}\ \mathbf{false}\},\\ \{Open\ \texttt{=}\ \mathbf{true}\},\\ \{Cmd\_P1\ \texttt{=?}\ cmd1\texttt{:}\mathbf{bool},\ Cmd\_P2\ \texttt{=?}\ cmd2\texttt{:}\mathbf{bool}\ \mathbf{where}\ cmd1\texttt{<>}cmd2\}\end{array}\right)$$

□

To conclude that a safety property is violated on an LTS, it suffices to have one (finite) execution sequence forbidden by the property but occurring in the LTS. For instance, the following execution sequence violates the property specified in Example 6.6:

$$s_0 \xrightarrow{\cdots} s_1 \xrightarrow{Open=false} s_2 \xrightarrow{\cdots} s_3 \xrightarrow{\cdots} s_4 \xrightarrow{Cmd\_P1=false,\ Cmd\_P2=false} s_1 \xrightarrow{Open=true}$$

Note however that even though a safety property holds on an LTS, there is no proof about the existence of the actions referenced in the property. Property Never ({*Open* = **false**}) holds on an LTS containing no action Open. This is called the *vacuity* problem. In addition to a safety property holding on an LTS, one must check the existence of all actions referenced in the formula. This is possible using pattern Some (see Section 6.5.2).

6.5.2 Patterns for liveness properties

A system can fulfill all safety properties by forever doing nothing as this will never entail undesirable situations. In the car park application, it suffices that no car enters the car park to make the properties specified in Examples 6.5 and 6.6 hold on the corresponding LTS. Such behaviour is usually useless for a system. For this reason, safety properties need to be complemented by liveness properties, which express progress.

Reachability properties

A typical instantiation of progress properties is the *reachability* of an execution sequence. In terms of LTSs, such a property asserts that from the initial state, there is some outgoing execution sequence that satisfies a regular formula R. This is denoted by the following pattern:

Some (R)

A stronger progress pattern is the *universal reachability*, i.e., reachability on all execution sequences. It asserts that it is always possible for an action to be eventually reached. In terms of LTSs, such a property asserts that from each state, there is some outgoing execution sequence that satisfies a regular formula R. This is denoted by the following pattern:

Always_Some (R)

Example 6.7. The informal specification of the car park application (Section 3.1, page 29) states that: *if there still are unoccupied parking spots, ... a green light is maintained on; otherwise, a red light is turned on.* Some reachability properties are the following:

Property	Formalisation
The red light may be on	Some (**true*** . {Green =? **any**, Yellow =? **any**, Red = **true**})
The green light may always turn on	Always_Some ({Green = **true**, Yellow =? **any**, Red =? **any**})

□

Response properties

Other frequent instantiations of liveness properties are the so-called *response* properties. They assert that whenever certain actions occur (request), they must be followed by other actions in the future (response). Depending on the way the response is requested to occur in the LTS, two typical response patterns can be distinguished: *potentiality* and *inevitability*.

Potentiality The potentiality response pattern expresses the occurrence of the response on some execution sequence. It specifies that every execution sequence satisfying a regular formula R_1 is *potentially* followed by another execution sequence satisfying a regular formula R_2. This is denoted by the following pattern:

After_Some (R_1, R_2)

Such a property is satisfied by a state of the LTS if: each of its outgoing execution sequences satisfying R_1 leads to another state, from which there is some outgoing execution sequence satisfying R_2. A typical particular case of this pattern specifies that every action A is potentially followed by an action B. This is denoted by the following

pattern:

$$\mathsf{After\_Some}\ (\mathbf{true}^*\ .\ A,\ \mathbf{true}^*\ .\ B)$$

If this property is valid on an LTS, it means that from every state following an action A, there is at least one outgoing execution sequence leading to an action B. The pattern $\mathsf{After\_Some}\ (A,\ B)$ is the action-based counterpart of the fair inevitability operator proposed in [QS83] in the state-based setting.

The dual pattern of After_Some specifies that there is an execution sequence satisfying R_1, that leads to a state, from which the outgoing execution sequences satisfying R_2 are forbidden. This is denoted by the following pattern:

$$\mathsf{Some\_Never}\ (R_1,\ R_2) \quad = \quad \mathbf{not}\ \ \mathsf{After\_Some}\ (R_1,\ R_2)$$

Inevitability The inevitability response pattern expresses the occurrence of the response on all execution sequences. It specifies that every execution sequence satisfying the regular formula R_1 is *eventually* followed by another execution sequence satisfying the regular formula R_2. This is denoted by the following pattern:

$$\mathsf{After\_Inev}\ (R_1,\ R_2)$$

Similarly, a useful specialisation of this pattern specifies that every action A is eventually followed by an action B:

$$\mathsf{After\_Inev}\ (\mathbf{true}^*\ .\ A,\ \ \mathbf{true}^*\ .\ B)$$

If this property is valid on the LTS, it means that from every occurrence of action A, all execution sequences contains an action B.

Example 6.8. For the car park application, we specify the properties summarised in the table below. The last one is extracted from the informal specification (Section 3.1, page 29) stating that: *once the car leaves, the exit PLC informs the storey PLC referenced in the car ticket, which in turn informs the entrance PLC.*

Property	Formalisation
If the car park is full, a parking car eventually leaves	$\mathsf{After\_Some}\left(\begin{array}{l}\mathbf{true}^*\ .\ \{Green\ \texttt{=?}\ \mathbf{any}, Yellow\ \texttt{=?}\ \mathbf{any}, Red\ \texttt{=}\ \mathbf{true}\},\\ \{Open\ \texttt{=}\ \mathbf{true}\}\end{array}\right)$
The car park may never be full	$\mathsf{Some\_Never}\left(\begin{array}{l}\mathbf{true}^*,\\ \mathbf{true}^*\ .\ \{Green\ \texttt{=?}\ \mathbf{any}, Yellow\ \texttt{=?}\ \mathbf{any}, Red\ \texttt{=}\ \mathbf{true}\}\end{array}\right)$
A message sent by the exit PLC on a car leaving must be transmitted to the entrance PLC	$\mathsf{After\_Inev}\left(\begin{array}{l}\mathbf{true}^*\ .\ \{S\_Out1\ \texttt{=}\ \mathbf{true}\},\\ \mathbf{true}^*\ .\ \{R\_Out1\ \texttt{=}\ \mathbf{true}\}\end{array}\right)$

□

To conclude that a liveness property is violated on an LTS, all (infinite) execution sequences should be visited to check that the execution sequence required by the property is absent in the LTS. Nonetheless, response properties exhibit no guarantee about the

existence of requests. Accordingly, they are not violated by infinite execution sequences in which only responses occur but never requests (or finitely many requests).

Remark 6.2. The stronger response pattern After_Inev (**true*** . A, **true*** . B) may not hold on an LTS, unexpectedly, if there are cycles after an occurrence of A and before the subsequent occurrence of B. If these cycles correspond to unrealistic executions of the system, one may check the first version After_Some (**true*** . A, **true*** . B) of the response pattern. If the property holds, a scheduler (i.e., activation strategy) can be implemented, to avoid the unrealistic execution cycles. ■

6.5.3 Patterns for fairness properties

Fairness assumptions capture infinite behaviors that are considered unrealistic. In the car park application, a possible unfair scenario is the following: once a car leaves the car park, the entrance PLC waits infinitely long without receiving the information from the exit PLC (via a storey PLC). The availability of the car park is then never updated.

In the action-based setting, fairness can be specified by characterising the LTS cycles denoting infinite unfair execution sequences. The following property pattern specifies the existence of an infinite execution sequence satisfying the regular formula R:

Looping (R)

This is the action-based counterpart of the LTL property GFp expressing that a state property p occurs infinitely often [CGP00]. A useful particular case of the Looping pattern specifies the existence of an execution sequence on which an action A occurs infinitely often:

Looping ((**not** A)* . A)

Example 6.9. For the car park application, we can specify that each PLC is executed infinitely often.

Looping ({*Entrance*}) Looping ({*Storey1*})
Looping ({*Exit*}) Looping ({*Storey2*})

□

The dual pattern of Looping specifies that an execution sequence R can be repeated only a finite number of times:

Saturation (R) = **not** Looping (R)

This is the action-based counterpart of the LTL property FGp expressing the invariance of a state property p [CGP00]. As above, a useful particular case of the Saturation pattern specifies that all execution sequences may contain only a finite number of occurrences of actions different from A.

Saturation (**true*** . **not** A)

Looping and saturation patterns can be combined with the response patterns to accomodate the presence or absence of certain infinite sequences after other execution sequences

have occurred:

After_Looping (R_1, R_2) After_Saturation (R_1, R_2)

The dual patterns are respectively:

Some_Saturation (R_1, R_2) Some_Looping (R_1, R_2)

Fairness verification is of particular interest for nondeterministic systems by detecting whether a possible choice is consistently ignored. Asynchronous concurrent systems, among which GALS systems, are particularly concerned as concurrency is often modelled by interleaving behaviours. Further explanation on this concern will follow in the specific case of GALS systems.

6.5.4 Translation into MCL

The semantics of muGRL are defined by translation into MCL. Each muGRL expression has a one-to-one straightforward correspondence with its counterpart in MCL. We call *e2mcl* the translation function of muGRL expressions. Regular formula has a one-to-one straightforward correspondence with its counterpart in MCL. We call *r2mcl* the translation function from muGRL regular expressions to their MCL counterparts.

The translation of offer formulas is given when translating action formulas. We need the following intermediate functions:

var (X_0 = e_0)	=	X_0	*val* (X_0 = e_0)	=	!e_0
var (X_0 =? x_0:T_0)	=	X_0	*val* (X_0 =? x_0:T_0)	=	?x_0:T_0
var (X_0 =? **any**)	=	X_0	*val* (X_0 =? **any**)	=	?**any**

The translation of action formulas is summarised in Table 6.5.

Action formula A	Translation into MCL *a2mcl (A)*
{S}	{Start! S}
{$O_1, \ldots, O_n$ **where** E}	{Gate_$var(O_1)$_..._$var(O_n)$!$val(O_1)$ ... !$val(O_n)$ **where** $e2mcl(E)$}
i	**i**
false	**false**
true	**true**
(A)	(*a2mcl* (A))
not A	**not** *a2mcl* (A)
A_1 **or** A_2	*a2mcl* (A_1) **or** *a2mcl* (A_2)
A_1 **and** A_2	*a2mcl* (A_1) **and** *a2mcl* (A_2)

Table 6.5: Translation of action formulas

Safety property patterns. The translation of safety patterns, using function *p2mcl*, is summarised in Table 6.6. Safety patterns can be naturally expressed using necessity modalities containing regular formulas.

Pattern P	Translation into MCL *p2mcl (P)*
Never (*R*)	[*r2mcl* (*R*)] **false**
Not_To_Unless (*A*$_1$, *A*$_2$, *A*$_3$)	[*r2mcl* (**true*** . *A*$_1$. (**not** *A*$_3$)* . *A*$_2$)] **false**

Table 6.6: Translation of safety patterns

Liveness property patterns. The translation of liveness properties, using function *p2mcl*, is summarised in Table 6.7. Potentiality pattern Some can be directly expressed using possibility modalities containing regular formulas. Potentiality response pattern After_Some can be expressed by combining necessity and possibility modalities operators. The Some_Never pattern is the dual of the After_Some one. The Always_Some pattern can be expressed by specialising the After_Some pattern with a first argument matching all sequences. Finally, the inevitability response pattern can be expressed by combining necessity and possibility modalities together with a fixed point operator. More precisely, the MCL formula states that all execution sequences containing subsequences satisfying *R* must lead to states from which action *A* is reachable. The inevitable reachability of action *A* is ensured by the minimal fixed point operator binding variable *X*.

Pattern P	Translation into MCL *p2mcl (P)*
Some (*R*)	< *r2mcl* (*R*) > **true**
After_Some (*R*1, *R*2)	[*r2mcl* (*R*$_1$)]< *r2mcl* (*R*$_2$) > **true**
Some_Never (*R*1, *R*2)	< *r2mcl* (*R*$_1$) >[*r2mcl* (*R*$_2$)] **false**
Always_Some (*R*)	[**true***]< *r2mcl* (*R*) > **true**
After_Inev (*R*, *A*)	[*r2mcl* (*R*)] **mu** *X* . (< **true** > **true and** [**not** *a2mcl*(*A*)] *X*)

Table 6.7: Translation of liveness patterns

Remark 6.3. Concerning inevitability patterns, we restrict the translation to the particular case After_Inev (*R*, *A*). The translation of the general case After_Inev (*R*$_1$, *R*$_2$) into MCL is quite tedious, for the time being. ■

Fairness property patterns. The translation of fairness properties, using function *p2mcl*, is summarised in Table 6.8. Fairness patterns are translated by using the infinite looping and finite saturation operators.

6.6 Deadlock, livelock, and instability

This section proposes a set of GALS-specific property patterns. The proposed patterns exploit the behaviour of GALS systems along two axis: activation strategies (*activation patterns*) and data handling (*data patterns*).

Pattern P	Translation into MCL *p2mcl (P)*
Looping (*R*)	< *r2mcl* (*R*) > @
Saturation (*R*)	[*r2mcl* (*R*)] -|
After_Looping (R_1, R_2)	[*r2mcl* (R_1)] < *r2mcl* (*R2*) > @
After_Saturation (R_1, R_2)	[*r2mcl* (R_1)] [*r2mcl* (*R2*)] -|
Some_Looping (R_1, R_2)	< *r2mcl* (R_1) > < *r2mcl* (*R2*) > @
Some_Saturation (R_1, R_2)	< *r2mcl* (R_1) > [*r2mcl* (*R2*)] -|

Table 6.8: Translation of fairness patterns

Activation patterns characterise activation actions corresponding to one (S) or several (S_1, ..., S_n) components. We write $\overline{S}$ as shorthand for the list $\langle S_1, ..., S_n \rangle$.

Data patterns characterise one or several data actions corresponding to one or several components. For a data action "X_1 = e_1, ..., X_n = e_n", we write $\overline{X}$ as shorthand for the list of its action variables $\langle X_1, ..., X_n \rangle$. For several data actions, we write $\overline{\overline{X}}$ as shorthand for the list $\langle \overline{X}_1, ..., \overline{X}_n \rangle$.

Sometimes, we will need to specify to which component a data action corresponds. To this aim, we write $\overline{X} \in S$, where S is a component name[2]. To denote all data actions of component S, we write $\overline{\overline{X}} \sim S$[3].

We extend value matching and value extraction for a set of action variables as follows:

Symbol	Meaning
$\overline{X}$=$\overline{e}$	X_1 =e_1,...,X_n =e_n
$\overline{X}$=?$\overline{x}$:$\overline{T}$	X_1 =?x_1:T_1,...,X_n =?x_n:T_n
$\overline{X} \neq \overline{e}$	X_1 =?x_1:T_1,...,X_n =?x_n:T_n **where** x_1<>e_1 **or** ... **or** x_n<>e_n
$\overline{X}$=? **any**	X_1 =?**any**,...,X_n =?**any**

The interpretation of patterns will be mostly given in muGRL, i.e., by specialising the previously presented patterns. Sometimes, it will be given directly in MCL, when the expression in muGRL is impossible (due to expressiveness limitation) or for more efficient interpretation.

6.6.1 Deadlock

Since GALS components execute continuously, termination, called *deadlock*, is highly undesirable and mostly symptomatic of design errors. An exception is the case in which termination is deliberately modelled, e.g., to denote the end of finite scenarios or component failure. We distinguish two cases of deadlock. *Activation deadlock* is concerned with system halt. *Data deadlock* is concerned with computation termination.

[2]In terms of LTS_{LNT}, this means that action S is the last activation action preceding the data action "X_1 = e_1, ..., X_n = e_n", where $\overline{X} = \langle X_1, ..., X_n \rangle$.

[3]In terms of LTS_{LNT}, this means that $\overline{\overline{X}}$ corresponds to all data actions following the activation action S before the next occurrence of another activation action.

Activation deadlock (system halt)

Activation patterns may involve either one or several components or the whole system, i.e., all system components. Their interpretation is summarised in Table 6.9.

Pattern		Interpretation
Deadlock	(S)	Some_Never (**true***, **true*** . {S})
Alive	(S)	Always_Some (**true*** . {S})
Some_Alive	$(\overline{S})$	[**true***] $\bigvee_{S \in \overline{S}}$ (<**true*** . {S}> **true**)
All_Alive	$(\overline{S})$	[**true***] $\bigwedge_{S \in \overline{S}}$ (<**true*** . {S}> **true**)
Deadlock	$(\overline{S})$	<**true***> $\bigvee_{S \in \overline{S}}$ ([**true*** . {S}] **false**)
Progress	$(\overline{S})$	[**true***] $\bigvee_{S \in \overline{S}}$ (<**true*** . {S}> **true**)

Table 6.9: Interpretation of activation deadlock patterns

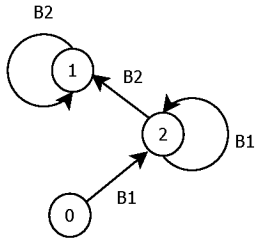
Component halt and aliveness It is desirable to ensure that GALS components do not halt. In terms of LTSs, a component S comes to a halt if there is a state from which no action S can be reachable. This is detected by the following property pattern:

Deadlock (S)

The dual pattern of Deadlock expresses the continuity in component execution. A component S is said *alive* if it executes continuously. In terms of LTSs, from each state, there should be some execution sequence that leads to some action S. This is detected by the following property pattern:

Alive (S)

Example 6.10. Consider a GRL system composed of two blocks *B1* and *B2*, whose activation strategy is described by the LTS below. The evaluation results of patterns Deadlock and Alive on both blocks are summarised in the table below.



	Deadlock	Alive
B1	true	false
B2	false	true

□

Component aliveness can be generalised to encompass a set of components $\overline{S}$ as follows:

Some_Alive ($\overline{S}$) All_Alive ($\overline{S}$)

Pattern Some_Alive asserts that some components in $\overline{S}$ are alive. In terms of LTSs, it evaluates to false if there is a state from which, no component in $\overline{S}$ is alive. Pattern All_Alive asserts that all components in $\overline{S}$ are alive. In terms of LTSs, it evaluates to false if there is a state from which, at least one component in $\overline{S}$ comes to a halt.

Remark 6.4. Patterns Some_Alive and All_Alive could have been interpreted in terms of pattern Alive as follows:

$$\text{Some\_Alive } (\overline{S}) = \bigvee_{S \in \overline{S}} \text{Alive } (S) \qquad \text{All\_Alive } (\overline{S}) = \bigwedge_{S \in \overline{S}} \text{Alive } (S)$$

This interpretation is less efficient than, while being semantically equivalent to, the one we chose, since it leads to MCL formulas with more operators[4]. Each kleene-star operator denotes an implicit fix point operator as it requires to visit all the LTS states in the worst case. ■

A particularly interesting application of patterns Some_Alive and All_Alive concerns redundant systems, in which the same program is executed by several components. This way, the reliability of the application is increased by providing it with fault tolerance. The system is considered operational whenever at least one among redundant components is working properly. If all redundant components fail, the entire system fails as well.

Example 6.11. We enhance the car park application by adding redundancy. The entrance gate is henceforth managed by two redundant PLCs, named *Entrance1* and *Entrance2*. The exit gate is also managed by two redundant PLCs, named *Exit1* and *Exit2*. Basically, the system function is ensured by *primary* PLCs *Entrance1* and *Exit1*, *secondary* PLCs *Entrance2* and *Exit2* being not working. Such a redundancy is called *cold*. Once one of the primary PLCs fails, primary PLCs are both stopped and secondary ones started, instead. The following properties can be specified:

Property	Formalisation
There is always some entrance and some exit PLC working properly	Some_Alive (Entrance1, Entrance2) **and** Some_Alive (Exit1, Exit2)
The car park is always operational	All_Alive (Entrance1, Exit1) **or** All_Alive (Entrance2, Exit2)

□

[4]The complexity of model checking increases with the size of the formula, i.e., the number of logic operators. Thus, it is desirable to verify formulas as small as possible to enhance the efficiency of model checking.

System deadlock and progress A GALS system comes to a halt when all its components do so. However, in a non-terminating GALS system, no conclusion can be derived about whether (or not) it contains halted components. Since (synchronous) components do not interlock the execution of each other, the halting of a component does not entail other components to halt.

Assume the GALS system under study consists of a set of components, denoted $\overline{S}$. Termination of the entire system is expressed by the following pattern:

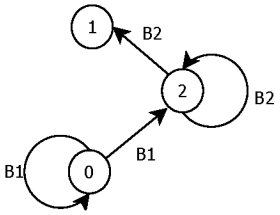
Deadlock ($\overline{S}$)

In terms of LTSs, the pattern evaluates to true if there is a state from which all actions in $\overline{S}$ are unreachable. Hence, the LTS contains necessarily *sink* states, i.e., states without outgoing transitions. This coincides with the definition of deadlock in asynchronous concurrent systems, in which global deadlocks are terminal states from which no more action is possible.

The dual pattern of Deadlock expresses that always some component is executing. In terms of LTSs, it suffices to ensure that from each state, some action in $\overline{S}$ is reachable. This is done by following pattern:

Progress ($\overline{S}$)

Example 6.12. Consider a GRL system composed of two blocks *B1* and *B2*, whose activation strategy is described by the LTS below. Property Deadlock (*B1*, *B2*) evaluates to true while property Progress (*B1*, *B2*) does not. The same verification results are transferable to the LTS corresponding to the whole GRL system (i.e., including data actions).



□

Data deadlock (idleness)

Data patterns may involve one or several data actions of one or several components. Their interpretation is summarised in Table 6.10.

A component step can be idle, i.e., inputs and outputs[5] carry the same values as in the

[5]The term input (resp. output) denotes a data action of a synchronous component corresponding to inputs received (resp. outputs sent) from (resp. to) either the environment or other synchronous components through the network.

Pattern		Interpretation
Idle	$(\overline{X})$	Never $(\mathbf{true}^* \,.\, \{\overline{X}{=}?\overline{x}\} \,.\, \mathbf{true}^* \,.\, \{\overline{X} \neq \overline{x}\})$
Some_Idle	$(\overline{\overline{X}})$	$\bigvee_{\overline{X} \in \overline{\overline{X}}}$ Idle $(\overline{X})$
All_Idle	$(\overline{\overline{X}})$	$\bigwedge_{\overline{X} \in \overline{\overline{X}}}$ Idle $(\overline{X})$
Idle	(S)	All_Idle $(\overline{\overline{X}})$ where $\overline{\overline{X}} \sim S$
Idle	$(\overline{S})$	$\bigwedge_{S \in \overline{S}}$ Idle (S)
Progress	$(\overline{X})$	Always_Some $(\mathbf{true}^* \,.\, \{\overline{X}{=}?\overline{x}\} \,.\, \mathbf{true}^* \,.\, \{\overline{X} \neq \overline{x}\})$
Some_Progress	$(\overline{\overline{X}})$	$\bigvee_{\overline{X} \in \overline{\overline{X}}}$ Progress $(\overline{X})$
All_Progress	$(\overline{\overline{X}})$	$\bigwedge_{\overline{X} \in \overline{\overline{X}}}$ Progress $(\overline{X})$
Progress	(S)	Some_Progress $(\overline{\overline{X}})$ where $\overline{\overline{X}} \sim S$
Progress	$(\overline{S})$	$\bigvee_{S \in \overline{S}}$ Progress (S)

Table 6.10: Interpretation of idleness and progress patterns

previous step. Performing indefinitely idle steps is useless for the system progress. In the car park application, the idleness of the entrance PLC makes the car park inaccessible.

Action and component idleness A data action is said *idle* if there is a state from which its variables, denoted $\overline{X}$, carry indefinitely the same values. The following pattern forbids the presence of execution sequences in an LTS, in which action variables $\overline{X}$ carry different sets of data values:

Idle $(\overline{X})$

Property Idle can be extended to encompass a set of data actions, whose variables are denoted $\overline{\overline{X}}$. Property Some_Idle (resp. All_Idle) holds on an LTS if one or several (resp. all) actions are active.

Some_Idle $(\overline{\overline{X}})$ All_Idle $(\overline{\overline{X}})$

A component S is said idle if it reaches a state from which all its inputs and outputs are idle. Component idleness is a specialisation of property All_Idle applied to all input and output actions of the component. This is detected by the following property:

Idle (S)

Example 6.13. In the car park example, the idleness of the exit PLC, written Idle (Exit), is equivalent to the following property:
Idle (Cmd_P1, Cmd_P2) **and** Idle (Open) **and** Idle (Out_P1) **and** Idle (Out_P2)

□

Remark 6.5. There may be situations in which property Idle (S) does not hold for a component S whereas the component is functionally idle. For illustration, consider the GRL block B below, where u represents the edges of a clock signal. The table in the right-hand side shows the values of parameters u, x, and y during the first six steps. Beyond the sixth step, x and y carry indefinitely the same values.

```
1  block B (in  u: bool,
2           in  x: nat,
3           out y: nat)
4  is
5     if u then y := x
6     else y := 0
7     end if
8  end block
```

u	false	true	false	true	false	true
x	1	1	0	0	0	0
y	0	1	0	0	0	0

The behaviour of the block is idle after its second step, even though u continues to carry different values. The evaluation of idleness properties on the actions corresponding to block B are given in the table below.

Property	Evaluation
Idle (u)	false
Idle (x)	true
Idle (y)	true

Property	Evaluation
Some_Idle (u, x, y)	true
All_Idle (x, y)	true
Idle (B)	false

Idleness is captured by property All_Idle (x, y) but not by Idle (B), which is equal to All_Idle (u, x, y). In general, to capture the idleness of synchronous components, it is more adequate to apply the pattern All_Idle to a well-chosen subset of inputs and outputs. In particular, only functional inputs, whose values are used to compute outputs, should be considered. ■

System idleness A GALS system is said idle if there is a state from which all its synchronous components are idle. In such case, no progress is made anymore by none of the components. For a GALS system composed of a set of components, denoted $\overline{S}$, the idleness of the system is detected by the following pattern:

Idle ($\overline{S}$)

Action, component, and system progress Instead of checking idleness, one might want to check that an action, a component, or a system makes progress. A data action is said to progress if its variables continue forever to carry different values. The following pattern indicates that it is always possible for an action to progress.

Progress ($\overline{X}$)

We extend pattern Progress to encompass a set of actions, a component, and a set of

components, similarly to idleness properties. For a set of actions (resp. components), the patterns Some_Progress and All_Progress assert the progress of some or all actions independently from each other. A GALS system is said to progress if some of its components progress as well.

Some_Progress ($\overline{\overline{X}}$)	All_Progress ($\overline{\overline{X}}$)
Progress (S)	Progress ($\overline{S}$)

6.6.2 Livelock

Informally speaking, a system is said to livelock (or diverge) if it continues to execute without doing the tasks for which it was designed. The presence of livelock may invalidate some verification results and is often due to a bug in the modelling. The absence of livelock ensures not only that the system execution is continuous but also that the execution is meaningful.

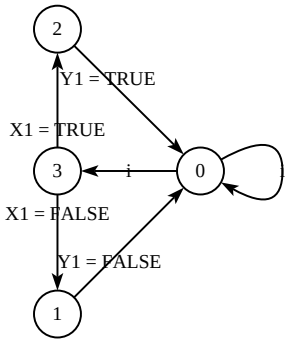
Several yet nonequivalent notions of livelock properties have been defined in the literature of concurrent systems. In process algebra, livelocks arise mainly from the use of the hiding operator. A livelock occurs if a process reaches a state from which it may execute indefinitely an infinite sequence of consecutive hidden actions, which cannot be observed from the process outside. Following this definition, livelock can be expressed by the following pattern:

$$\mathsf{LIVELOCK} = \mathsf{Some\_Looping}(\mathbf{true}^*,\ i)$$

In GRL models, the above described livelock occurs only if all the input and output actions of one or several blocks are hidden. In such case, livelock can be checked by a static analysis of the GRL specification. This analysis is very fast as it bypasses the generation of the LTS and its exploration by model checking. If we were to generate the LTS, livelock analysis would be carried out in time linear in the size of the LTS, which may however be exponential (or worse) in the code size of the GRL system.

Example 6.14. Consider the GRL system defined below. Its corresponding LTS contains a livelock (i-loop in state 0). In the general case, static livelock checking is an over-approximation, i.e., it may assert the presence of livelocks whereas the actual system is livelock-free.

```
1   system Main (X1, Y1: bool) is
2     var X2, Y2: bool
3     block list
4       B (X1, ?Y1),
5       B (X2, ?Y2)
6   end system
7
8   block B (in X:bool, out Y:bool) is
9     Y := X
10  end block
```



□

Another definition of livelock is the state-based view consisting in preventing a process from performing some particular actions [LcW06]. These actions, called *progress actions*, are generally intended to make the system progress, e.g., deliver outputs or respond to the environment and other components. In terms of LTSs, such a livelock specifies a state from which only non-progress actions are executed indefinitely. That is to say, all progress actions are repeated only a finite number of times. Let $\overline{A}$ be the set of progress actions. Such a livelock is expressed by the following pattern:

$$\mathsf{LIVELOCK}\ (\overline{A}) \quad = \quad \textbf{not}\ \mathsf{Looping}\ (\textbf{not}\ (\bigvee_{A \in \overline{A}} A)^* \ .\ \bigvee_{A \in \overline{A}} A)$$

Example 6.15. The car park progresses by checking that there are continuously entry and exit traffic flows. The progress actions can be output *Open_Park* of the entrance PLC, indicating a car entry; and output *Open* of the exit PLC, indicating a car leaving.

$$\mathsf{LIVELOCK}\ (\{Open\_Park = \textbf{true}\}, \{Open = \textbf{true}\})$$

□

In addition to this general definition, we propose two specific cases for GALS systems. *Activation livelock* considers as progress actions the activation of components. *Data livelock* considers as progress actions the output actions of components.

Activation livelock (starvation)

Modelling concurrency by interleaving may introduce unfair strategies. An example is a component that is consistently ignored, thus never makes progress. Such a situation is called *starvation*. The interpretation of starvation patterns is summarised in Table 6.11.

Example 6.16. Consider the activation policies of blocks *Storey1* and *Storey2*, depicted in Figure 6.2. The left-hand LTS (noted $\mathrm{LTS}_{\mathrm{Default}}$) represents the default activation policy, i.e., without activation constraints. The right-hand LTS (noted $\mathrm{LTS}_{\mathrm{Quasi}}$) represents blocks evolving at the same pace (Example 3.12, page 47). In $\mathrm{LTS}_{\mathrm{Default}}$, while each action can be selected infinitely often, the system can always choose only

action STOREY1 or only action STOREY2. To remove such unfair strategies in GRL models, activation strategies should be implemented, as in $\text{LTS}_{\text{Quasi}}$.

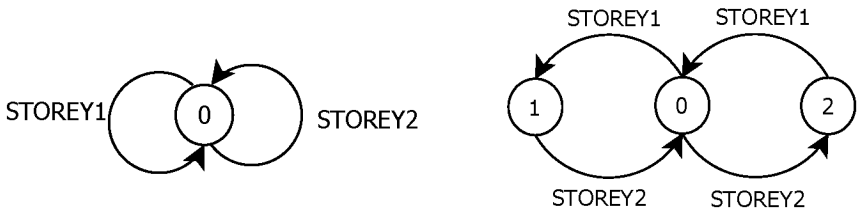


Figure 6.2: LTSs describing activation policies of *Storey1* and *Storey2*

□

We define the *individual starvation* of a component S by the execution of S only a finite number of times. Thus, unfair execution sequences are infinite sequences in which action S is continuously ignored. Thus, unfair situations are states from which actions S are continuously ignored. The absence of such situations is ensured by the following pattern:

Starvation_Freedom (S)

Example 6.17. Consider the LTSs given in Figure 6.2. The individual starvation freedom property is satisfied by $\text{LTS}_{\text{Quasi}}$ but not $\text{LTS}_{\text{Default}}$. For $\text{LTS}_{\text{Default}}$, a counterexample is depicted in Figure 6.3. It indicates that the system can perform an action STOREY1 after which it performs indefinitely only action STOREY2 (state 1). The storey PLC of the first floor is thus in starvation situation. Similarly, the storey PLC of the second floor is in starvation situation in state 2 of the LTS. □

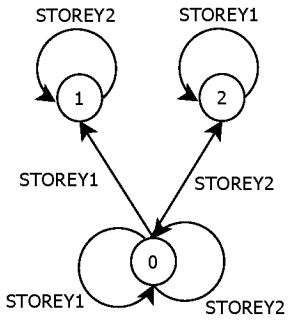


Figure 6.3: Individual starvation of storey PLCs

In addition to individual starvation, we define the starvation of a component w.r.t. another component. This states that a component S' is not indefinitely privileged over another component S. In terms of LTSs, this can be expressed by the absence of an

infinite execution sequence in which only action S' is encountered but never action S. The starvation of S' w.r.t to S is expressed by the following pattern:

Starvation_Freedom (S, S')

This pattern can be extended to encompass sets of components $\overline{S}$ and $\overline{S'}$, as follows:

Starvation_Freedom ($\overline{S}$, $\overline{S'}$)

Example 6.18. In the car park application, the starvation of the exit PLC entails that a car, once inside the car park, can never leave. □

Pattern	Interpretation
Starvation_Freedom (S)	Saturation (**true*** . **not** {S})
Starvation_Freedom (S, S')	Saturation ((**not** {S})* . {S'})
Starvation_Freedom ($\overline{S}$, $\overline{S'}$)	Saturation ((**not** $\bigvee_{S \in \overline{S}}$ {S})* . $\bigvee_{S' \in \overline{S'}}$ {S'})

Table 6.11: Interpretation of starvation patterns

Data livelock (non progress)

We call a data livelock of a component a state from which all output actions are idle. Such a situation is highly undesirable since the system does not progress anymore. There are two possible situations:

- Input idleness has caused output idleness. This situation is equivalent to component idleness, introduced in Section 6.6.1.
- Inputs are not idle. This is (generally) symptomatic to a modelling error, as illustrated in Example 6.19.

Example 6.19. Consider the GRL block below. Condition $pre\_c > 0$ is never satisfied, which pushes the block in a data livelock situation.

```
1  block Inconsistent (in a, b: nat, out c: nat) is
2    static var pre_c: nat := 0
3    if (pre_c > 0) then
4      c := b − a; pre_c := c
5    else
6      c := pre_c
7    end if
8  end block
```

□

We define *consistency* properties asserting that if one or several input actions continue to progress, the same must hold for one or several output actions. The interpretation of consistency patterns is summarised in Table 6.12.

Pattern	Interpretation
Out_Consistent ($\overline{X}$, $\overline{Y}$)	After_Saturation ({$\overline{Y}$=?$\overline{y}$}, Check_X_Y) where Check_X_Y is a regular expression defined as follows: Check_X_Y = (**not** {$\overline{X}$ =?**any**} ∨ **not** {$\overline{Y}$ =?**any**})* . {$\overline{X}$=?$\overline{x}$} . (**not** {$\overline{X}$ =?**any**} ∨ **not** {$\overline{Y}$ =?**any**})* . {$\overline{Y} = \overline{y}$} . (**not** {$\overline{X}$ =?**any**} ∨ **not** {$\overline{Y}$ =?**any**})* . {$\overline{X} \neq \overline{x}$} . (**not** {$\overline{X}$ =?**any**} ∨ **not** {$\overline{Y}$ =?**any**})* . {$\overline{Y} = \overline{y}$}
Some_Consistent ($\overline{\overline{X}}$, $\overline{Y}$)	$\bigvee_{\overline{X} \in \overline{\overline{X}}}$ Out_Consistent ($\overline{X}$, $\overline{Y}$)
All_Consistent ($\overline{\overline{X}}$, $\overline{Y}$)	$\bigwedge_{\overline{X} \in \overline{\overline{X}}}$ Out_Consistent ($\overline{X}$, $\overline{Y}$)
All_Consistent ($\overline{\overline{X}}$, $\overline{\overline{Y}}$)	$\bigwedge_{\overline{Y} \in \overline{\overline{Y}}}$ All_Consistent ($\overline{\overline{X}}$, $\overline{Y}$)

Table 6.12: Interpretation of consistency patterns

The consistency of an output action, whose variables are denoted Y, w.r.t an input action, whose variables are denoted $\overline{X}$, is captured by the property pattern below. The property pattern ensures the absence of infinite execution sequences in which the input action progresses while the output action does not.

Out_Consistent ($\overline{X}$,$\overline{Y}$)

Example 6.20. Consider the block defined in Example 6.19. The consistency of output c w.r.t. inputs a and b is specified as follows:

Out_Consistent ($\langle a, b \rangle$,c)

The property is evaluated to false on the LTS corresponding to block *Inconsistent*. □

Output consistency can be extended to encompass a set of input actions, whose variables are denoted $\overline{\overline{X}}$. The following Some_Consistent (resp. All_Consistent) pattern asserts that if some (resp. all) input actions continue to progress, the same holds for the output action:

Some_Consistent ($\overline{\overline{X}}$, $\overline{Y}$) All_Consistent ($\overline{\overline{X}}$, $\overline{Y}$)

A set of output actions $\overline{\overline{Y}}$ is consistent if the same holds for all the output actions.

All_Consistent ($\overline{\overline{X}}$, $\overline{\overline{Y}}$)

6.6.3 Instability

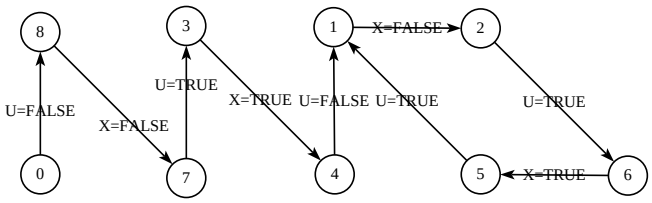
Another case of undesirable situations in synchronous components is *instability*. In a stable component, if there is a state from which the inputs remain idle, both the outputs and the internal state should stabilise in the future, i.e., become idle. Not only the progress of an unstable component is meaningless but also it is neither visible nor controllable by the environment. In [Cas00], Caspi identifies stability as one of the robustness properties to guarantee a correct distribution of synchronous components. The interpretation of stability property patterns is summarised in Table 6.13.

Pattern	Interpretation
Stability $(\overline{\overline{X}}, \overline{Y})$	$(\bigwedge_{\overline{X} \in \overline{\overline{X}}}$ Never (**true*** . $\{\overline{X} = \overline{x}\}$. **true*** . $\{\overline{X} \neq \overline{x}\}$)) **implies** Saturation (**true*** . $\{\overline{Y} = \overline{y}\}$. **true*** . $\{\overline{Y} \neq \overline{y}\}$)
Stability $(\overline{\overline{X}}, \overline{\overline{Y}})$	$(\bigwedge_{\overline{X} \in \overline{\overline{X}}}$ Never (**true*** . $\{\overline{X} = \overline{x}\}$. **true*** . $\{\overline{X} \neq \overline{x}\}$)) **implies** $\bigwedge_{\overline{Y} \in \overline{\overline{Y}}}$ Saturation (**true*** . $\{\overline{Y} = \overline{y}\}$. **true*** . $\{\overline{Y} \neq \overline{y}\}$)

Table 6.13: Interpretation of stability patterns

Example 6.21. Consider the GRL block *Unstable* below. Assume input u starts changing and there exists a state from which it remains true forever. Even though u remains unchanged, the output x indefinitely oscillates between values *true* and *false*. Instability is illustrated by the cycle of states {1, 2, 6, 5} in the LTS below, corresponding to block *unstable*.

```
1  block Unstable
2    (in  u: bool,
3     out x: bool)
4  is
5    static var pre_x := true
6    x     := u and not pre_x;
7    pre_x := x
8  end block
```



□

To detect such an undesirable divergence, we define the *stability* property asserting that the idleness of inputs should imply the idleness of outputs. In terms of LTSs, this can be expressed by first detecting the states after which all input actions are idle, then checking that the output actions progress only a finite number of times. This is expressed by the following pattern, where $\overline{\overline{X}}$ and $\overline{Y}$ stand for the respective variables of a set of input actions and an output action:

$$\text{Stability } (\overline{\overline{X}}, \overline{Y})$$

Stability can be extended to encompass a set of output actions, whose variables are denoted $\overline{\overline{Y}}$.

$$\text{Stability } (\overline{\overline{X}}, \overline{\overline{Y}})$$

Example 6.22. Consider the block defined in Example 6.21. The stability of output x is specified as follows:

$$\text{Stability } (u, x)$$

□

6.7 Discrete real-time properties

> Any verification using model-based techniques is only as good as the model of the system.
>
> Principles of Model Checking

A GALS system may depend on real-time constraints. In this case, its correctness depends, in addition to computational results, on the time at which results are produced. In a redundant system, it is vital that on failure detection, the function is passed from primary components to secondary ones shortly.

When real-time constraints are required, relative (discrete) time can be measured in terms of the number of component steps. In this section, we present two types of real-time properties. *Component real-time properties* involve individual components. *System real-time properties* involve several components. The interpretation of real-time patterns is summarised in Table 6.14.

Pattern	Interpretation
Deadline (R, A_1, A_2, n)	Never (R . (**not** A_1)* . (A_1 . (**not** (A_1 **or** A_2))*)$\{n+1\}$)
Sustain (R, A_1, A_2, n)	[R] **nu** *Count* (c: **nat** := *1*) . (((c < n) **implies** ([A_2] **false** **and** [A_1] *Count* ($n+1$))) **and** [**not** (A_1 **or** A_2)] *Count* (c))
From_To_Most (A_1, A_2, A_3, n)	Deadline (**true*** . A_1, A_3, A_2, n)
From_To_Least (A_1, A_2, A_3, n)	Sustain (**true*** . A_1, A_3, A_2, n)

Table 6.14: Interpretation of real-time patterns

Component real-time properties

For an individual component, the time difference between any pair of actions can be interpreted as a multiple of the component steps. For example, consider the property: *whenever a failure occurs, an alarm should be raised in at most 30 seconds*. Assuming the component period is 5 seconds, the 30 second delay corresponds to 6 block steps.

A typical instantiation of component real-time properties is *deadlines*. A deadline property asserts that whenever a certain action occurs, it must be followed by another action in a bounded future. Consider (i) an action A_1; (ii) a natural number n, quantifying the deadline in terms of occurrences of A_1; (iii) an action A_2, to occur necessarily before reaching the deadline; and (iv) a regular formula R specifying the condition triggering the countdown. The following pattern forbids the presence of execution sequences, in which there are subsequences satisfying R, followed by $n+1$ occurrence of A_1, without the occurrence of A_2.

Deadline (R, A_1, A_2, n)

Example 6.23. Assume the exit PLC period is 1 second. The property: *once a car asks to leave the car park, the exit gate must open in less than 15 seconds* can be expressed as follows:

Deadline (**true*** . {Cmd_P1 =? *cmd1*:**bool**, Cmd_P2 =? *cmd2*:**bool where** *cmd1*<>*cmd2*},
{*Exit*},
{Open = **true**},
15)

□

Another instantiation of component real-time properties is the sustain of an event for a bounded time duration. Consider: (i) an action A_1, corresponding to the event to sustain; (ii) a natural number n, quantifying the sustain duration in terms of occurrences of A_1; (iii) an action A_2, to not occur before the end of the sustain duration; and (iv) a regular formula R specifying the condition triggering the sustain. The following pattern states that all execution sequences containing subsequences satisfying R must lead to action A_2 before n occurrences of action A_1.

Sustain (R, A_1, A_2, n)

Example 6.24. The informal specification of the car park application (Section 3.1, page 29) states that: *If the access is granted, the entrance gate remains open for a fixed amount of time and a yellow light is turned on until the gate closure.* The property can be expressed as follows, where n encodes the "amount of time":

Sustain (**true***. {Open_Park = **true**},
{Green =? **any**, Yellow = **true**, *Red* =? **any**},
{Green =? **any**, Yellow = **false**, *Red* =? **any**},
n)

□

System real-time properties

The correctness of system real-time properties depends on the accuracy of activation constraints. The finest abstraction encountered in the literature to express activation constraints in a discrete model of time is the generalisation of the quasi-synchronous approach (see Chapter 7, section 7.1, for implementations in GRL and muGRL). In such systems, requirements match the pattern: *within a given time-interval, some event should occur a bounded number of times.* The notion "time-interval" is interpreted in terms of action occurrences. The following patterns assert that whenever an action A_1 occurs, an action A_2 may occur at least m (resp. at most m) times before the next occurrence of A_3:

From_To_Least (A_1, A_2, A_3, m) From_To_Most (A_1, A_2, A_3, m)

Examples will follow in Chapter 7.

6.8 Conclusion

This chapter identifies an open-ended set of properties for GALS systems. We attempted to collect the most recurring properties in the literature of synchronous and asynchronous systems and to accommodate them to GALS systems. In particular, the proposed properties exploit the activation strategies and data-handling that a GALS system exhibit, as well as some discrete real-time aspects. We hope our study would contribute to the state-of-the-art verification of GALS systems. Its principles would be transferable to any temporal logic as well, provided the temporal logic supports regular expressions and data handling.

Properties are encoded by means of a system of patterns, muGRL. Our aim is to guide potential users in the verification task, cutting down the learning effort. As regards the implementation of muGRL, patterns with a fixed number of parameters can be encoded using MCL macros then stored in reusable libraries. This provides the user with templates whose parameters should be filled in. Here are some examples:

```
1  macro Not_To_Unless (A, B, C) =
2      [true*. A. (not (C))*. B] false
3  end_macro
4
5  macro always_some (A) =
6     [true*] <true*. A> true
7  end_macro
8
9  macro Sustain (R, A1, A2, n) =
10   [ R ] nu Counter (c: nat := 1) . (
11           ((c < n) implies ([ A2 ] false and [ A1 ] Counter (c + 1)))
12           and [ not (A1 or A2) ] Counter (c)
13          )
14 end_macro
```

At the time of writing, the complete translation into MCL is not yet automated.

Chapter 7

Formal Modelling and Verification of GALS Applications

This chapter presents applications of GRL, muGRL, and CADP to model and verify concrete GALS systems. The first application is quasi-synchronous systems, which involve a bounded level of nondeterminism. In a related concern, we give insights on the way deterministic applications can be described. The second application is a simplified version of an *AutoFlight Control System* (AFCS), provided by *Thales Avionics*. We use the AFCS to illustrate various modelling, generation, and verification scenarios and techniques. Finally, we report our industrial experience with *Crouzet Automatismes* in the framework of the Bluesky industrial project.

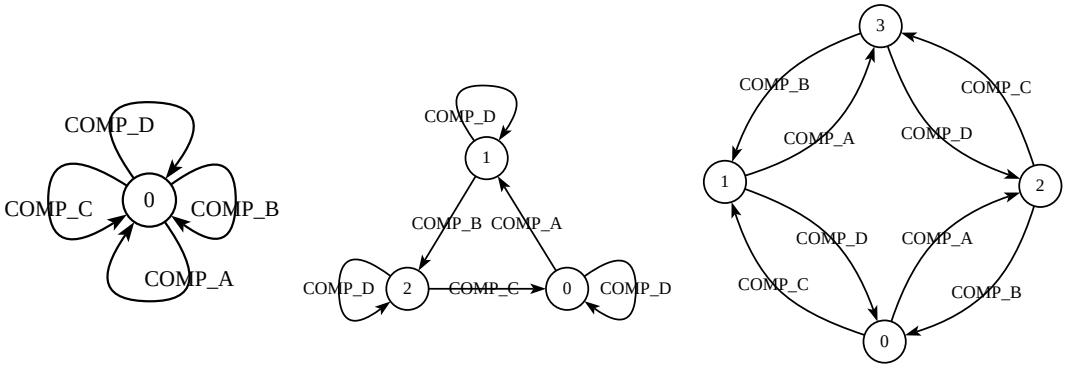
7.1 Quasi-synchronous systems

A quasi-synchronous system is one whose synchronous components are not governed by a common clock but evolve almost at the same pace, or multiples of the same pace. The quasi-synchronous abstraction, first proposed by Caspi [Cas00] in the 2000's then formalised in [HM06], states that a component clock cannot tick more than twice before all other component clocks have ticked at least once. Stated differently, each component clock cannot deviate more than one tick. We use the term *basic quasi-synchrony* to denote this abstraction. Recent works [BMY+14, BBP15] have studied generalisations of the quasi-synchronous abstraction. We consider a generalised version in which each component clock cannot tick more than an upper bound before all other component clocks have ticked at least once. We use the term *generalised quasi-synchrony* to denote this abstraction.

In GRL, without activation constraints, a block may perform infinitely many activations between two successive activations of another block. This situation becomes unrealistic when the aim is to describe clock constraints. For illustration, consider the activa-

tion strategies depicted in Figure 7.1, where *COMP_A*, *COMP_B*, *COMP_C*, and *COMP_D* denote blocks.

- In case (a), no constraint is put on block activations. A block can execute arbitrarily often without any other block executing meanwhile.
- In case (b), blocks *COMP_A*, *COMP_B*, and *COMP_C* are constrained so as to execute in this specific order but no constraint is put on the activation of block *COMP_D*. As a result, block *COMP_D* can execute arbitrarily often from each state in the LTS.
- In case (c), constraints on the pair (*COMP_A*, *COMP_B*) and on the pair (*COMP_C*, *COMP_D*) are described independently, e.g., in different environments. Again, (*COMP_A*, *COMP_B*) can execute arbitrarily often between two successive executions of (*COMP_C*, *COMP_D*), and conversely.



(a) (b) (c)

Figure 7.1: LTSs describing different activation strategies for a set of blocks

In the sequel, we propose two implementations of quasi-synchrony in GRL and discuss the relation between both implementations. To this aim, all blocks used in this section are instances of the following toy block:

```
1  block Bool_Id (in X: bool, out Y: bool) is
2    Y := X
3  end block
```

7.1.1 Primary implementation

This section proposes a primary implementation of quasi-synchrony, with a focus on the basic version. To this aim, we consider a system, named *Basic_Two*, encapsulating two blocks *COMP_A* and *COMP_B*. We also consider another system, named *Basic_Three*, with three blocks *COMP_A*, *COMP_B*, and *COMP_C*.

Property specification in muGRL Basic quasi-synchrony with two blocks requires the following properties to be fulfilled:

(P1) It is possible for action *COMP_A* (resp. *COMP_B*) to not occur between two successive occurrences of action *COMP_B* (resp. *COMP_A*)
(P2) It is possible for action *COMP_A* (resp. *COMP_B*) to occur once between two successive occurrences of action *COMP_B* (resp. *COMP_A*)
(P3) It is possible for action *COMP_A* (resp. *COMP_B*) to occur twice between two successive occurrences of action *COMP_B* (resp. *COMP_A*)
(P4) Action *COMP_A* (resp. *COMP_B*) cannot occur more than twice between two occurrences of action *COMP_B* (resp. *COMP_A*)
(P5) Actions *COMP_A* and *COMP_B* occur infinitely often

The formalisation of properties P1, P4, and P5 in muGRL is summarised in the following table:

Property	Formalisation
P1	Some (**true*** . {*COMP_A*} . (**not** {*COMP_B*})* . {*COMP_A*}) Some (**true*** . {*COMP_B*} . (**not** {*COMP_A*})* . {*COMP_B*})
P4	From_To_Most (**true*** . {*COMP_A*}, {*COMP_A*}, {*COMP_B*}, 2) From_To_Most (**true*** . {*COMP_B*}, {*COMP_B*}, {*COMP_A*}, 2)
P5	Looping ((**not** {*COMP_A*})* . {*COMP_A*}) Looping ((**not** {*COMP_B*})* . {*COMP_B*})

Modelling in GRL A first way to implement basic quasi-synchrony has been proposed in Example 3.12 (page 47). We give below an enhanced version, parameterised with relative block paces (*maxA* and *maxB*). The default values correspond to basic quasi-synchrony. When block *A* is activated *maxA* times, block *B* is activated *maxB* times, in the meanwhile. The order in which block activations are achieved is arbitrary.

```
1   environment Primary_Quasi_2 {maxA: nat := 1, maxB: nat := 1}
2                               (block A, B)
3   is
4     -- maxA, maxB should be >= 1
5     static var countA, countB: nat  := 0
6     select  -- activate A, if allowed, maxA times
7       if (countA < maxA) then
8         enable A;
9         countA := countA + 1;
10      end if
11    [] -- activate B, if allowed, maxB times
12      if (countB < maxB) then
13        enable B;
14        countB := countB + 1;
15      end if
16    end select;
17    if (countA >= maxA) and (countB >= maxB)  then -- reinitialise
```

```
18        countA := 0; countB := 0
19      end if
20    end environment
```

We use environment *Primary_Quasi_2* to implement basic quasi-synchrony inside a system as follows:

```
1   system Basic_Two (X1, Y1, X2, Y2 :bool) is
2     alias Bool_Id as Comp_A, Bool_Id as Comp_B
3     block list
4       Comp_A (X1, ?Y1),
5       Comp_B (X2, ?Y2)
6     environment list
7       Primary_Quasi_2 {_, _}(Comp_A, Comp_B) — default values are considered
8   end system
```

Basic LTS generation To visualise block activations in a GRL system, the simplest way is to first generate the system LTS. We use GRL.OPEN and the GENERATOR tool of CADP, as given by the following Bourne shell command:

```
1   % grl.open -showall -root "Basic_Two" Primary_Quasi.grl generator Basic_Two.bcg
```

Remark 7.1. Basic LTS generation requires the generation of the system LTS to succeed. We will propose, in Section 7.3.4, an alternative scenario, which can be used when LTS generation fails or lasts too long. ■

Remark 7.2. In the general case, activation constraints may be distributed in several environments and combined with data constraints. Debugging the activation strategy on the complete system LTS could be difficult and cumbersome, in particular when the system LTS is large. It is then of interest to visualise the activation strategy of blocks, regardless the data carried by their input and output actions. ■

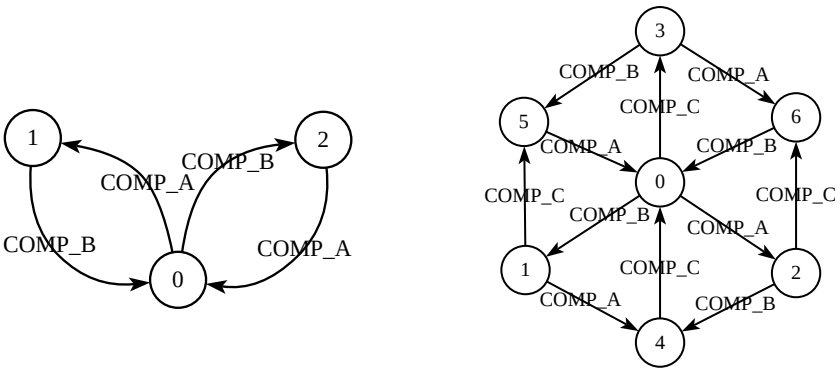
Since we focus on block activations, we consider only actions *Start*. In the LTS corresponding to system *Basic_Two*, we hide all actions but *Start* and rename the remaining actions by removing prefix "Start !" introduced by GRL2LNT. Finally, we reduce the LTS modulo divbranching bisimulation to remove irrelevant hidden actions while preserving the branching structure of the LTS. All these steps are performed using the following SVL statement:

```
1   "Basic_Two.bcg" = divbranching reduction of
2                       total rename "START !\(.*\)" -> "\1" in
3                         gate hide all but "START" in "Basic_Two.bcg";
```

For systems *Basic_Two* and *Basic_Three*, Figure 7.2 depicts the activation strategy of blocks. In both LTSs, there is a central state (state 0), from which all outgoing and ingoing transition sequences contain exactly one activation of each block. This

situation is equivalent to the presence of an "abstract global clock", the ticks of which are represented by two successive occurrences of the central state when traversing the LTS. Due to the interleaving of block activations, each block may perform two steps between two successive steps of each other component. This is illustrated by the following action sequence in the left-hand LTS.

$$0 \xrightarrow{COMP\_B} 2 \xrightarrow{COMP\_A} 0 \xrightarrow{COMP\_A} 1 \xrightarrow{COMP\_B} 0$$



two blocks three blocks

Figure 7.2: LTS describing the activation strategy of basic quasi-synchrony (primary implementation)

Verification results Checking that properties P1, ..., P5 hold on the LTSs depicted in Figure 7.2 is straightforward.

7.1.2 Refined implementation

This section proposes a refined implementation of quasi-synchrony.

Modelling in GRL Instead of considering an "abstract global clock" and counting the activations of each block w.r.t. clock ticks, we propose to count the activations of each block since the last activation of each other block. This is implemented in environment *Refined_Quasi_2*, given below, for two blocks.

```
1   environment Refined_Quasi_2 {maxA: nat := 2, maxB: nat := 2}
2                               (block A, B)
3   is
4     static var A_since_B: nat := 0,
5                B_since_A: nat := 0
6     select
7       if (B_since_A < maxB) then
8         enable B;
9         A_since_B := 0;
10        B_since_A := B_since_A + 1
```

```
11        end if
12      []
13        if (A_since_B < maxA) then
14          enable A;
15          B_since_A := 0;
16          A_since_B := A_since_B + 1
17        end if
18      end select
19    end environment
```

The environment is parameterised with the maximal number of activations of a block between two successive activations of the other block. The default configuration of parameters corresponds to basic quasi-synchrony.

We use environment *Refined_Quasi_2* to implement basic quasi-synchrony inside a system as follows:

```
1   system Main (X1, Y1, X2, Y2: bool) is
2     alias Bool_Id as Comp_A, Bool_Id as Comp_B
3     block list
4       Comp_A (X1, ?Y1),
5       Comp_B (X2, ?Y2)
6     environment list
7       Refined_Quasi_2 {_, _}(Comp_A, Comp_B)
8   end system
```

Property specification in muGRL The refined implementation of generalised quasi-synchrony requires the following properties to be fulfilled:

(P6) There are at most *maxA* occurrences of action *COMP_A* between two successive occurrences of action *COMP_B*
(P7) The activation strategy introduces no deadlock
(P8) Blocks *COMP_A* and *COMP_B* are always alive

The formalisation of these properties in muGRL is summarised in the following table:

Property	Formalisation
P6	From_To_Least (**true*** . {*COMP_B*}, {*COMP_B*}, {*COMP_A*}, *maxA*) From_To_Least (**true*** . {*COMP_A*}, {*COMP_A*}, {*COMP_B*}, *maxB*)
P7	Deadlock (*COMP_A*, *COMP_B*)
P8	All_Alive (*COMP_A*, *COMP_B*)

LTS generation and verification The activation strategy of blocks is given in Figure 7.3. Checking that properties P6, P7, and P8 hold on the LTS is straightforward.

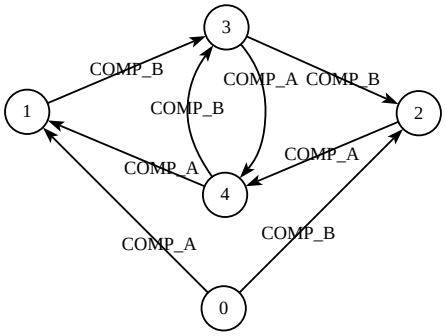


Figure 7.3: LTS describing the activation strategy of basic quasi-synchrony (refined implementation)

7.1.3 Discussion

We discuss the relation between our primary and refined implementations of quasi-synchrony, proposed in the previous sections. Both of them are correct implementations of quasi-synchrony, according to the verification results. We exploit the equivalence between LTSs depicted in Figure 7.2 (noted $\text{LTS}_{\text{primary}}$) and Figure 7.3 (noted $\text{LTS}_{\text{refined}}$). Such equivalence, modulo branching bisimulation, is checked using the following SVL statement:

```
"Equiv_Quasi.bcg" = branching comparison "Quasi_Gen.bcg" == "Quasi_Basic.bcg";
```

The LTSs are not equivalent, unexpectedly. The counterexample, produced in file "Equiv_Quasi.bcg", states that the following transition sequence is present in $\text{LTS}_{\text{refined}}$ but absent in $\text{LTS}_{\text{primary}}$.

$$0 \xrightarrow{COMP\_A} 1 \xrightarrow{COMP\_B} 2 \xrightarrow{COMP\_A} 3 \xrightarrow{COMP\_A} 4 \xrightarrow{COMP\_B} 5$$

Nonetheless, $\text{LTS}_{\text{primary}}$ is included in $\text{LTS}_{\text{refined}}$. Such inclusion, modulo the preorder corresponding to branching bisimulation, is checked using the following SVL statement:

```
"Inclu_Quasi.bcg" = branching comparison "Quasi_Adv.bcg" >= "Quasi_Basic.bcg";
```

We conclude that $\text{LTS}_{\text{refined}}$ is more general than $\text{LTS}_{\text{primary}}$ and can be seen as a refinement of $\text{LTS}_{\text{primary}}$.

7.2 Deterministic GALS systems

A deterministic GALS system is one in which messages are delivered in the order in which they have been received, without message loss.

In GRL, message loss, if not explicitly modelled (see Example 3.16, page 50), is caused by discrepancies between the rates of message submission by a block emitter and message delivery by a block receiver; such discrepancy is induced by the arbitrary activation

of blocks. Consider two blocks evolving under basic quasi-synchrony (see Section 7.1) and communicating via reliable mediums with single-place buffers. Two successive activations of a block emitter, without the activation of the block receiver in between, causes message overwriting.

To model deterministic GALS systems in GRL, an activation strategy of blocks should be implemented. This entails bounded interleaving between block activations, making message loss also bounded. Then, a buffering mechanism with well-chosen dimensions sould be implemented. Such dimensions should comply with the activation strategy of blocks. For basic quasi-synchrony, at most two message submissions may occur between two message deliveries in each transmission, and conversely. Hence, a double-place FIFO is sufficient to ensure message transmission without loss.

7.3 AutoFlight Control System (AFCS)

This section reports our experience in modelling and verifying a simplified part of an *AutoFlight Control System* (*AFCS*). The system is provided by *Thales Avionics* in a collaboration with IRT-Saint Exupéry. Our goal here is not to model a complete AFCS, but to study the way such a system with stringent timing requirements can be abstracted in GRL. The part of the *AFCS* we address has been first studied in [BBJ14] using the Fiacre/Tina toolbox[1] for Time Petri Nets.

In the sequel, we first present an overview of the target system. Then, we study the modelling, state-space generation, and verification phases of separate components and of the whole system.

7.3.1 Overview of the system

The AFCS improves the quality of flight and enhances the operational capability of the aircraft, e.g., by guiding the aircraft on a defined trajectory. The architecture of the AFCS is of type *dual COM/MON*. In such an architecture, each function is performed by two *channels*, in *hot redundancy*: a *master* channel and a *slave* channel, such that only the commands emitted by the master are taken into account by the rest of the system. If the master channel fails, it is deactivated and the commands emitted by the slave will be considered.

Each channel is itself divided into two channels: a *command channel* (COM), which implements the expected functionality, and a *monitoring channel* (MON), which checks whether the command channel operates correctly.

We focus on a simple use case: the altitude target acquisition. Our target system, depicted in Figure 7.4, comprises two parts:

[1] http://projects.laas.fr/tina/

- The *FCP* (*Flight Control Panel*) enables the pilot to interact with the system. *FCP_COM* and *FCP_MON* are cadenced by the same clock.
- The AFS (*Automatic Flight System*) acquires the altitude target. *AFS_COM* and *AFS_MON* communicate asynchronously using an AFDX communication medium.

Components communicate by sampling, i.e., only the most recent message is read by the receiver, message queuing being not supported. Asynchronous communication mediums are assumed reliable.

The behaviour of the system is as follows. The pilot sets the target altitude by rotating a knob on the FCP. Two different simultaneous informations are sent from *FCP* to *AFS*:

- *FCP_COM* sends, via an *A429* bus, a value quantifying the knob rotation to *AFS_COM*. In turn, *AFS_COM* computes an altitude order, based on the received value and sends the order to *AFS_MON* for validation purpose.
- *FCP_MON* sends a Boolean value indicating a movement detection to *AFS_MON*, using a discrete signal. This information enables *AFS_MON* to check the validity of the altitude order received from *AFS_COM* and to provide it with a verdict. The order is considered valid if: (i) it didn't change from the previous step or (ii) it has changed and a movement has also been detected by *FCP*. In case of invalidity, a presumed correct value of the altitude order is considered.

Because of the asynchrony in the system architecture, *AFS_MON* may wrongly invalidate a correct order. To cope with such asynchrony, components *FCP_MON* and *AFS_MON* sustain the movement detection information during some specific amount of time, in order to make the information observable by other components, independently

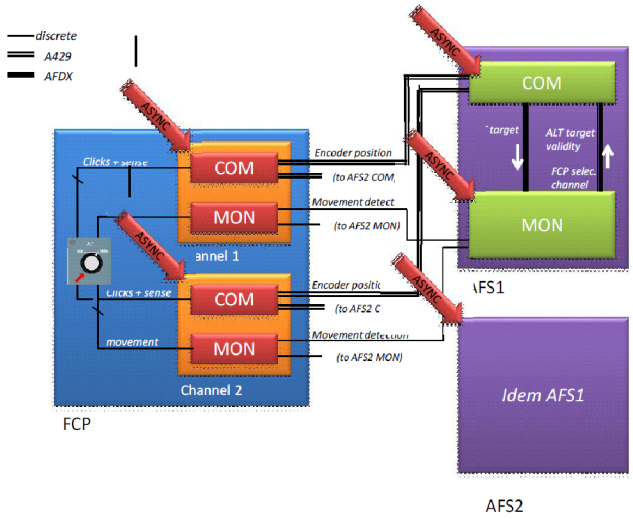


Figure 7.4: Architecture of the AFCS

of the asynchronous execution of the system components.

In the sequel, we address the modelling and verification of the *FCP* and *AFS* components along with the *AFCS* system. Instead of giving complete GRL code, which is available in Appendix A, we give code excerpts and schematic representations.

We consider the following real-time constraints. Components *FCP_MON*, *AFS_MON*, and *AFS_COM* evolve at periods 25ms, 50ms, and 150ms, respectively. *FCP* (resp. *AFS_MON*) sustains the movement detection information for 13 (resp. 6) steps of *FCP* (resp. *AFS_MON*). Transmission delays between components are significantly shorter than the periods of components.

To verify the *AFCS*, we specified the following properties:

(P1) Blocks do not halt
(P2) There is no block in starvation situation
(P3) The system may deadlock, i.e., both the master and the slave channels may fail
(P4) All input and output actions of components progress
(P5) No movement detection information is sent to *AFS* if no knob rotation has occurred
(P6) The movement detection information is sustained for at least 13 steps of *FCP*
(P7) The movement detection information is sustained for exactly 13 steps of *FCP*, if no knob rotation has occurred ever since
(P8) A change in the knob position is always accompanied by a movement detection
(P9) A movement detection information sent by *FCP* is received by *AFS*
(P10) The countdown to sustain a movement detection information in *AFS_MON* is always set to value 6 when a movement is detected
(P11) Movement detection information is sustained for 6 steps of *AFS_MON*, if no new movement detection has occurred ever since
(P12) A movement detected in *FCP* is sustained enough to be observed by *AFS_MON*
(P13) No new altitude order is provided by *AFS_COM* unless a movement has been detected in *AFS_MON*

7.3.2 Modelling and verifying component FCP

Modelling in GRL The FCP component is modelled by the GRL block *FCP* below, a schematic view of which is given in Figure 7.5. Block *FCP* encapsulates three subblocks corresponding to a movement encoder (subblock *Encoder*), the monitoring channel (subblock *CP_MON*), and the command channel (subblock *CP_COM*). According to GRL semantics, those subblocks evolve by default at the same pace, the one of the higher-level block *FCP*.

```
1  — Default duration to sustain the movement detection information
2  const default_fcp_mvt_prolong :nat := 13
3
4  block FCP {fcp_mvt_prolong: nat := default_fcp_mvt_prolong}
5            (in   pilot_rot      : int,
```

```
6                   pilot_mvt       : bool)
7              [send to_afs_position: int,
8               send to_afs_mvt      : bool]
9   is
10    var click_nb     : int,
11        detected_mvt: bool
12    Encoder (pilot_rot, pilot_mvt,
13             ?click_nb, ?detected_mvt);
14    CP_COM  (click_nb, ?to_afs_position);
15    CP_MON  {fcp_mvt_prolong} (detected_mvt, ?to_mon_mvt)
16  end block
```

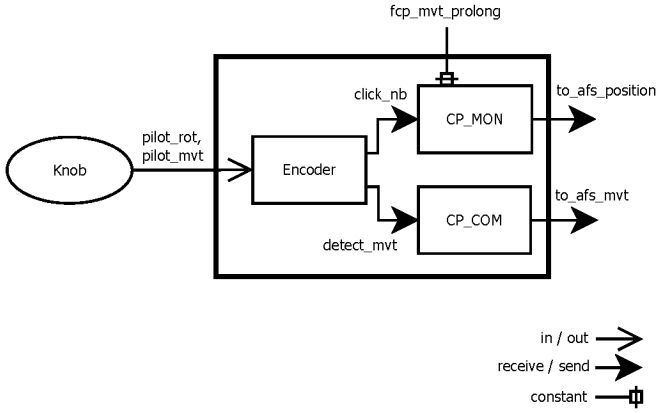


Figure 7.5: Schematic view of the *FCP* components

The block is parameterised with *fcp_mvt_prolong*, which represents the duration to sustain the movement detection information. Such duration is necessarily a multiple of the block period. We set the default value to 13.

We abstract the behaviour of the knob rotation using environment *Knob* below. For simplicity, only integers varying from -2 to 2 are considered to quantify the rotation.

```
1   environment Knob (out click_number: int, mvt: bool) is
2     static var pre_click_number: int := 0
3     when <click_number, mvt> ->
4     select
5       -- The knob position changes
6       click_number := any int where ((click_number <= 2) and (click_number >= -2));
7       pre_click_number := click_number; -- store position
8       mvt := true
9     [] -- The knob position does not change
10      click_number := pre_click_number;
11      mvt := false
12    end select
13  end environment
```

LTS generation To generate and minimise the LTS corresponding to *FCP*, we encapsulate the block with environment *Knob* inside a parameterised system named *Main_FCP*. Once the LTS is constructed, we perform a minimisation modulo the divbranching bisimulation. All these steps are done by the following SVL script:

```
% grl.open -showall -root "Main_FCP (13 of nat)" FCP.grl generator FCP_Orig.bcg
 "FCP.div.bcg" = divbranching reduction of
                    total rename "GATE_\(.*\)" -> "\1" in "FCP.Orig.bcg";
```

The LTS, produced in file *FCP.Orig*, is small enough to be generated directly. It contains 802 states and 1157 transitions with a deterministic behaviour for all actions. The LTS minimisation modulo divbranching bisimulation yields an LTS with 315 states and 640 transitions.

Property formalisation and verification results Properties P4, ..., P8 capture the behaviour of *FCP*. The formalisation of properties P4, P6, and P8 in muGRL is summarised in the table below. All properties hold on the system LTS.

Property	Formalisation in muGRL
P4	Progress (*pilot_mvt*, *pilot_rot*) Progress (*to_afs_position*) Progress (*to_afs_mvt*)
P6	Sustain (**true***. {*pilot_rot* =? **any**, *pilot_mvt* = **true**}. (**not** {*pilot_rot* =? **any**, *pilot_mvt* =? **any**})*. {*pilot_rot* =? **any**, *pilot_mvt* = **false**}, {*afs_mon_mvt* = **true**}, {*afs_mon_mvt* = **false**}, 13)
P8	Never (**true***. {*pilot_rot* =? *rot1*:**int**, *pilot_mvt* =? *mvt1*:**bool**}. (**not** {*pilot_rot* =? **any**, *pilot_mvt* =? **any**})*. {*pilot_rot* =? *rot2*:**int**, *pilot_mvt* =? *mvt2*:**bool where** (*rot1* <> *rot2*) **and** (*mvt2* = **false**)})

7.3.3 Modelling and verifying component AFS

Modelling in GRL The command and the monitoring channels of *AFS* are modelled in GRL by blocks *AFS_COM* and *AFS_MON*, as depicted in Figure 7.6. Output *interrupt* of block *AFS_MON* indicates whether, despite the absence of movement detection by *FCP*, *AFS_COM* has computed a new target altitude. *AFS_MON* is three times faster than *AFS_COM*. Such activation constraint is modelled in environment *AFS_Act*, following the modelling proposed in Section 7.1.1. This is done as follows:

```
alias Primary_Quasi_2 {3, 1} as AFS_Act
...
AFS_Act (AFS_MON, AFS_COM)
```

The AFDX communication bus is modelled by two GRL mediums *AFDX_COM_to_MON* and *AFDX_MON_to_COM*. An additional medium *Stub*, given below, is connected to *AFS_COM*. It simulates the behaviour of *FCP_COM* by producing integer numbers ranging from -2 to 2. This enables a realistic modelling and helps to keep small the state space of *AFS*.

```
1  medium Stub [send from_cp_rot:int] is
2    when from_cp_rot ->
3      from_cp_rot := any int where ((from_cp_rot <= 2)
4                                   and  (from_cp_rot >= -2))
5  end medium
```

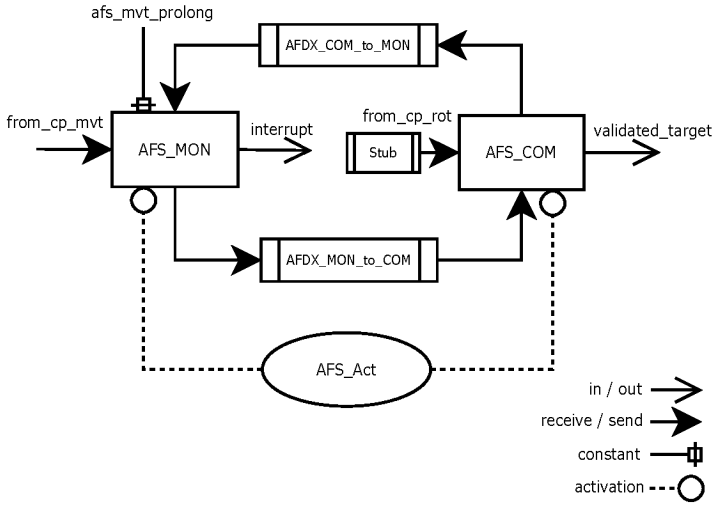


Figure 7.6: Schematic view of the *AFS* components

Remark 7.3. Transmission delays between *AFS_COM* and *AFS_MON* are significantly shorter than the component periods. According to observations of Caspi [Cas00], such delays can be modeled by a unit of the global clock, i.e., the shortest measurable delay in a synchronous discrete-time model. Thus, if a message has been emitted at an instant t of the global clock, the message will be received at instant t+1, which represents non-instantaneity. In GRL, the emission of a message by a block and its reception by another block are by default non-instantaneous. This is granted by asynchrony. ■

LTS generation We first generate the LTS corresponding to each *AFS* component separately then generate the one corresponding to the whole *AFS*. Table 7.1 reports the size of LTSs in terms of number of states and transitions.

Note that the separate generation of system LTSs yields relatively large LTSs for blocks *AFS_MON* and *AFS_COM* whereas the generation of the whole *AFS* at once leads

	Non-minimised		Divbranching min.	
	States	Transitions	States	Transitions
AFS_MON	929,281	68,691,969	265,227	17,043,978
AFS_COM	197,377	33,751,553	196,864	33,751,040
AFDX_MON_to_COM	513	263,169	512	262,656
AFDX_COM_to_MON	257	66,049	256	65,792
Stub	1	5	1	5
AFS_Act	7	10	7	10
AFS	6,867	8,370	577	750

Table 7.1: LTS sizes of *AFS* components and of whole *AFS*, before and after minimisation

to a much smaller LTS. Each block is constrained by its connected mediums and environments, which are themselves constrained by other connected blocks, and so on. Accordingly, many states of components are never explored, since they are irrelevant for the current *AFS* composition. For example, block *AFS_MON* defines a receive channel containing a variable of type integer to be connected to *AFS_COM*, through medium *AFDX_COM_to_MON*. The LTS of *AFS_MON* considers all possible values, while in the current *AFS* only values ranging from -2 to 2 are used.

Property formalisation and verification results Properties P1, P2, and P4 capture the observable behaviour of *AFS*, i.e., activation strategy and data carried by block inputs and outputs. All of them hold on the system LTS. Properties P10 and P11 capture the non-observable behaviour of *AFS_MON*, i.e., the internal computations. We address here the formalisation of P10 and P11.

To express such properties, we take inspiration from synchronous observers. Properties can be specified by means of additional GRL subblocks, which we call *observers*. For example, property P10 can be expressed as follows:

```
1  — Default duration to sustain the movement detection information
2  const default_afs_mvt_prolong :nat := 6
3
4  block Observer_P10 {init_count: nat := default_afs_mvt_prolong}
5                     (in  cp_mon_mvt: bool, mvt_prolong: nat, out ok: bool)
6  is
7    if (cp_mon_mvt) then — a movement is detected
8      — check counter "mvt_prolong" against value "init_count"
9      ok := (mvt_prolong == init_count)
10   else
11     ok := true
12   end if
13 end block
```

The observer emits an alert, value *false* on output *Ok*, whenever the desired property is violated. Then, observers are invoked inside the block under verification to monitor its internal behaviour. Here is an excerpt of block *AFS_MON* after adding the invocations of observers; a schematic representation is given in Figure 7.7.

```
1  block AFS_MON (..., out ok_P10, ok_P11: bool)
2                  ...
3  is
4    ... -- internal behaviour of AFS_MON
5    -- observers
6    Observer_P10 {afs_mvt_prolong}(cp_mon_mvt, detected_mvt, ?ok_P10);
7    Observer_P11 {afs_mvt_prolong}(cp_mon_mvt, countdown, ?ok_P11)
8  end block
```

To make the truth values of observers visible on the LTS, their outputs are connected to additional outputs of block *AFS_MON*.

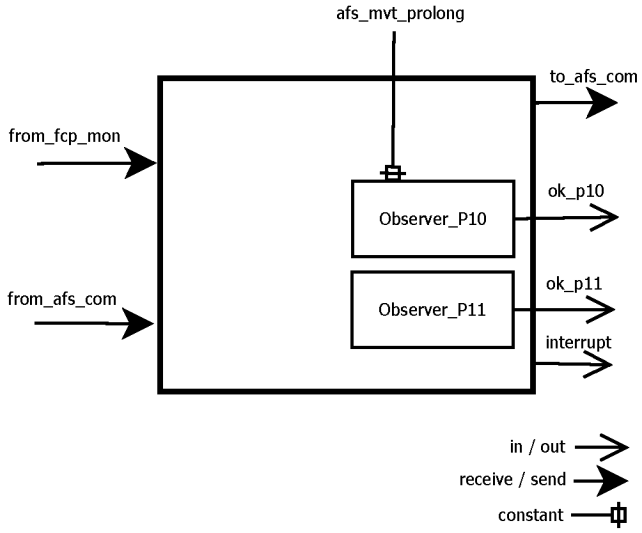


Figure 7.7: Adding observers to *AFS_MON*

Finally, the validity of the properties encoded by observers can be simply determined by visual checking. It suffices to list the actions in the LTS of *AFS*, using a dedicated option in CADP, and check whether the observer outputs raise a problem. Below is an excerpt of the actions list of the system LTS. The whole list shows that variables *ok_p10* and *ok_p11*, corresponding to observer outputs, never take value *false*, meaning that the respective properties hold on the LTS.

```
from_fcp_com = -2      from_fcp_com = 1       from_fcp_mvt = true
from_fcp_com = -1      from_fcp_com = 2       interrupt = false
from_fcp_com = 0       from_fcp_mvt = false   ok_p10 = true, ok_p11 = true
```

Alternatively, the validity of properties P10 and P11 can be checked by formulation in muGRL:

Never ({ok_p10=**false**, ok_p11=**?any**} **or** {ok_p10=**?any**, ok_p11=**false**})

Remark 7.4. Writing down observers in GRL increases the size of the system LTS, since additional transitions are required to visualise the truth values of the properties. For properties P10 and P11, Table 7.2 summarises the size of the LTSs of observers and their impact on the size of the system LTS.

	Non-minimised		Divbranching min.	
	States	Transitions	States	Transitions
Observer_P10	58	94	17	35
Observer_P11	6	517	3	514
AFS without observers	6867	8370	577	750
AFS with one (or both) observers	20421	24558	651	824

Table 7.2: Size of the LTSs corresponding to observers and to *AFS*

Note that the size of the system LTS after adding observers is independent of both the number of observers and the static variables they define. The reason is that observers have no effect on the enclosing block behaviour. In our example, property P11 requires static variables to store the information about the acquiring of movement detection information and the duration for which the information is sustained; property P10, contrarily, defines no static variables. ■

7.3.4 Modelling and verifying the AFCS system

We study an *AFCS* system without redundancy, i.e., composed of one *FCP* and one *AFS* channels. We first consider the *AFCS* with the default values of *fcp_mvt_prolong* (13) and *afs_mvt_prolong* (6). Then, we parameterise the system, following [BBJ14], to illustrate the way parameterised models can be automatically generated and verified, using SVL.

Default model

Blocks *FCP*, *AFS_COM*, and *AFS_MON* are composed as highest-level blocks inside a system, named *Main*, as depicted in Figure 7.8. All parameters of the system are made observable. The activation strategy of the blocks is modelled by environment *AFCS_Act*, following the primary implementation of quasi-synchrony (see Section 7.1.1). Once output *interrupt* of block *AFS_MON* takes value *true*, the activation of all blocks is prohibited, which corresponds to the deactivation of the current *COM/MON* channel (switch from the master to the slave channels in a system with redundancy).

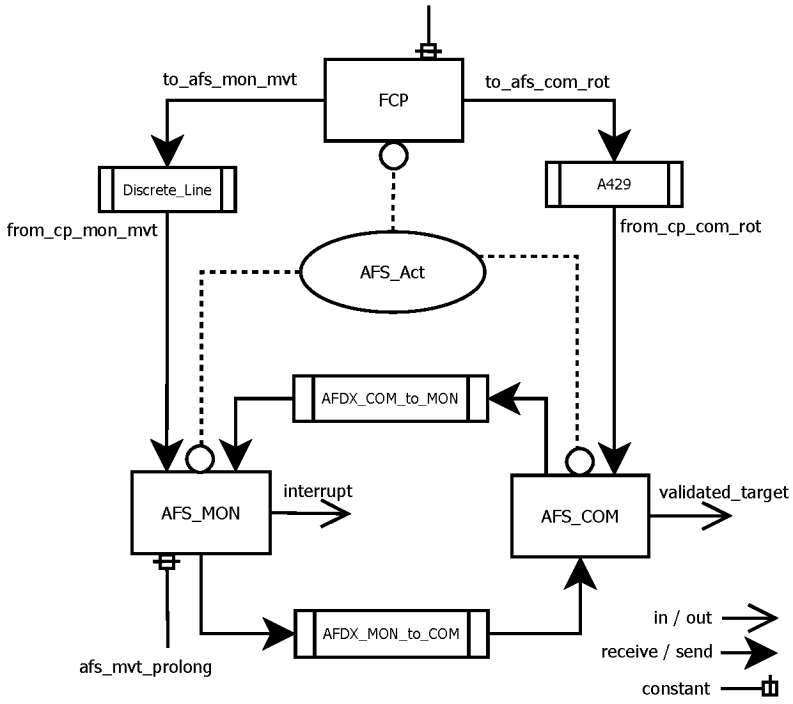


Figure 7.8: Schematic view of the *AFCS* system

Standard LTS generation We generate the system LTS; it contains 44,479,727 states and 60,130,709 transitions. The minimisation modulo divbranching bisimulation yields an LTS with 1,516,951 states, 2,848,395 transitions. The generation and minimisation steps are achieved in 23 minutes on a 64-bit computer.

Property-dependent LTS reduction Since the LTS is relatively large, its reduction before checking properties improves the performance of model checking. Instead of reducing the LTS modulo an equivalence relation or specialising the reduction with regards to each property, we exploit the nature of properties. We first group properties into three classes: system data properties, block data properties, and activation properties. The following table summarises property classification for the *AFCS*.

System data properties	Block data properties		Activation properties
	FCP	*AFS_MON*	
P9, P12, P13	P4, P5, P6, P7, P8	P4, P10, P11	P1, P2, P3

Then, we specialise the reduction for each class. For system data properties, we specialise the reduction with regards to each property. To this aim, we take advantage of the approach proposed in [MW14]. It consists in first synthesising the maximal set of actions that can be hidden in the LTS, without changing the truth value of the property. After applying this *maximal hiding*, the LTS can be reduced modulo an adequate equivalence relation before checking the property. An illustration will be given later when addressing

the parameterised model.

For block data properties, we extract the block LTS from the system LTS by hiding all actions corresponding to other blocks and reducing the resulting LTS. This is more efficient than generating one LTS for each property. For block *FCP*, this can be done as follows:

```
1  "AFCS.Orig.bcg" =
2    divbranching reduction of generation of
3      partial hide all but
4        "GATE_PILOT_ROTATION_PILOT_MVT.*", "GATE_ERROR_INJECTION.*",
5        "GATE_TO_AFS_COM_ROTATION.*", "GATE_TO_AFS_MON_MVT.*"
6    in
7      "AFCS.Orig.bcg"
8    end hide;
```

For activation properties, it suffices to generate the LTS corresponding to the activation strategy. The generation scenario suggested in Section 7.1.1 could be expensive in terms of time and memory, given the size of the system LTS. Alternatively, we generate the LTS corresponding to environment *AFCS_Act*, inside which all activation constraints are encoded. Since this is not possible in GRL (due to semantic restrictions), we use the corresponding LNT process named *Main_AFCS_Act*, as follows:

```
1  "AFCS_Act.bcg" =
2    divbranching reduction of generation of
3      hide Gate_Interrupt in
4        "AFCS_System.lnt":"Main_AFCS_Act"
5      end hide;
```

The gate named *Gate_Interrupt* corresponds to input *interrupt* of *AFCS_Act*. It is hidden to keep only *Start* actions in the resulting LTS (line 3). Finally, a minimisation modulo divbranching bisimulation is achieved (line 2). The LTS is generated and reduced in few seconds and contains 55 states and 173 transitions, whereas the scenario suggested in Section 7.1.1 took around 20 minutes and generates an LTS containing 55 states and 118 transitions. The difference in LTS sizes is explained in Remark 7.5.

Remark 7.5. In the current *AFCS*, input *interrupt* is connected to block *AFS_MON*, which influences the activation strategy of the system. Hence, some states of environment *AFCS_Act* are never explored after component composition inside *AFCS*. Contrarily, in the above-considered scenario, input *interrupt* takes arbitrary values. The obtained LTS is then larger than (but includes) the current activation strategy of the system. ■

Verification It is preferable to check the activation strategy of the system in the beginning of the verification process. An erroneous activation strategy is likely to entail several other errors in message transmission between different components.

In the AFCS system, the formalisation of activation properties P1, ..., P3 is summarised in the table below. Property P1 and P2 hold on the system LTS whereas property P3 does not, meaning that *interrupt* never takes value *true*.

Property	Formalisation in muGRL
P1	All_Alive (*FCP*, *AFS_MON*, *AFS_COM*)
P2	Starvation_Freedom (*FCP*) Starvation_Freedom (*AFS_MON*) Starvation_Freedom (*AFS_COM*)
P3	Deadlock (*FCP*, *AFS_MON*, *AFS_COM*)

In a second step, we checked that block data properties, specified in Sections 7.3.2 and 7.3.3, keep their truth values after component composition inside the system. Finally, we checked that system data properties hold on the system LTS.

Parameterised model

We vary the durations for which blocks *FCP_MON* and *AFS_MON* sustain the movement detection information. The verification task is to check for which durations properties P12 and P13 hold. To this aim, we parametrise LTS generation and verification.

Parameterised generation First, we export parameters *fcp_mvt_prolong* and *afs_mvt_prolong* of *FCP_MON* and *AFS_MON*, respectively, to system level.

```
1  system Main_Param {fcp_mvt_prolong, afs_mvt_prolong: nat}
2                    ( ... )
3  is
4    alias FCP      {fcp_mvt_prolong} as FCP,
5          AFS_MON {afs_mvt_prolong} as AFS_MON,
6    ...
7  end system
```

Then, as suggested in [BBJ14], we vary *fcp_mvt_prolong* and *afs_mvt_prolong* from 1 to 6 periods of *AFS_MON*. Since *fcp_mvt_prolong* is expressed as a multiple of *FCP* period, which is twice as fast as *AFS_MON*, *fcp_mvt_prolong* should take the following values {2, 4, 6, 8, 10, 12}. These steps are achieved using the following SVL script:

```
1  % for i in 2 4 6 8 10 12; do
2   % for j in 1 2 3 4 5 6; do
3    % MODEL="AFCS_CP_${i}_AFS_${j}"
4     % grl.open -root "Main_Param ($i of nat, $j of nat)" AFCS_System.grl
5       generator "$MODEL.bcg"
6   % done
7  % done
```

The script enables the generation of 36 LTSs in around 48 hours. The average size of LTSs (without minimisation) is around 20 million states and 30 million transitions. In-

terestingly, this corresponds to the average in state spaces generated by Tina in [BBJ14] after minimisation.

Parameterised verification Property specification is also parameterised thanks to SVL. The SVL statement below specifies property P12, parameterised by *SPEC*, which is a BCG file name, and by *RESULT*, which is a variable storing the truth value of the property. The LTS in the BCG file will be checked against the formula enclosed between symbols "|=" and ";" and a diagnostic will be given in a BCG file named *diag_SPEC*.

```
property P12 (SPEC, RESULT)
  "A movement detected in FCP is sustained enough to be observed by AFS_MON"
is
  "diag_${SPEC}.bcg" =
   "${SPEC}.bcg" |=
      NEVER ((not {TO_AFS_MON_MVT !TRUE})*.  {TO_AFS_MON_MVT !TRUE}.
             (not {FROM_CP_MON_MVT !TRUE})*. {TO_AFS_MON_MVT !FALSE}.
             (not {TO_AFS_MON_MVT !TRUE})*.  {TO_AFS_MON_MVT !TRUE}
            );
    result "$RESULT"
  expected TRUE
end property
```

Finally, the following SVL script automates the minimisation of all the generated LTSs, after applying a maximal hiding, and checks property P12 on the reduced LTSs. When the property does not hold on an LTS, meaning that variable *RESULT* evaluates to *false*, the produced counter-example is reduced. Otherwise, a witness is provided, in which case it is removed. All these steps are achieved by the following SVL script:

```
% for i in 2 4 6 8 10 12; do
 % for j in 1 2 3 4 5 6; do
  % MODEL="AFCS_CP_${i}_AFS_${j}"
   "P12_${MODEL}.bcg" =
     total branching reduction of
       partial hide all but "TO_AFS_MON_MVT.*", "FROM_CP_MON_MVT.*"
       in "${MODEL}.bcg";
   check P12 ("P12_$MODEL", "RESULT");
   % if [ "$RESULT" = FALSE ]; then
     "diag_P12_${MODEL}.bcg" =
       total branching reduction of "diag_P12_${MODEL}.bcg";
   % else
     % rm "diag_P12_${MODEL}.bcg"
   % fi
 % done
% done
```

Remark 7.6. For property P12, the maximal hiding set is large: only 4 actions out of 85 must remain visible. In such a case, one could perform on-the-fly minimisation at generation phase, i.e., using a forward traversal of the LTS to compute state successors modulo the divbranching bisimulation reduction. ■

Verification results For the values of *fcp_mvt_prolong* and *afs_mvt_prolong* satisfying "$fcp\_mvt\_prolong/2 + afs\_mvt\_prolong < 9$", properties P12 and P13 do not hold on both our model and in [BBJ14]. For all other values, properties hold in [BBJ14]. On our model, contrarily, the properties hold only for values satisfying "$fcp\_mvt\_prolong/2 + afs\_mvt\_prolong > 12$".

7.3.5 Discussion

We confronted GRL, which is time-abstract, with systems involving strict real-time constraints. Hence, we attempted to over-approximate real-time constraints. On GRL synchronous blocks, they were described as multiples of block periods, using static variables; whereas on GRL asynchronous systems, they were described inside environments, using activation signals. For the latter, we experimented a primary implementation of quasi-synchrony to express block activation paces. We concluded that quasi-synchrony is appropriate to model the activation of realistic GALS systems.

The verification task was about timed aspects. As expected, we had less accurate results than in [BBJ14], which use the Tina toolbox. The reason is that we over-approximated real-time constraints in our framework where as such constraints were accurately described in [BBJ14]. Our results are thus still reasonable. Nonetheless, we believe that the refined implementation of quasi-synchrony (see Section 7.1.2) would lead to better verification results, i.e., closer to [BBJ14], since it is more accurate than the primary implementation. Unfortunately, we could not validate our intuition by experimentation, due to time pressure.

Reasoning about timed aspects of systems is known to be more detailed and complex than reasoning about untimed ones. To reduce this complexity, one could develop a system meeting a specification in which constraints are abstracted or over-approximated in the first design phases where errors are frequent. In a second phase, the system can be refined to meet precise real-time constraints.

7.4 Networks of Programmable Logic Controllers

This section reports our experience in early integration of GRL in the design process of PLCs. The experience is in collaboration with Crouzet Automatismes in the context of the industry-led Bluesky project. Crouzet has an internal software named *em4soft* for the design of PLCs. The software builds upon a synchronous dataflow language, with a graphical syntax based on block diagrams, and with no formal semantics. For example, the block diagram depicted in Figure 7.9 is the design sheet of the *exit* PLC in the car park application (See Example 3.7, page 42 for the corresponding GRL code).

The aim of the project is to distribute several PLCs and make them communicate via either wired or wireless network. This paves the way for distributed PLC-applications such as green buildings in which PLCs cooperate together to enhance energy management.

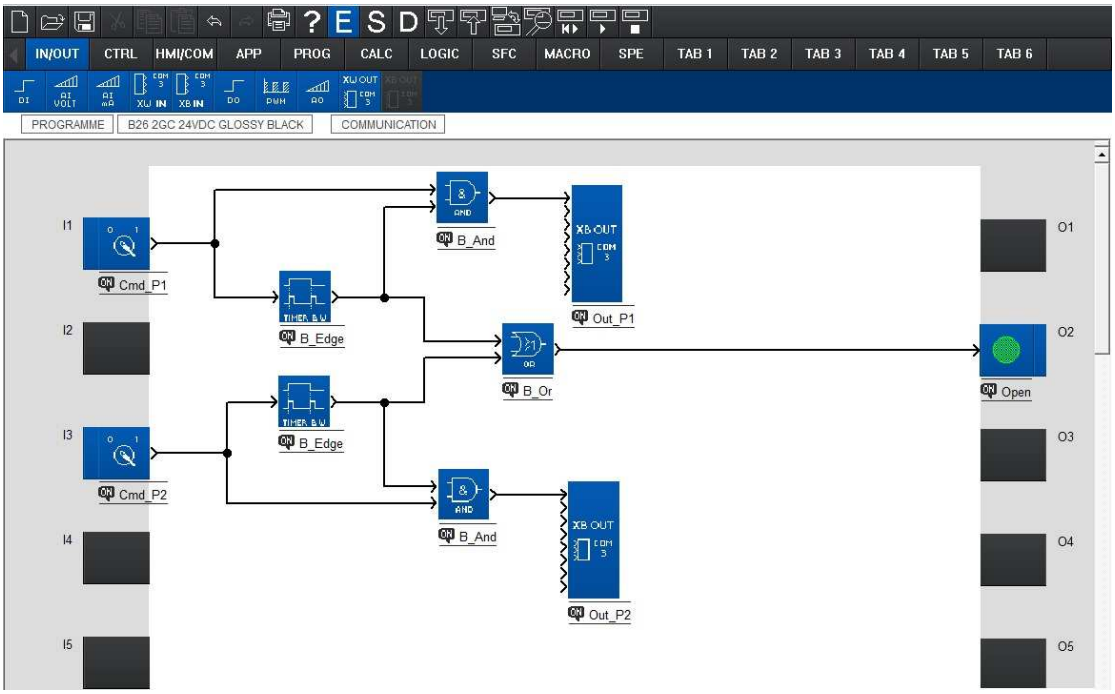


Figure 7.9: Block diagram of the exit PLC (car park application)

Assumptions on the global behaviour of applications include the following. Periods of individual PLCs are presumed with no influence on the expected service of applications. Communication is by sampling and is subject to message loss. Correct message transmission is thus ensured by dedicated communication protocols, e.g., Modbus or Publish-Subscribe protocols.

Modelling aspects of the car park application have been introduced in Chapter 3 and the complete GRL model is available in Appendix B. In the sequel, we report few experiments on LTS generation and verification. Then, we briefly sketch the current status of the Bluesky project.

7.4.1 The car park application

Property specification The verification task concerns the following properties, which are formalised in muGRL in the table below:

(P1) The system composed of blocks *Entrance*, *Storey1*, *Storey2*, and *Exit* may be idle
(P2) Each time a car leaves, the car park availability is updated

Property	Formalisation in muGRL
P1	Idle (*Entrance*, *Storey1*, *Storey2*, *Exit*)
P2	After_Inev (**true*** . {*S_Out*1 = **true**}, **true*** . {*R_Out*1 = **true**})

State space generation The car park capacity is set to 5. We first attempted the generation of LTSs corresponding to highest-level blocks *Entrance*, *Storey1*, *Storey2*, and *Exit*, independently. Table 7.3 reports the size of component LTSs. We have stopped the generation of the LTS corresponding to block *Entrance* after around 11 hours. We then attempted to generate the system LTS by composing the different blocks to communicate with each other through mediums, without putting activation and data constraints. The generation leads to state space explosion as well.

	States	Transitions
Entrance	> 775,922,512	> 1,269,630,000
Exit	37	49
Storey1	55	83
Storey2	55	83

Table 7.3: LTS size of the car park components

In a second step, we fine-tuned the behaviour of the car park by adding the following data constraints:

- a leaving request cannot occur if there is no car in the car park
- an entrance or exit request cannot occur in two successive steps of a PLC
- a ticket given to a car references exactly one storey

The generation of the system LTS succeeds leading to an LTS with 917,184 states and 1,178,349 transitions. As such, not only data constraints provide a realistic view of the system behaviour but they also help to keep tractable the size of LTSs. Additionally, we constrained the activation of the different blocks so that they evolve at the same pace. The resulting LTS contains 2,194,731 states and 2,658,808 transitions, i.e., larger than the LTS involving only data constraints. Table 7.4 illustrates the influence of the car park capacity on the size of the system LTS.

	Non-minimised		Divbranching min.	
Capacity	States	Transitions	States	Transitions
1	395,051	476,920	7,321	10,121
5	2,194,731	2,658,808	30,235	42,463
10	4,444,331	5,386,168	58,655	82,503
15	6,693,931	8,113,528	87,075	122,543
20	8,943,531	10,840,888	115,495	162,583
25	11,193,131	13,568,248	143,915	202,623

Table 7.4: LTS size for different capacities of the car park

Verification results For the car park system involving only data constraints, properties P1 and P2 do not hold on the system LTS. For property P1, idleness was expected: when no car enters or leaves the car park, blocks do not progress and remain idle. Property P2 requires that each message sent by the exit PLC, indicating a car leaving, is received by the entrance PLC. The reason for which the property does not hold is twofold:

- No assumption is made on the relative paces of PLCs. Each PLC can then perform several steps before any other PLC could execute in the meanwhile. This situation can be captured by checking the starvation of *Entrance*, *Storey1*, and *Storey2*, as follows:

  ```
  Starvation_Freedom (Entrance, ⟨ Exit, Storey1, Storey2 ⟩)
  Starvation_Freedom (Storey1, Exit)
  Starvation_Freedom (Storey2, Exit)
  ```

- The message indicating the car leaving can be silently lost or duplicated due to medium unreliability.

An error management mechanism should then be implemented in the system, alerting the user as soon as a transmission failure occurs.

7.4.2 Industrial use of GRL

A connection from *em4soft* to GRL is automated. Basically, *em4soft* generates executable code to be embedded on the PLCs, after performing static analysis, including causality analysis. The *em4soft* compiler has been enhanced to also generate GRL models of blocks. Such connection is quite straightforward, once causality analysis has already been done. Additionally, GRL environments constraining the data of individual PLCs are automatically generated. This enables verification of synchronous components to be performed either by visual checking or by writing down temporal logic properties (mainly safety patterns).

Still, GRL mediums together with activation constraints should be encoded by hand by engineers, at the time being. The reason is that the software does not yet support multi-sheets enabling GALS design. As a future direction, the aim is to develop a catalogue of generic GRL environments and mediums, that can be automatically generated from *em4soft*. Crouzet investigates to invest in using GRL as a textual language with formal semantics for *em4soft*.

Chapter 8

Conclusion

GALS systems are composed of several synchronous components, interacting asynchronously. While asynchronous concurrency is a central topic in GALS modelling, the practical impact of the underlying verification techniques is still limited in many design processes. A main reason is the inherent complexity of asynchronous concurrency and dedicated techniques, resulting in a steep learning curve.

Summary of contributions

This thesis proposes a fresh look at formal modelling and verifying GALS systems, taking asynchronous concurrency as major subject. The main intent behind the proposed approach was twofold: (i) transferring verification techniques for asynchronous systems to GALS design (ii) studying the adequacy of these techniques with GALS behaviour. Our solution consists in devising specific languages matching the knowledge and intuition of GALS designers. To this aim, we defined a behavioural description language, GRL, and a property specification language, muGRL.

We originally designed GRL as an intermediate format that connects GALS design languages to formal verification tools. Synchronous blocks can be defined using a minimal set of core constructs to which synchronous languages can be translated. Asynchronous environments and mediums can be defined using built-in constructs to model arbitrarily-complex GALS systems. The degree of asynchronous concurrency is by default maximal, enabling all possible behaviours to be modelled. It can be adjusted using environments to meet bounded nondeterministic and deterministic GALS applications or even yet sequential scenarios. Communication mediums with complex buffering mechanisms can be described. This enables to address the frequently occurring communication schemes in GALS systems, namely, communication by sampling and FIFO queues, which may be reliable or not.

We formally defined the semantics of GRL. Formal semantics improved our understanding of all the language subtleties and helped us in tool construction.

We defined a translation from GRL into LNT. An encoding of GALS behaviour in asynchronous process languages is proposed. In particular, the atomicity of blocks is preserved by a locking mechanism. Not only the locking mechanism reduces the resulting state spaces, but also is particularly useful to count block steps for verification purpose.

We designed muGRL to assist non-expert users in specifying system requirements. It builds on a collection of high-level and parameterisable patterns that capture recurring properties for GALS systems. Properties result from a pragmatic survey on the state-of-the-art verification of synchronous and asynchronous systems. muGRL is implemented by a translation (not completely automated yet) into MCL.

We confronted GRL and muGRL with several case studies to explore their capability in addressing real-life systems. Case studies are issued from both academia and industry. An implementation of quasi-synchrony in an asynchronous model of time is proposed and its functional correctness is verified. In a collaboration with IRT-Saint Exupéry (Toulouse, France), the modelling and verification of an AutoFlight Control System with stringent timing requirements are studied. The quasi-synchrony implementation has been used for this system, leading to reasonable experimentation results. In the framework of the Bluesky project, networks of PLCs with a high degree of asynchronous concurrency are addressed.

We concluded that verification tools for asynchronous systems, including CADP, are adequate to address arbitrarily-complex GALS systems. The accuracy of verification results relies on suitable behavioural abstractions of the intended system.

Industrial feedback

Despite its young age, GRL appears to have a good acceptance by GALS designers.

From the outset of the Bluesky project, Crouzet provided us with significant amount of insights and feedback about both design choices and user-convenience of GRL and muGRL. An early integration of GRL in Crouzet's design process of PLCs has started and is under experimentation. At the time being, only synchronous blocks and generic environments can be automatically generated. Unfortunately, we could not explore the asynchronous aspects of GRL and muGRL because the current design process does not yet support GALS systems. Crouzet is still very optimistic about the usefulness and user-convenience of GRL and muGRL. A new PhD thesis is going to start as a continuation of this work.

Beyond the *Bluesky* project, our collaboration with IRT-Saint Exupéry was fruitful. Quoting them: "*We are extremely interested by this work, because we believe it deals with an essential problem*". GRL is currently evaluated in IRT-Saint Exupéry.

These experiences reinforce our conviction about the pressing need to deal with asynchronous concurrency in the construction of GALS systems. It also shows that domain-

specific languages are an appropriate solution for disseminating asynchronous concurrency techniques to industry.

Directions for future work

The focus of this thesis was on designing GALS-specific languages and tools. While we achieved most of our objectives, we foresee several directions for future work.

From a language-design point of view, GRL could be extended with the following aspects:

Generic libraries. For the frequently used synchronous programming operators, activation strategies, and communication mediums, one could write libraries in a generic and reusable fashion. These libraries, once functionally verified, could be reused safely.

Equivalence checking. A GALS system can be checked against a more abstract behaviour of the expected service. Work[1] on this direction has started in the *Bluesky* project, but faced a lack of expressiveness in GRL. Service description requires to abstract from the system composition into components, which is not possible in the actual version of GRL. In this respect, GRL systems can be extended to ease the formalisation of services.

From a language-implementation point of view, proving formally the correctness of the translation from GRL to LNT is an ambitious task. GRL is still a young language in experimentation phase; it may undergo several changes in the future. We prefer to postpone the translation proof to more stable versions of GRL. In a nearer future, we foresee to fully automate the translation from muGRL into MCL and map verification diagnostics back to GRL.

The connection of GRL to CADP provides the user with various verification tools and techniques. The following would be useful for GALS systems:

Compositional verification. Asynchronous concurrency may lead to state-space explosion. In this respect, compositional verification techniques achieve promising reductions. These techniques are still inherently complex and rely on the target system architecture. *Automatic generation of interfaces* [KM97, Lan06] could be particularly interesting for GRL systems. This technique generates a component LTS by considering the behavioural restrictions imposed by the component neighborhood. Hence, the states that are never explored in the LTS of the whole system are eliminated in the component LTS. For more efficiency, automatic generation of interfaces could be combined with property-dependent reductions (Chapter 7).

Probabilistic verification. Real-life GALS systems are subject to unreliable and unpredictable phenomena, such as message loss and component failure. We illustrated

[1] This work is available under a project deliverable, which is not diffused publicly. The interested reader may write to Radu.Mateescu@inria.fr for discussion or documentation.

the way such stochastic phenomena could be modelled in GRL using nondeterminism. Once the GRL model is functionally verified, its LTS could be enriched by attaching probabilities to transitions. This way, transitions could be chosen probabilistically instead of nondeterministically. Probabilistic verification could be achieved on such LTSs. For example, one could estimate what is the failure rate of redundant components in fault-tolerant systems.

Appendix A

The GRL Model and SVL Verification Scripts of the AFCS

This appendix presents the GRL model and SVL verification script of the *AutoFlight Control Systems*. Since muGRL is not fully implemented, properties are written in MCL.

A.1 The GRL model

A.1.1 Global constants

```
1  ------------------------------------------------------------------------
2  -- Global constants
3  const default_fcp_mvt_prolong: nat  := 13
4  const default_afs_mvt_prolong: nat  := 6
5  const disable_error_injection: bool := false
6  ------------------------------------------------------------------------
```

A.1.2 Component FCP

```
1  ------------------------------------------------------------------------
2  -- System to generate the LTS corresponding to the FCP component
3  system Main_FCP {fcp_mvt_prolong : nat}
4                  (error_injection : bool,
5                   pilot_rotation  : int,
6                   pilot_mvt       : bool,
7                   afs_com_rotation: int,
8                   afs_mon_mvt     : bool)
9  is
10   alias FCP {fcp_mvt_prolong} as FCP,
11         Error_Enable as Error_Enable,
12         Knob as Knob
13   block list
14     FCP (<pilot_rotation, pilot_mvt>, error_injection)
15         [?afs_com_rotation, ?afs_mon_mvt]
```

```
16    environment list
17      Error_Enable (?error_injection),
18      Knob (?<pilot_rotation, pilot_mvt>)
19  end system
20  ------------------------------------------------------------------------
21  -- Block modelling the FCP component
22  block FCP {-- duration to sustain movement detection
23             fcp_mvt_prolong: nat := default_fcp_mvt_prolong}
24            (in    knob_click_number       : int,
25                   knob_mvt                : bool,
26             in    error_injection         : bool)
27            [send  afs_com_target_position: int,
28             send  afs_mon_mvt             : bool]
29  is
30    var cp_click_number:int,
31        detected_mvt  :bool
32    Encoder (error_injection, knob_click_number, knob_mvt,
33             ?cp_click_number, ?detected_mvt);
34    CP_COM  (cp_click_number, ?afs_com_target_position);
35    CP_MON {fcp_mvt_prolong} (detected_mvt, ?afs_mon_mvt)
36  end block
37  ------------------------------------------------------------------------
38  -- Block modelling the encoder behaviour
39  block Encoder (in  error_injection  : bool,
40                 in  knob_click_number: int,
41                     pilot_mvt        : bool,
42                 out cp_click_number  : int,
43                     detected_mvt     : bool)
44  is
45    static var permanent_failure:bool := false
46    cp_click_number := knob_click_number;
47    if not (permanent_failure) then
48      permanent_failure := error_injection;
49      detected_mvt := pilot_mvt
50    else
51      detected_mvt := false
52    end if
53  end block
54  ------------------------------------------------------------------------
55  -- Block modelling the command channel of the FCP component
56  block CP_COM (in  click_number       : int,
57                out target_position_afs: int)
58  is
59    static var pre_click_number:int := 0
60    -- compute a new position, to be sent to AFS
61    target_position_afs := pre_click_number + click_number;
62    pre_click_number    := click_number -- store the computed position
63  end block
64  ------------------------------------------------------------------------
65  -- Block modelling the monitoring channel of the FCP component
66  block CP_MON {prolong_duration          : nat := default_fcp_mvt_prolong}
67               (in    cp_mon_detected_mvt: bool,
```

```
68                out   to_afs_detected_mvt: bool)
69  is
70    static var countdown  : nat  := prolong_duration,
71               isdetected : bool := false
72    if (cp_mon_detected_mvt) then
73      -- a movement is detected
74      isdetected          := true; -- store detection information
75      to_afs_detected_mvt := true;  -- inform AFS
76      countdown           := prolong_duration - 1 -- trigger countdown
77    elsif (isdetected and (0 < countdown)) then
78      -- sustain signal detection for duration "prolong_duration"
79      to_afs_detected_mvt := true;
80      countdown           := countdown - 1
81    else  -- duration "prolong_duration" has elapsed
82      to_afs_detected_mvt := false;
83      isdetected := false
84    end if
85  end block
86  ------------------------------------------------------------------------
87  -- Environment modelling the knob behaviour
88  environment Knob (out click_number: int,
89                          mvt         : bool)
90  is
91    static var pre_click_number:int := 0
92    when <click_number, mvt> ->
93    select
94      -- The knob position changes. We consider values from -2 to 2
95      click_number := any int where ((click_number < 3) and (click_number > -3));
96      pre_click_number := click_number; -- store position
97      mvt := true
98    [] -- The knob position does not change
99      click_number := pre_click_number;
100     mvt := false
101   end select
102 end environment
103 ------------------------------------------------------------------------
104 -- Environment disabling the error injection input of the FCP component
105 environment Error_Enable (out error_injection: bool) is
106   when error_injection ->
107     error_injection := disable_error_injection
108 end environment
109 ------------------------------------------------------------------------
```

A.1.3 Component AFS

```
1   ------------------------------------------------------------------------
2   -- System to generate the LTS corresponding to the AFS components
3   system Main_AFS {afs_mvt_prolong                    : nat}
4                   (validated_target                   : int,
5                    interrupt                          : bool,
6                    cp_com_target_position             : int,
7                    cp_mon_mvt                         : bool,
```

```
8                   com_to_mon_afs_target_validation   : int,
9                   mon_from_com_afs_target_validation: int,
10                  mon_to_com_afs_target_validated    : int,
11                  mon_to_com_afs_target_isvalid      : bool,
12                  com_from_mon_afs_target_validated  : int,
13                  com_from_mon_afs_target_isvalid    : bool)
14  is
15    alias AFS_COM                    as AFS_COM,
16          AFS_MON {afs_mvt_prolong}  as AFS_MON,
17          AFS_Act {_, _}             as Activation,
18          Stub                       as Stub,
19          AFDX_COM_to_MON            as AFDX_COM_to_MON,
20          AFDX_MON_to_COM            as AFDX_MON_to_COM
21    block list
22      AFS_COM (?validated_target)
23              [cp_com_target_position,
24               <com_from_mon_afs_target_validated,
25                com_from_mon_afs_target_isvalid>,
26               ?com_to_mon_afs_target_validation],
27      AFS_MON (?interrupt)
28              [cp_mon_mvt, mon_from_com_afs_target_validation,
29              ?<mon_to_com_afs_target_validated, mon_to_com_afs_target_isvalid>]
30    environment list
31      Activation (AFS_COM, AFS_MON)
32    medium list
33      Stub [?cp_com_target_position],
34      AFDX_COM_to_MON [com_to_mon_afs_target_validation,
35                      ?mon_from_com_afs_target_validation],
36      AFDX_MON_to_COM [<mon_to_com_afs_target_validated,
37                       mon_to_com_afs_target_isvalid>,
38                      ?<com_from_mon_afs_target_validated,
39                        com_from_mon_afs_target_isvalid>]
40  end system
41  ------------------------------------------------------------------------------
42  -- Block modelling the command channel of the AFS component
43  block AFS_COM (out      validated_target: int)
44     [receive cp_com_target_position: int,  -- receive knob position from CP_COM
45      receive mon_alt_target_value  : int,  -- receive target value from AFS_MON
46              mon_alt_target_valid  : bool, -- along with its validilty verdict
47      send    afs_mon_alt_target    : int]
48  is
49    static var last_validated_target:int := 0
50    -- establish an altitude target based on the received position from CP COM
51    -- and send it to AFS MON for validation
52    Compute_Alt (cp_com_target_position, ?afs_mon_alt_target);
53    -- a decision is taken based on the valididty verdict
54    if (mon_alt_target_valid) then
55      validated_target      := mon_alt_target_value;
56      last_validated_target := validated_target
57    else
58      validated_target := last_validated_target
59    end if
```

```
60  end block
61  ------------------------------------------------------------------------
62  block Compute_Alt (in  cp_com_target_position:int,
63                     out afs_mon_alt_target    :int) is
64     afs_mon_alt_target := cp_com_target_position
65  end block
66  ------------------------------------------------------------------------
67  -- Block modelling the monitoring channel of the AFS component
68  block AFS_MON {afs_mvt_prolong: nat := default_afs_mvt_prolong}
69     (out     interrupt          : bool) -- in case of problem, raise an alert
70     [receive cp_mon_mvt         : bool,
71      receive afs_com_alt_target: int,
72      send    afs_com_alt_target_value: int,
73              afs_com_alt_target_valid: bool]
74  is
75    static var pre_alt_target:int := 0, -- last altitude target from AFSCOM
76               pre_valid_alt_target:int := 0, -- last valid altitude target
77               countdown   :nat  := afs_mvt_prolong,
78               isdetected  :bool := false
79     var detected_mvt: bool
80    if (cp_mon_mvt) then  -- a movement is detected
81      detected_mvt := true;
82      isdetected   := true; -- store detection information
83      countdown    := afs_mvt_prolong -- trigger countdown
84    elsif ((countdown > 0) and isdetected) then
85      -- sustain signal detection for duration "afs_mvt_prolong"
86      detected_mvt := true;
87      countdown := countdown - 1
88    else
89      detected_mvt := false;
90      isdetected   := false
91    end if;
92    if ((afs_com_alt_target == pre_alt_target)  -- valid
93      or ((afs_com_alt_target != pre_alt_target) and detected_mvt))
94    then
95      interrupt := false;
96      afs_com_alt_target_value := afs_com_alt_target;
97      afs_com_alt_target_valid := true;
98      pre_valid_alt_target := afs_com_alt_target
99    else  --invalid
100     interrupt := true;
101     afs_com_alt_target_value := pre_valid_alt_target;
102     afs_com_alt_target_valid := false
103   end if;
104   pre_alt_target :=  afs_com_alt_target
105  end block
106 ------------------------------------------------------------------------
107 -- Block modelling the monitoring channel of the AFS component and defining
108 -- observers
109 block AFS_MON_Observ_P13_P14 {afs_mvt_prolong: nat := default_afs_mvt_prolong}
110    (out     interrupt              : bool,
111     out     OK_P13, OK_P14         : bool)
```

```
112     [receive cp_mon_mvt              : bool,
113      receive afs_com_alt_target      : int,
114      send    afs_com_alt_target_value: int,
115              afs_com_alt_target_valid: bool]
116  is
117    static var pre_alt_target      : int := 0,
118               pre_valid_alt_target: int := 0,
119               countdown           : nat  := afs_mvt_prolong,
120               isdetected          : bool := false
121    var detected_mvt: bool
122    if (cp_mon_mvt) then
123      detected_mvt := true;
124      isdetected   := true;
125      countdown    := afs_mvt_prolong
126    elsif ((countdown > 0) and isdetected) then
127      detected_mvt := true;
128      countdown := countdown - 1
129    else
130      detected_mvt := false;
131      isdetected   := false
132    end if;
133    if ((afs_com_alt_target == pre_alt_target)  --valid
134      or ((afs_com_alt_target != pre_alt_target) and detected_mvt))
135    then
136      interrupt := false;
137      afs_com_alt_target_value := afs_com_alt_target;
138      afs_com_alt_target_valid := true;
139      pre_valid_alt_target := afs_com_alt_target
140    else  --invalid
141      interrupt := true;
142      afs_com_alt_target_value := pre_valid_alt_target;
143      afs_com_alt_target_valid := false
144    end if;
145    pre_alt_target :=  afs_com_alt_target;
146    -- observers
147    Observer_P13 {default_afs_mvt_prolong}(cp_mon_mvt, detected_mvt, ?OK_P13);
148    Observer_P14 {default_afs_mvt_prolong}(cp_mon_mvt, countdown, ?OK_P14)
149  end block
150  ----------------------------------------------------------------------
151  -- Block observer checking that the countdown to sustain a movement detection
152  -- information is always set to a predefined value when a movement is detected
153  block Observer_P13 {duration: nat := default_afs_mvt_prolong}
154                     (in  cp_mon_mvt: bool, mvt_prolong:nat,
155                      out ok        : bool)
156  is
157    if (cp_mon_mvt) then
158      ok := (mvt_prolong == duration)
159    else
160      ok := true
161    end if
162  end block
163  ----------------------------------------------------------------------
```

```
164  -- Block observer checking that a movement detection information is sustained
165  -- for a fixed duration, if no new movement detection has occurred ever since
166  block Observer_P14 {duration: nat := default_afs_mvt_prolong}
167                     (in cp_mon_mvt, detected_mvt: bool,
168                      out ok: bool)
169  is
170    static var counter: nat := 0, pre_cp_mon_mvt: bool := false,
171               trigger_count :bool := false
172     if (not (pre_cp_mon_mvt) and cp_mon_mvt) then  -- a movement is detected
173       trigger_count := true
174     end if;
175     if (trigger_count and (counter < duration)) then
176       ok := (detected_mvt == true);
177       counter := counter + 1
178     else
179       ok := true;
180       trigger_count := false
181    end if;
182    pre_cp_mon_mvt := cp_mon_mvt
183  end block
184  ------------------------------------------------------------------------
185  -- Environment ensuring that two blocks evolve at multiples of the same pace
186  environment AFS_Act {max_com: nat := 1, max_mon: nat := 3}
187                      (block AFS_COM, AFS_MON)
188  is
189     static var count_mon, count_com: nat := 0
190     select
191        if (count_com < max_com) then
192       count_com := count_com + 1;
193       enable AFS_COM
194        end if
195     []
196        if (count_mon < max_mon) then
197       count_mon := count_mon + 1;
198       enable AFS_MON
199        end if
200     end select;
201     if (count_com >= max_com) and (count_mon >= max_mon) then
202        count_com := 0;
203        count_mon := 0
204     end if
205  end environment
206  ------------------------------------------------------------------------
207  -- Medium modelling an AFDX
208  medium AFDX [receive com_to_mon_afs_target_validation   : int,
209               send    mon_from_com_afs_target_validation: int,
210               receive mon_to_com_afs_target_validated   : int,
211                       mon_to_com_afs_target_isvalid     : bool,
212               send    com_from_mon_afs_target_validated : int,
213                       com_from_mon_afs_target_isvalid   : bool]
214  is
215    static var afs_target_validation: int  := 0,
```

```
216             validated_target          : int  := 0,
217             isvalid_target            : bool := false
218       select
219         select
220           when ?<com_to_mon_afs_target_validation> ->
221             afs_target_validation := com_to_mon_afs_target_validation
222         []
223           when <mon_from_com_afs_target_validation> ->
224             mon_from_com_afs_target_validation := afs_target_validation
225         end select
226         []
227         select
228           when ?<mon_to_com_afs_target_validated, mon_to_com_afs_target_isvalid> ->
229             validated_target := mon_to_com_afs_target_validated;
230             isvalid_target   := mon_to_com_afs_target_isvalid
231         []
232           when <com_from_mon_afs_target_validated, com_from_mon_afs_target_isvalid>
233             -> com_from_mon_afs_target_validated := validated_target;
234                com_from_mon_afs_target_isvalid   := isvalid_target
235         end select
236       end select
237     end medium
238     ------------------------------------------------------------------------
239     -- Medium modelling communication from the monitoring to the command channels
240     medium AFDX_MON_to_COM [receive mon_to_com_afs_target_validated   : int,
241                                     mon_to_com_afs_target_isvalid     : bool,
242                             send    com_from_mon_afs_target_validated : int,
243                                     com_from_mon_afs_target_isvalid   : bool]
244     is
245       static var validated_target: int  := 0,
246                  isvalid_target  : bool := false
247       select
248         when ?<mon_to_com_afs_target_validated, mon_to_com_afs_target_isvalid> ->
249           validated_target := mon_to_com_afs_target_validated;
250           isvalid_target   := mon_to_com_afs_target_isvalid
251       []
252         when <com_from_mon_afs_target_validated, com_from_mon_afs_target_isvalid>
253           -> com_from_mon_afs_target_validated := validated_target;
254              com_from_mon_afs_target_isvalid   := isvalid_target
255       end select
256     end medium
257     ------------------------------------------------------------------------
258     -- Medium modelling communication from the command to the monitoring channels
259     medium AFDX_COM_to_MON [receive com_to_mon_afs_target_validation  : int,
260                             send    mon_from_com_afs_target_validation: int]
261     is
262       static var afs_target_validation: int  := 0
263       select
264         when ?com_to_mon_afs_target_validation ->
265           afs_target_validation := com_to_mon_afs_target_validation
266       []
267         when mon_from_com_afs_target_validation ->
```

```
268         mon_from_com_afs_target_validation := afs_target_validation
269       end select
270     end medium
271     ------------------------------------------------------------------------
272     -- Medium simualating the communication between the command channel of the
273     -- AFS component and the FCP component
274     medium Stub [send cp_com_target_position: int] is
275       when cp_com_target_position ->
276         cp_com_target_position := any int
277                                   where ((cp_com_target_position < 3)
278                                   and (cp_com_target_position > -3))
279     end medium
280     ------------------------------------------------------------------------
```

A.1.4 System AFCS

```
1     ------------------------------------------------------------------------
2     -- System to generate the LTS corresponding to the AFCS system
3     system Main (error_injection                    : bool,
4                  pilot_rotation                     : int,
5                  pilot_mvt                          : bool,
6                  to_afs_com_rotation                : int,
7                  to_afs_mon_mvt                     : bool,
8                  validated_target                   : int,
9                  interrupt                          : bool,
10                 cp_com_target_position             : int,
11                 from_cp_mon_mvt                    : bool,
12                 com_to_mon_afs_target_validation   : int,
13                 mon_from_com_afs_target_validation: int,
14                 mon_to_com_afs_target_validated    : int,
15                 mon_to_com_afs_target_isvalid      : bool,
16                 com_from_mon_afs_target_validated  : int,
17                 com_from_mon_afs_target_isvalid    : bool)
18    is
19      alias FCP      {_}       as FCP,
20            AFS_MON  {_}       as AFS_MON,
21            AFS_COM            as AFS_COM,
22            A429_COM           as A429_COM,
23            Discrete_MON       as Discrete_MON,
24            AFDX               as AFDX,
25            Error_Enable       as Error_Enable,
26            Knob               as Knob,
27            AFCS_Act {_, _, _} as Activation
28      block list
29        FCP      (<pilot_rotation, pilot_mvt>, error_injection)
30                 [?to_afs_com_rotation, ?to_afs_mon_mvt],
31        AFS_COM (?validated_target)
32                 [cp_com_target_position,
33                  <com_from_mon_afs_target_validated,
34                    com_from_mon_afs_target_isvalid>,
35                  ?com_to_mon_afs_target_validation],
36        AFS_MON (?interrupt)
```

```
37              [from_cp_mon_mvt,
38               mon_from_com_afs_target_validation,
39               ?<mon_to_com_afs_target_validated,
40                 mon_to_com_afs_target_isvalid>]
41    environment list
42      Error_Enable (?error_injection),
43      Knob         (?<pilot_rotation, pilot_mvt>),
44      Activation   (interrupt, FCP, AFS_COM, AFS_MON)
45    medium list
46      AFDX         [<com_to_mon_afs_target_validation>,
47                    ?<mon_from_com_afs_target_validation>,
48                   <mon_to_com_afs_target_validated,
49                    mon_to_com_afs_target_isvalid>,
50                    ?<com_from_mon_afs_target_validated,
51                      com_from_mon_afs_target_isvalid>],
52      A429_COM     [to_afs_com_rotation, ?cp_com_target_position],
53      Discrete_MON [to_afs_mon_mvt, ?from_cp_mon_mvt]
54  end system
55  ------------------------------------------------------------------------
56  -- Parameterised system to generate the LTS corresponding to the AFCS system
57  system Main_Param {fcp_mvt_prolong, afs_mvt_prolong: nat}
58              (error_injection                    : bool,
59               pilot_rotation                     : int,
60               pilot_mvt                          : bool,
61               to_afs_com_rotation                : int,
62               to_afs_mon_mvt                     : bool,
63               validated_target                   : int,
64               interrupt                          : bool,
65               cp_com_target_position             : int,
66               from_cp_mon_mvt                    : bool,
67               com_to_mon_afs_target_validation   : int,
68               mon_from_com_afs_target_validation: int,
69               mon_to_com_afs_target_validated    : int,
70               mon_to_com_afs_target_isvalid      : bool,
71               com_from_mon_afs_target_validated  : int,
72               com_from_mon_afs_target_isvalid    : bool)
73  is
74    alias FCP {fcp_mvt_prolong}      as FCP,
75          AFS_MON {afs_mvt_prolong} as AFS_MON,
76          AFS_COM                   as AFS_COM,
77          A429_COM                  as A429_COM,
78          Discrete_MON              as Discrete_MON,
79          AFDX                      as AFDX,
80          Error_Enable              as Error_Enable,
81          Knob                      as Knob,
82          AFCS_Act {_, _, _}         as Activation
83    block list
84      FCP      (<pilot_rotation, pilot_mvt>, error_injection)
85               [?to_afs_com_rotation, ?to_afs_mon_mvt],
86      AFS_COM (?validated_target)
87               [cp_com_target_position,
88                <com_from_mon_afs_target_validated,
```

```
89                  com_from_mon_afs_target_isvalid>,
90                 ?com_to_mon_afs_target_validation],
91        AFS_MON (?interrupt)
92                [from_cp_mon_mvt,
93                 mon_from_com_afs_target_validation,
94                 ?<mon_to_com_afs_target_validated,
95                  mon_to_com_afs_target_isvalid>]
96      environment list
97        Error_Enable (?error_injection),
98        Knob         (?<pilot_rotation, pilot_mvt>),
99        Activation   (interrupt, FCP, AFS_COM, AFS_MON)
100     medium list
101       AFDX         [<com_to_mon_afs_target_validation>,
102                     ?<mon_from_com_afs_target_validation>,
103                     <mon_to_com_afs_target_validated,
104                      mon_to_com_afs_target_isvalid>,
105                     ?<com_from_mon_afs_target_validated,
106                       com_from_mon_afs_target_isvalid>],
107       A429_COM     [to_afs_com_rotation>, ?cp_com_target_position],
108       Discrete_MON [to_afs_mon_mvt, ?from_cp_mon_mvt]
109   end system
110   ------------------------------------------------------------------------
111   -- Medium modelling A429 communication
112   medium A429_COM [receive from_cp_com: int,
113                    send    to_afs_com : int]
114   is
115     static var from_cp_com_to_afs_com:int := 0
116     select
117       when ?from_cp_com ->
118         from_cp_com_to_afs_com := from_cp_com
119     []
120       when to_afs_com ->
121         to_afs_com := from_cp_com_to_afs_com
122     end select
123   end medium
124   ------------------------------------------------------------------------
125   -- Medium modelling discrete communication
126   medium Discrete_MON [receive from_cp_mon: bool ,
127                        send    to_afs_mon : bool]
128   is
129     static var from_cp_mon_to_afs_mon:bool := false
130     select
131       when ?from_cp_mon -> from_cp_mon_to_afs_mon := from_cp_mon
132     []
133       when to_afs_mon   -> to_afs_mon := from_cp_mon_to_afs_mon
134      end select
135   end medium
136   ------------------------------------------------------------------------
137   -- Environment ensuring that:
138   --- three blocks evolve at multiples of the same pace
139   --- blocks are halted if a failure occurs
140   environment AFCS_Act {max_fcp     : nat := 10,
```

```
141                          max_afs_com: nat := 1,
142                          max_afs_mon: nat := 3}
143                         (in     interrupt:bool,
144                          block FCP, AFS_COM, AFS_MON)
145   is
146     static var count_fcp, count_afs_com, count_afs_mon: nat  := 0
147     static var failure                                  : bool := false
148     select
149       -- FCP
150       if ((count_fcp < max_fcp) and not (failure)) then
151         enable FCP;
152         count_fcp := count_fcp + 1
153       end if
154     []
155       -- AFS_COM
156       if ((count_afs_com < max_afs_com) and not (failure)) then
157         enable AFS_COM;
158         count_afs_com := count_afs_com + 1
159       end if
160     []
161       -- AFS_MON
162       if ((count_afs_mon < max_afs_mon) and not (failure)) then
163         enable AFS_MON;
164         count_afs_mon := count_afs_mon + 1
165       end if
166     []
167       when ?interrupt -> failure := interrupt
168     end select;
169     -- reinitialise
170     if ((count_fcp     >= max_fcp) and
171         (count_afs_com >= max_afs_com) and
172         (count_afs_mon >= max_afs_mon))
173     then
174       count_fcp     := 0;
175       count_afs_com := 0;
176       count_afs_mon := 0
177     end if
178   end environment
179   ------------------------------------------------------------------------------
```

A.2 The SVL verification script

A.2.1 Generation and verification script

```
1   ------------------------------------------------------------------------------
2   -- User parameters
3   % MODEL=$1
4   % GENERATE_MODELS=1
5   % FCP_MVT_PROLONG=13
6   % AFS_MVT_PROLONG=6
7   % GENERATE_PARAMETRISED_AFCS=0
8   ------------------------------------------------------------------------------
```

```
9   -- CADP parameters
10  % DEFAULT_VERIFY_TOOL="evaluator4"
11  % CADP_TIME="memtime"
12  % DEFAULT_MCL_LIBRARIES="library.mcl"
13  ------------------------------------------------------------------------
14  -- Script parameter verification
15  % if test "$MODEL" != "FCP_Component" && test "$MODEL" != "AFS_Component"
16      && test "$MODEL" != "AFCS_System"
17  % then
18  %   echo "\nThis SVL script requires the name of a GRL model as argument"
19  %   echo "The name should belong to the set:"
20  %   echo "FCP_Component, AFS_Component, AFCS_System\n"
21  %   exit
22  % fi
23  ------------------------------------------------------------------------
24  -- Parameter setting to execute GRL models
25  % case "$MODEL" in
26  %   "FCP_Component" ) \
27  %     eval "PARAM=${FCP_MVT_PROLONG}"
28  %     echo "\nVerification of the FCP component"
29  %   ;;
30  %   "AFS_Component" ) \
31  %     eval "PARAM=${AFS_MVT_PROLONG}"
32  %     echo "\nVerification of the AFS component"
33  %   ;;
34  %   "AFCS_System" ) \
35  %     eval "PARAM1=${FCP_MVT_PROLONG}"
36  %     eval "PARAM2=${AFS_MVT_PROLONG}"
37  %     echo "\nVerification of the AFCS system"
38  %   ;;
39  % esac
40  ------------------------------------------------------------------------
41  -- Generation of LTSs
42  ------------------------------------------------------------------------
43  % if [ "${GENERATE_MODELS}" = "1" ]
44  % then
45  ------------------------------------------------------------------------
46    -- Original LTS generation
47  % echo "\nLTS genenration: LTS will be given in file ${MODEL}.Orig.bcg"
48  % case "$MODEL" in
49  %   "FCP_Component" | "AFS_Component") \
50  %   grl.open -showall -root "Main_${MODEL} ($PARAM of nat)" ${MODEL}.grl
51        generator ${MODEL}.Orig.bcg
52  %   ;;
53  %   "AFCS_System" ) \
54  %   if [ "${GENERATE_PARAMETRISED_AFCS}" = "0" ]
55  %   then
56  %     grl.open -showall -root "Main_Param ($PARAM1 of nat, $PARAM2 of nat)"
57          ${MODEL}.grl generator ${MODEL}.Orig.bcg
58  %   else
59  %     for i in 3 7 10 13 17 20
60  %     do
```

```
61  %       for j in 1 2 3 4 5 6
62  %       do
63  %         MODEL="AFCS_CP_${i}_AFS_${j}"
64  %         grl.open -root "Main_Param ($i of nat, $j of nat)" AFCS_System.grl
65                generator "$MODEL.bcg"
66              "$MODEL.bcg" = total rename "GATE_\(.*\)" -> "\1" in
67                               total divbranching reduction of "$MODEL.bcg";
68  %       done
69  %     done
70  %   fi
71  %   ;;
72  % esac
73  ------------------------------------------------------------------------
74    -- Data and activation LTS extraction
75  "${MODEL}.Data.bcg" = total divbranching reduction of
76                          total rename
77                            "GATE_\(.*\)"       -> "\1",
78                            "START !GRL_\(.*\)" -> "\1"
79                          in
80                            "${MODEL}.Orig.bcg";
81  "${MODEL}.Act.bcg" = total divbranching reduction of
82                         total rename "START !GRL_\(.*\)" -> "\1" in
83                           partial hide all but ".*START.*"  in
84                             "${MODEL}.Orig.bcg";
85  % echo "\nThe reduced LTS is given in file ${MODEL}.Data.bcg"
86  % echo "The activation strategy is given in file ${MODEL}.Act.bcg\n"
87  % fi
88  ------------------------------------------------------------------------
89  -- Properties specification
90  ------------------------------------------------------------------------
91    -- Deadlock absence in components bahaviour
92  ------------------------------------------------------------------------
93  property FCP_Deadlock_Absence (MODEL)
94    "Check that the behaviour of FCP is deadlock-free"
95  is
96    "Diag.FCP_Deadlock_Absence.bcg" =
97      "${MODEL}.Act.bcg" |=
98        Always_Some ({FCP});
99    expected TRUE
100 end property
101 ------------------------------------------------------------------------
102 property AFS_COM_Deadlock_Absence (MODEL)
103   "Check that the behaviour of AFS_COM is deadlock-free"
104 is
105   "Diag.AFS_COM_Deadlock_Absence.bcg" =
106     "${MODEL}.Act.bcg" |=
107       Always_Some ({AFS_COM});
108   expected TRUE
109 end property
110 ------------------------------------------------------------------------
111 property AFS_MON_Deadlock_Absence (MODEL)
112   "Check that the behaviour of AFS_MON is deadlock-free"
```

```
113  is
114    "Diag.AFS_MON_Deadlock_Absence.bcg" =
115      "${MODEL}.Act.bcg" |=
116        Always_Some ({AFS_MON});
117    expected TRUE
118  end property
119  ------------------------------------------------------------------------
120     -- Progress of components inputs and outputs
121  ------------------------------------------------------------------------
122  property Progress_PILOT_ROTATION_PILOT_MVT (MODEL)
123    "Check that channel <pilot_rotation, pilot_mvt> continue to progress"
124  is
125    "Progress_PILOT_ROTATION_PILOT_MVT.bcg" =
126      "${MODEL}.Data.bcg" |=
127        Always_Some (true*.
128                     {PILOT_ROTATION_PILOT_MVT ?rot1:int ? mvt1:bool}.
129                     true*.
130                     {PILOT_ROTATION_PILOT_MVT ?rot2:int ?mvt2:bool
131                           where ((rot2 <> rot1) and (mvt2 <> mvt1))}
132        );
133    expected TRUE
134  end property
135  ------------------------------------------------------------------------
136  property Progress_AFS_COM_ROTATION (MODEL)
137    "Check that channel afs_com_rotation continue to progress"
138  is
139    "Progress_AFS_COM_ROTATION.bcg" =
140      "${MODEL}.Data.bcg" |=
141        Always_Some (true*.
142                     {AFS_COM_ROTATION ?rot1:int}.
143                     true*.
144                     {AFS_COM_ROTATION ?rot2:int where rot2 <> rot1}
145        );
146    expected TRUE
147  end property
148  ------------------------------------------------------------------------
149  property Progress_AFS_MON_MVT (MODEL)
150    "Check that channel afs_mont_mvt continue to progress"
151  is
152    "Progress_AFS_MON_MVT.bcg" =
153      "${MODEL}.Data.bcg" |=
154        Always_Some (true*.
155                     {AFS_MON_MVT ?mvt1:bool}.
156                     true*.
157                     {AFS_MON_MVT ?mvt2:bool where mvt2 <> mvt1}
158        );
159    expected TRUE
160  end property
161  ------------------------------------------------------------------------
162  property Progress_VALIDATED_TARGET (MODEL)
163    "Check that channel validated_target continue to progress"
164  is
```

```
165     "Progress_VALIDATED_TARGET.bcg" =
166       "${MODEL}.Data.bcg" |=
167         Always_Some (true*.
168                      {VALIDATED_TARGET ?targ1:int}.
169                      true*.
170                      {VALIDATED_TARGET ?targ2:int where (targ2 <> targ1)}
171         );
172     expected TRUE
173   end property
174   ------------------------------------------------------------------------
175   property Progress_CP_COM_TARGET_POSITION (MODEL)
176     "Check that channel cp_com_target_position continue to progress"
177   is
178     "Progress_CP_COM_TARGET_POSITION.bcg" =
179       "${MODEL}.Data.bcg" |=
180         Always_Some (true*.
181                      {CP_COM_TARGET_POSITION ?pos1:int}.
182                      true*.
183                      {CP_COM_TARGET_POSITION ?pos2:int where (pos2 <> pos1)}
184         );
185     expected TRUE
186   end property
187   ------------------------------------------------------------------------
188   property Progress_CP_MON_MVT (MODEL)
189     "Check that channel cp_mon_mvt continue to progress"
190   is
191     "Progress_CP_MON_MVT.bcg" =
192       "${MODEL}.Data.bcg" |=
193         Always_Some (true*.
194                      {CP_MON_MVT ?mvt1:bool}.
195                      true*.
196                      {CP_MON_MVT ?mvt2:bool where (mvt2 <> mvt1)}
197         );
198     expected TRUE
199   end property
200   ------------------------------------------------------------------------
201   property Progress_COM_TO_MON_AFS_TARGET_VALIDATION (MODEL)
202     "Check that channel com_to_mon_afs_target_validation continue to progress"
203   is
204     "Progress_COM_TO_MON_AFS_TARGET_VALIDATION.bcg" =
205       "${MODEL}.Data.bcg" |=
206          Always_Some (true*.
207                      {COM_TO_MON_AFS_TARGET_VALIDATION ?targ1:int}.
208                      true*.
209                      {COM_TO_MON_AFS_TARGET_VALIDATION ?targ2:int
210                        where (targ2 <> targ1)}
211          );
212     expected TRUE
213   end property
214   ------------------------------------------------------------------------
215   property Progress_MON_FROM_COM_AFS_TARGET_VALIDATION (MODEL)
216     "Check that channel mon_from_com_afs_target_validation continue to progress"
```

```
217  is
218    "Progress_MON_FROM_COM_AFS_TARGET_VALIDATION.bcg" =
219      "${MODEL}.Data.bcg" |=
220        Always_Some (true*.
221                     {MON_FROM_COM_AFS_TARGET_VALIDATION ?targ1:int}.
222                     true*.
223                     {MON_FROM_COM_AFS_TARGET_VALIDATION ?targ2:int
224                       where (targ2 <> targ1)}
225        );
226    expected TRUE
227  end property
228  ------------------------------------------------------------------------
229  property Progress_MON_TO_COM_AFS_TARGET_VALIDATED (MODEL)
230    "Check that channel <mon_to_com_afs_target_validated,"
231    "mon_to_com_afs_target_isvalid> continue to progress"
232  is
233    "Progress_MON_TO_COM_AFS_TARGET_VALIDATED.bcg" =
234      "${MODEL}.Data.bcg" |=
235        Always_Some (
236          true*.
237          {MON_TO_COM_AFS_TARGET_VALIDATED_MON_TO_COM_AFS_TARGET_ISVALID
238            ?targ1:int ?valid1:bool}.
239          true*.
240          {MON_TO_COM_AFS_TARGET_VALIDATED_MON_TO_COM_AFS_TARGET_ISVALID
241            ?targ2:int ?valid2:bool where
242            ((targ2 <> targ1) and (valid2 <> valid1))}
243        );
244    expected TRUE
245  end property
246  ------------------------------------------------------------------------
247  property Progress_COM_FROM_MON_AFS_TARGET_VALIDATED (MODEL)
248    "Check that channel <com_from_mon_afs_target_validated,"
249    "com_from_mon_afs_target_isvalid> continue to progress"
250
251  is
252    "Progress_COM_FROM_MON_AFS_TARGET_VALIDATED.bcg" =
253      "${MODEL}.Data.bcg" |=
254        Always_Some (
255          true*.
256          {COM_FROM_MON_AFS_TARGET_VALIDATED_COM_FROM_MON_AFS_TARGET_ISVALID
257            ?targ1:int ?valid1:bool}.
258          true*.
259          {COM_FROM_MON_AFS_TARGET_VALIDATED_COM_FROM_MON_AFS_TARGET_ISVALID
260            ?targ2:int ?valid2:bool where
261            ((targ2 <> targ1) and (valid2 <> valid1))}
262        );
263    expected TRUE
264  end property
265  ------------------------------------------------------------------------
266    -- Functional properties
267  ------------------------------------------------------------------------
268  property Movement_Detection_Causality (MODEL)
```

```
269     "Check that no movement detection information is sent to AFS"
270     "unless the knob is rotated"
271   is
272     "Movement_Detection_Causality.bcg" =
273       "${MODEL}.Data.bcg" |=
274         Not_To_Unless ({AFS_MON_MVT !FALSE},
275                        {AFS_MON_MVT !TRUE},
276                        {PILOT_ROTATION_PILOT_MVT ?any !TRUE}
277         );
278     expected TRUE
279   end property
280   ------------------------------------------------------------------------
281   property Rotation_and_Movement_Detection (MODEL)
282     "Check that a knob rotation is accompanied by a movement detection"
283   is
284     "Movement_Detection.bcg" =
285       "${MODEL}.Data.bcg" |=
286         Never ({PILOT_ROTATION_PILOT_MVT ?rot1:int ?mvt1:bool}.
287                not ({PILOT_ROTATION_PILOT_MVT ?any ?any}).
288                {PILOT_ROTATION_PILOT_MVT ?rot2:int ?mvt2:bool
289                  where ((rot2 <> rot1) and (mvt2 = FALSE))}
290         );
291     expected TRUE
292   end property
293   ------------------------------------------------------------------------
294   property Movement_Correlation (MODEL)
295     "A new knob position information sent to AFS_COM should be accompanied"
296     "by a movement detection information sent to AFS_MON"
297   is
298     "Movement_Detection.bcg" =
299       "${MODEL}.Data.bcg" |=
300         Never (true*.
301                {AFS_COM_ROTATION ?mvt1:int}.
302                not({AFS_COM_ROTATION ?any})*.
303                {AFS_COM_ROTATION ?mvt2:int where mvt2 <> mvt1}.
304                not ({AFS_MON_MVT ?any}).
305                {AFS_MON_MVT !TRUE}
306         );
307     expected TRUE
308   end property
309   ------------------------------------------------------------------------
310   property Movement_Detection_Sending (MODEL)
311     "Check that whenever the knob is rotated, a movement detection"
312     "information is sent to AFS"
313   is
314     "Movement_Detection_Sending.bcg" =
315       "${MODEL}.Data.bcg" |=
316         Never (true*.
317                {PILOT_ROTATION_PILOT_MVT ?any !TRUE}.
318                not({AFS_MON_MVT ?any})*.
319                {AFS_MON_MVT !FALSE}
320         );
```

```
321   expected TRUE
322 end property
323 ------------------------------------------------------------------------
324 property Movement_Detection_Sustain_FCP_1 (MODEL, FCP_MVT_PROLONG)
325   "Check that the detection movement information is maintained TRUE"
326   "for ${FCP_MVT_PROLONG} cycle duration"
327 is
328   "Movement_Detection_Sustain_1.bcg" =
329     "${MODEL}.Data.bcg" |=
330       Sustain (true*.
331                {PILOT_ROTATION_PILOT_MVT ?any !TRUE}.
332                not ({PILOT_ROTATION_PILOT_MVT ?any ?any})*.
333                {PILOT_ROTATION_PILOT_MVT ?any !FALSE},
334                {AFS_MON_MVT !TRUE},
335                {AFS_MON_MVT !FALSE},
336                ${FCP_MVT_PROLONG}
337       );
338   expected TRUE
339 end property
340 ------------------------------------------------------------------------
341 property Movement_Detection_Sustain_FCP_2 (MODEL, FCP_MVT_PROLONG)
342   "Check that the movement signal information is not maintained TRUE"
343   "for more than FCP_MVT_PROLONG cycle duration if no knob rotation"
344   "has occured in the meanwhile"
345 is
346   "Movement_Detection_Sustain_2.bcg" =
347     "${MODEL}.Data.bcg" |=
348        Never (true*.
349               {PILOT_ROTATION_PILOT_MVT ?any !TRUE}.
350               ((not ({PILOT_ROTATION_PILOT_MVT ?any ?any}))*.
351                {PILOT_ROTATION_PILOT_MVT ?any !FALSE}.
352                (not ({AFS_MON_MVT ?any}
353                 or {PILOT_ROTATION_PILOT_MVT ?any ?any}))*.
354                {AFS_MON_MVT !TRUE}
355               ){${FCP_MVT_PROLONG}}
356        );
357   expected TRUE
358 end property
359 ------------------------------------------------------------------------
360 property Fault_Tolerance (MODEL)
361   "Check that the system is fault tolerant"
362 is
363   "Diag.Fault_Tolerance.${MODEL}.bcg" =
364     "${MODEL}.bcg" |=
365       Never (true*. {FAULT_TOLERANCE !"FALSE"});
366   expected TRUE
367 end property
368 ------------------------------------------------------------------------
369 property System_Movement_Observation (MODEL, RESULT)
370   "Check that a movement detected in FCP is sustained enough to be observed"
371   "by AFS_MON"
372 is
```

```
373     "Diag.System_Movement_Observation.${MODEL}.bcg" =
374       "${MODEL}.bcg" |=
375         NEVER ((not {TO_AFS_MON_MVT !TRUE}) *. {TO_AFS_MON_MVT !TRUE}.
376                (not {FROM_CP_MON_MVT !TRUE})*. {TO_AFS_MON_MVT !FALSE}.
377                (not {TO_AFS_MON_MVT !TRUE}) *. {TO_AFS_MON_MVT !TRUE}
378         );
379     result "$RESULT" expected TRUE
380   end property
381   ------------------------------------------------------------------------
382   property System_Movement_Detection (MODEL, RESULT)
383     "Check that no new altitude order is provided by AFS_COM unless a movement"
384     "has been detected in AFS_MON"
385   is
386     "Diag.System_Movement_Detection.${MODEL}.bcg" =
387        "${MODEL}.bcg" |=
388          Not_to_Unless ({MON_FROM_COM_AFS_TARGET_VALIDATION ? targ1:int},
389                         {MON_FROM_COM_AFS_TARGET_VALIDATION ? targ2:int
390                           where targ1 <> targ2},
391                         {FROM_CP_MON_MVT !"TRUE"}
392          );
393     result "$RESULT" expected TRUE
394   end property
395   ------------------------------------------------------------------------
396   -- Verification of LTSs
397   ------------------------------------------------------------------------
398   % case "$MODEL" in
399   %   "FCP_Component" | "AFCS_System") \
400         check FCP_Deadlock_Absence ("$MODEL");
401         check Progress_PILOT_ROTATION_PILOT_MVT ("$MODEL");
402         check Progress_AFS_COM_ROTATION ("$MODEL");
403         check Progress_AFS_MON_MVT ("$MODEL");
404         check Movement_Detection_Causality ("$MODEL");
405         check Rotation_and_Movement_Detection ("$MODEL");
406         check Movement_Correlation ("$MODEL");
407         check Movement_Detection_Sending ("$MODEL");
408         check Movement_Detection_Sustain_FCP_1 ("$MODEL", "$FCP_MVT_PROLONG");
409         check Movement_Detection_Sustain_FCP_2 ("$MODEL", "$FCP_MVT_PROLONG");
410        % ;;
411   %   "AFS_Component" | "AFCS_System") \
412         check AFS_COM_Deadlock_Absence ("$MODEL");
413         check AFS_MON_Deadlock_Absence ("$MODEL");
414         check Progress_VALIDATED_TARGET ("$MODEL");
415         check Progress_CP_COM_TARGET_POSITION ("$MODEL");
416         check Progress_CP_MON_MVT ("$MODEL");
417         check Progress_COM_TO_MON_AFS_TARGET_VALIDATION ("$MODEL");
418         check Progress_MON_FROM_COM_AFS_TARGET_VALIDATION ("$MODEL");
419         check Progress_MON_TO_COM_AFS_TARGET_VALIDATED ("$MODEL");
420         check Progress_COM_FROM_MON_AFS_TARGET_VALIDATED ("$MODEL");
421   %   ;;
422   %   "AFCS_System" ) \
423         check Fault_Tolerance ("$MODEL");
424         check System_Movement_Observation ("$MODEL", "RESULT");
```

```
425         check System_Movement_Detection ("$MODEL", "RESULT");
426 %       if [ "$CHECK_PARAMETRISED_MODEL" = "1" ]
427 %       then
428 %         for i in 3 7 10 13 17 20
429 %         do
430 %           for j in 1 2 3 4 5 6
431 %           do
432 %             MODEL="AFCS_CP_${i}_AFS_${j}"
433                 check System_Movement_Observation ("$MODEL", "RESULT");
434 %               if [ "$RESULT" = FALSE ]
435 %               then
436                   "Diag.System_Movement_Observation.${MODEL}.bcg" =
437                     total safety reduction of
438                       partial hide all but
439                         "MON_FROM_COM_AFS_TARGET_VALIDATION.*",
440                         "FROM_CP_MON_MVT.*"
441                       in "Diag.System_Movement_Observation.${MODEL}.bcg";
442 %               fi
443                 check System_Movement_Detection ("$MODEL", "RESULT");
444 %               if [ "$RESULT" = FALSE ]
445 %               then
446                   "Diag.System_Movement_Detection.${MODEL}.bcg" =
447                     total safety reduction of
448                       partial hide all but
449                         "TO_AFS_MON_MVT.*",
450                         "FROM_CP_MON_MVT.*"
451                       in "Diag.System_Movement_Detection.${MODEL}.bcg";
452 %               fi
453 %           done
454 %         done
455 %       fi
456 %     ;;
457 % esac
458   ------------------------------------------------------------------------
```

A.2.2 Property patterns

```
1   (* ------------------------------------------------------------------ *)
2   (* File library.mcl *)
3   (* ------------------------------------------------------------------ *)
4   macro Never (A) =
5     [A] false
6   end_macro
7   (* ------------------------------------------------------------------ *)
8   macro Not_TO_Unless (A, B, C) =
9     [true*. A. (not (C))*. B] false
10  end_macro
11  (* ------------------------------------------------------------------ *)
12  macro After_Inev (A, B) =
13    [true*. A] inev (B)
14  end_macro
15  (* ------------------------------------------------------------------ *)
```

```
16  macro inev (A) =
17    mu X . ( <true> true and [not (A)] X )
18  end_macro
19  (* ------------------------------------------------------------------------ *)
20  macro Always_Some (A) =
21    [true*] < true* . A> true
22  end_macro
23  (* ------------------------------------------------------------------------ *)
24  macro Alive_A (A) =
25    [true*. A] <true*. A> true
26  end_macro
27  (* ------------------------------------------------------------------------ *)
28  macro Deadline (R, A1, A2, n) =
29    Never (R. (not (A1))*. (A1. (not (A1 or A2))*){n+1})
30  end_macro
31  (* ------------------------------------------------------------------------ *)
32  macro Not_To_Unless_Most (A1, A2, A3, n) =
33    Deadline (true* . A1, A3, A2, n)
34  end_macro
35  (* ------------------------------------------------------------------------ *)
36  macro Sustain (R, A1, A2, n) =
37    [ R ] nu Counter (c: nat := 1) . (
38            ((c < n) implies ([ A2 ] false and [ A1 ] Counter (c + 1)))
39            and [ not (A1 or A2) ] Counter (c)
40           )
41  end_macro
42  (* ------------------------------------------------------------------------ *)
```

Appendix B

The GRL Model of the Car Park Application

This appendix presents the GRL model of the car park application.

B.1 Global constants

```
1  ------------------------------------------------------------------------
2  -- Global constants
3  const Park_Size: int16 := 5
4  const Cst_Bool_Empty_Buffer: bool := false
5  ------------------------------------------------------------------------
```

B.2 Subblocks modelling function blocks

```
1  ------------------------------------------------------------------------
2  -- Numerical constant block
3  block B_Num {Numeric_Constant : int16}
4              (out Numeric_Value : int16)
5  is
6    Numeric_Value := Numeric_Constant
7  end block
8  ------------------------------------------------------------------------
9  -- Logic And block (2 inputs)
10 block B_And (in  Left  : bool := true,
11              in  Right : bool := true,
12              out Res   : bool)
13 is
14   Res := (Left and Right)
15 end block
16 ------------------------------------------------------------------------
17 -- Logic And block (4 inputs)
18 block B_And_4 (in  In1  : bool := true,
19                in  In2 : bool := true,
```

```
20                in  In3 : bool := true,
21                in  In4 : bool := true,
22                out Res : bool)
23  is
24    Res := (In1 and In2 and In3 and In4)
25  end block
26  ------------------------------------------------------------------------
27  -- Logic Or block
28  block B_Or (in  Left  : bool := false,
29              in  Right : bool := false,
30              out Res   : bool) is
31    Res := (Left or Right)
32  end block
33  ------------------------------------------------------------------------
34  -- Logic Not block
35  block B_Not (in  Input: bool := false,
36               out Res  : bool)
37  is
38    Res := not (Input)
39  end block
40  ------------------------------------------------------------------------
41  -- Timer block
42  block B_Edge {Rising_Mode: bool := true, Falling_Mode: bool := false}
43               (in  Logic_Signal : bool := true,
44                out Edge_Detected: bool) is
45    static var Pre_Signal: bool := false
46    var Rise, Fall       : bool
47    Rise          := Logic_Signal and not (Pre_Signal);
48    Fall          := not (Rise);
49    Edge_Detected := (Rising_Mode and Rise) or (Falling_Mode and Fall);
50    Pre_Signal    := Logic_Signal
51  end block
52  ------------------------------------------------------------------------
53  -- Comparator block
54  block B_Compare {Mode : Type_Comparison := Equal}
55                  (in Validation : bool := true,
56                   in Left, Right : int16 := 0, out Res : bool)
57  is
58    case Mode is
59      Equal             -> Res := (Left == Right)
60    | Striclty_Superior -> Res := (Left > Right)
61    | Superior          -> Res := (Left >= Right)
62    | Strictly_Inferior -> Res := (Left < Right)
63    | Inferior          -> Res := (Left <= Right)
64    | Not_Equal         -> Res := (Left != Right)
65    end case;
66    Res := Res and Validation
67  end block
68
69  type Type_Comparison is
70    enum Equal, Striclty_Superior, Superior,
71         Strictly_Inferior, Inferior, Not_Equal
```

```
72  end type
73  ------------------------------------------------------------------------
74  -- Counter block
75  block B_UpDown_Count (in  Increase : bool := false,
76                        in  Decrease : bool := false,
77                        in  Reset    : bool := false,
78                        in  Preset   : bool := false,
79                        in  Preset_Value : int16 := 0,
80                        out Output : bool,
81                        out Current_Value : int16) is
82    static var Pre_Current_Value : int16 := 0,
83               Pre_Increase : bool := false,
84               Pre_Decrease : bool := false
85    if (Reset) then
86      Current_Value := 0;
87      Output := false
88    else
89      if (Preset) then
90        Current_Value := Preset_Value;
91        Output := false
92      else
93        if ((Increase) and not (Pre_Increase)
94            and (Pre_Current_Value < Preset_Value)) then
95          Current_Value := Pre_Current_Value + 1
96        else
97          Current_Value := Pre_Current_Value
98        end if;
99        if ((Decrease) and not (Pre_Decrease)
100           and (Pre_Current_Value > -32768)) then
101         Current_Value := Current_Value - 1
102       else
103         Current_Value := Current_Value
104       end if;
105       if (Current_Value == Preset_Value) then
106         Output := true
107       else
108         Output := false
109       end if
110     end if
111   end if;
112   Pre_Current_Value := Current_Value;
113   Pre_Increase := Increase;
114   Pre_Decrease := Decrease
115 end block
116 ------------------------------------------------------------------------
```

B.3 Highest-level blocks modelling PLCs

```
1  ------------------------------------------------------------------------
2  -- Block modelling the exit PLC
3  block Exit (in Cmd_P1, Cmd_P2: bool, out  Open   : bool)
4             [send Out_P1       : bool, send Out_P2: bool] is
```

```
5     var Edge_Cmd_P1, Edge_Cmd_P2 : bool
6     -- if a car parking in Storey1 arrives
7     B_Edge {true, false}(Cmd_P1, ?Edge_Cmd_P1); -- does the car ask for leaving?
8     B_And  (Edge_Cmd_P1, Cmd_P1, ?Out_P1);      -- inform Storey1
9     -- if a car parking in Storey2 arrives
10    B_Edge {true, false}(Cmd_P2, ?Edge_Cmd_P2); -- does the car ask for leaving?
11    B_And  (Edge_Cmd_P2, Cmd_P2, ?Out_P2);      -- inform Storey2
12    -- if a car asks for leaving, open the gate
13    B_Or   (Edge_Cmd_P1, Edge_Cmd_P2, ?Open)
14  end block
15  ------------------------------------------------------------------------
16  -- Block modelling a storey PLC
17  block Storey {Id_P1, Id_P2  : bool}
18                (in    Cmd_P1, Cmd_P2: bool,
19                 out   Open, Err     : bool)
20                [receive Car_Left_from_exit  : bool,
21                 send    Car_Left_to_entrance: bool] is
22    alias B_Edge {true, false} as Edge_Cmd_P1; Edge_Cmd_P2,
23          B_And                as B2_And; B3_And; B5_And; B6_And,
24          B_Or                 as B7_Or; B8_Or
25    var edge1, edge2, edge1_P1, edge1_P2, edge2_P1, edge2_P2: bool
26    Edge_Cmd_P1 (Cmd_P1, ?edge1);
27    B2_And      (edge1, Id_P1, ?edge1_P1);
28    B3_And      (edge1, Id_P2, ?edge1_P2);
29    Edge_Cmd_P2 (Cmd_P2, ?edge2);
30    B5_And      (edge2, Id_P2, ?edge2_P2);
31    B6_And      (edge2, Id_P1, ?edge2_P1);
32    B7_Or       (edge1_P1, edge2_P2, ?Open);
33    B8_Or       (edge1_P2, edge2_P1, ?Err);
34    Car_Left_to_entrance := Car_Left_from_exit
35  end block
36  ------------------------------------------------------------------------
37  -- Block modelling the entrance PLC
38  block Entrance {Size:int16 := Park_Size}
39                (in  Cmd: bool,
40                 out Green, Yellow, Red: bool,
41                 out Open, P1, P2: bool)
42                [receive Out1 : bool, receive Out2 : bool] is
43    alias B_Edge {true, false} as B1,
44          B_And_4 as B2,
45          B_Num {Size} as B3,
46          B_Compare {Strictly_Inferior} as B4,
47          B_UpDown_Count as B5,
48          B_Not as B6,
49          B_Not as B7,
50          B_And as B8,
51          B_And as B9,
52          B_And_4 as B10,
53          B_Compare {Strictly_Inferior} as B11,
54          B_UpDown_Count as B12,
55          B_Not as B13,
56          B_Not as B14,
```

```
57            B_And as B15,
58            B_And as B16,
59            B_And as B17,
60            B_Or as B18,
61            B_Not as B19,
62            B_And as B20,
63            B_Or as B21
64      static var pre_c5  : bool := true,
65                 pre_c9  : bool := true,
66                 pre_c10 : bool := true
67      var c1, c2, c3, c4, c5, c7, c9, c10, c11, c12, c13, c14, c15, c16 : bool,
68          c6, c8 : int16
69      B1 (Cmd, ?c1);
70      B2 (c1, pre_c9, pre_c10, _, ?c4);
71      B3 (?c6);
72      B5 (c4, Out1, _, _, c6, ?c7, ?c8);
73      B4 (_, c8, c6, ?c9);
74      P1 := c4;
75      B6 (P1, ?c10);
76      B8 (c7, c10, ?c12);
77      B7 (c7, ?c11);
78      B9 (c11, c10, ?c14);
79      B10 (c1, pre_c5, pre_c10, not (P1), ?c2);
80      B12 (c2, Out2, _, _, c6, ?c7, ?c8);
81      B11 (_, c8, c6, ?c5);
82      P2 := c2;
83      B13 (P2, ?c3);
84      B15 (c7, c3, ?c13);
85      B14 (c7, ?c11);
86      B16 (c11, c3, ?c15);
87      B17 (c12, c13, ?Red);
88      B18 (P1, P2, ?Open);
89      B21 (c14, c15, ?c16);
90      B20 (c16, not (Open), ?Green);
91      Yellow := Open;
92      B19 (Open, ?c10);
93      pre_c5 := c5;
94      pre_c9 := c9;
95      pre_c10 := c10
96  end block
97  ------------------------------------------------------------------------------
```

B.4 Environments

```
1  ------------------------------------------------------------------------------
2  -- Environment ensuring that:
3  ---- a leaving request cannot occur if there is no car in the car park
4  ---- an entrance or exit request cannot occur in two successive steps of a PLC
5  ---- a ticket given to a car references exactly one storey
6  environment Env_Cmd (in  Park_Open, Park_P1, Park_P2: bool,
7                       in  Out_Open: bool,
8                       out Cmd: bool,
```

```
 9                        out Exit_P1, Exit_P2: bool,
10                        out Cmd_P11, Cmd_P12: bool,
11                        out Cmd_P21, Cmd_P22: bool
12                       )
13  is
14    static var Pre_Cmd, Pre_Exit, Pre_Cmd_1, Pre_Cmd_2: bool := false,
15               Nb_Car: nat := 0
16    select
17      when Cmd -> if Pre_Cmd then
18                    Cmd := false
19                  else
20                    Cmd := any bool
21                  end if;
22                  Pre_Cmd := Cmd
23    []
24      when <Exit_P1, Exit_P2> -> if ((Nb_Car == 0) or Pre_Exit) then
25                                   Exit_P1 := false;
26                                   Exit_P2 := false
27                                 else
28                                   Exit_P1 := any bool;
29                                   Exit_P2 := any bool
30                                              where not (Exit_P1 and Exit_P2)
31                                 end if;
32                                 if (Exit_P1 or Exit_P2) then
33                                    Pre_Exit := true
34                                 end if
35    []
36      when <Cmd_P11, Cmd_P12> -> if ((Nb_Car == 0) or Pre_Cmd_1) then
37                                   Cmd_P11 := false;
38                                   Cmd_P12 := false
39                                 else
40                                   Cmd_P11 := any bool;
41                                   Cmd_P12 := any bool
42                                              where not (Cmd_P11 and Cmd_P12)
43                                 end if;
44                                 if (Cmd_P11 or Cmd_P12) then
45                                    Pre_Cmd_1 := true
46                                 end if
47    []
48      when <Cmd_P21, Cmd_P22> -> if ((Nb_Car == 0) or Pre_Cmd_2) then
49                                   Cmd_P21 := false;
50                                   Cmd_P22 := false
51                                 else
52                                   Cmd_P21 := any bool;
53                                   Cmd_P22 := any bool
54                                              where not (Cmd_P21 and Cmd_P22)
55                                 end if;
56                                 if (Cmd_P21 or Cmd_P22) then
57                                    Pre_Cmd_2 := true
58                                 end if
59    []
60      when ?<Park_Open, Park_P1, Park_P2> -> if (Park_Open) then
```

```
61                                                    Nb_Car := Nb_Car + 1
62                                                  end if;
63                                                  Park_P1 := Park_P1;
64                                                  Park_P2 := Park_P2
65
66      []
67        when ?Out_Open -> if (Out_Open) then
68                            Nb_Car := Nb_Car - 1
69                          end if
70      end select
71    end environment
72    ------------------------------------------------------------------------
73    -- Environment ensuring that two blocks evolve at the same pace
74    environment Quasisynch_2 (block Storey1, Storey2) is
75      static var ok1, ok2: bool := true  -- permission for blocks to execute
76      select
77         if (ok1) then -- execution of Storey1 if permitted
78            enable Storey1;
79            ok1 := false
80         end if
81      []
82         if (ok2) then -- execution of Storey2 if permitted
83            enable Storey2;
84            ok2 := false
85         end if
86      end select;
87      -- if both blocks have been executed once, reinitialize permissions
88      if (not (ok1) and not (ok2)) then
89          ok1 := true;
90          ok2 := true
91      end if
92    end environment
93    ------------------------------------------------------------------------
94    -- Environment ensuring that four blocks evolve at the same pace
95    environment Quasisynch_4 (block B1, B2, B3, B4) is
96      static var ok1, ok2, ok3, ok4: bool := true
97      select
98        if (ok1) then
99          enable B1;
100         ok1 := false
101       end if
102     []
103       if (ok2) then
104         enable B2;
105         ok2 := false
106       end if
107     []
108       if (ok3) then
109         enable B3;
110         ok3 := false
111       end if
112     []
```

```
113       if (ok4) then
114         enable B4;
115         ok4 := false
116       end if
117     end select;
118     if (not (ok1) and not (ok2) and not (ok3) and not (ok4)) then
119         ok1 := true;
120         ok2 := true;
121         ok3 := true;
122         ok4 := true
123     end if
124   end environment
125   ------------------------------------------------------------------------
126   -- Environments describing scenarios between blocks
127   ------------------------------------------------------------------------
128   type cases is
129     enum Car_Park, Car_P1, Car_P2, Car_Ex, None
130   end type
131   ------------------------------------------------------------------------
132   environment Scen_Act (block Entrance, Exit, Storey1, Storey2) is
133     static var action:cases := Car_Park
134     case action is
135       Car_Park -> action := Car_P2; enable Entrance
136     | Car_P2   -> action := Car_P1; enable Storey2
137     | Car_P1   -> action := Car_Ex; enable Storey1
138     | Car_Ex   -> action := None; enable Exit
139     | any      -> action := None
140     end case
141   end environment
142   ------------------------------------------------------------------------
143   environment Scen_Data (out Cmd_Park          : bool, out Cmd_P11, Cmd_P12: bool,
144                          out Cmd_P21, Cmd_P22: bool, out Exit_P1, Exit_P2: bool)
145   is
146     select
147       when <Cmd_Park>          -> Cmd_Park := true  -- Entrance data
148     [] when <Cmd_P21, Cmd_P22> -> Cmd_P21  := true; -- Storey1 data
149                                   Cmd_P22  := false
150     [] when <Cmd_P11, Cmd_P12> -> Cmd_P11  := true; -- Storey2 data
151                                   Cmd_P12  := false
152     [] when <Exit_P1, Exit_P2> -> Exit_P1  := true; -- Exit data
153                                   Exit_P2  := false
154     end select
155   end environment
156   ------------------------------------------------------------------------
```

B.5 Mediums

```
1   ------------------------------------------------------------------------
2   -- Medium modelling a lossy communication
3   medium Buffer_Bit [receive Input : bool,
4                      send    Output: bool]
5   is
```

```
6     static var Buffer : bool := Cst_Bool_Empty_Buffer
7     select
8       when ?Input -> select
9                        Buffer := Input
10                    []  null
11                    end select
12    []
13      when Output -> if (Buffer == Cst_Bool_Empty_Buffer) then
14                       Output := Cst_Bool_Default_Value
15                    else
16                       Output := Buffer;
17                       Buffer := Cst_Bool_Empty_Buffer
18                    end if
19    end select
20  end medium
21  ------------------------------------------------------------------------
```

B.6 Systems

```
1   ------------------------------------------------------------------------
2   -- System to generate the LTS corresponding to the entrance PLC
3   system Main_Entrance {Size:int16 := 1}
4                        (Cmd: bool, Green, Yellow, Red: bool,
5                         Open, P1, P2: bool, Out1: bool, Out2: bool)
6   is
7     block list
8       Entrance {Size}(Cmd, ?<Green, Yellow, Red>, ?<Open, P1, P2>)[Out1, Out2]
9   end system
10  ------------------------------------------------------------------------
11  -- System to generate the LTS corresponding to storey PLCs
12  system Main_Storey (Cmd_P11, Cmd_P21 : bool, R_Out1, S_Out1 : bool,
13                      Open1 : bool, Err1 : bool)
14  is
15     alias Storey {true, false} as Storey
16    block list
17       Storey (<Cmd_P11, Cmd_P21>, ?<Open1, Err1>)
18              [R_Out1, ?S_Out1]
19  end system
20  ------------------------------------------------------------------------
21  -- System to generate the LTS corresponding to the exit PLC
22  system Main_Exit (Cmd_P1, Cmd_P2: bool, Open: bool, Out_P1: bool, Out_P2: bool)
23  is
24    block list
25      Exit (<Cmd_P1, Cmd_P2>, ?Open)[?Out_P1, ?Out_P2]
26  end system
27  ------------------------------------------------------------------------
28  -- System to generate the LTS corresponding to a scenario of the car park
29  system Main_Scen (Cmd_Park, Park_P1, Park_P2: bool,
30                 Cmd_P11, Cmd_P12, Cmd_P21, Cmd_P22, Exit_P1, Exit_P2: bool,
31                 Green, Yellow, Red : bool,
32                 Park_Open, Open1, Open2, Out_Open : bool,
33                 Err1, Err2 : bool) is
```

```
34      alias Storey {true, false}  as Storey1,
35            Storey {false, true}  as Storey2,
36            Entrance {Park_Size} as Entrance,
37            Exit as Exit,
38            Scen_Data as Scen_Data,
39            Scen_Act  as Scen_Act,
40            Buffer_Bit as Med1; Med2; Med3; Med4
41       var S_Out1, S_Out2, R_Out1, R_Out2 : bool,
42            S_Full1, S_Full2, R_Full1, R_Full2 : bool
43       block list
44          Exit     (<Exit_P1, Exit_P2>, ?<Out_Open>) [?S_Out1, ?S_Out2],
45          Storey1  (<Cmd_P11, Cmd_P12>, ?<Open1, Err1>) [R_Out1, ?S_Full1],
46          Storey2  (<Cmd_P21, Cmd_P22>, ?<Open2, Err2>) [R_Out2, ?S_Full2],
47          Entrance (Cmd_Park, ?<Green, Yellow, Red>,
48                    ?<Park_Open, Park_P1, Park_P2>)
49                   [R_Full1, R_Full2]
50       environment list
51          Scen_Act  (Entrance, Storey1, Storey2, Exit),
52          Scen_Data (?<Cmd_Park>, ?<Cmd_P11, Cmd_P12>, ?<Cmd_P21, Cmd_P22>,
53                     ?<Exit_P1, Exit_P2>)
54       medium list
55          Med1 [S_Out1, ?R_Out1],
56          Med2 [S_Out2, ?R_Out2],
57          Med3 [S_Full1, ?R_Full1],
58          Med4 [S_Full2, ?R_Full2]
59   end system
60   ------------------------------------------------------------------------
61   -- System to generate the LTS corresponding to a car park with quasi-synchrony
62   -- and constraints on block inputs
63   system Main_Quasi {Size: int16}
64                     (Cmd_Park, Park_P1, Park_P2: bool,
65                      Cmd_P11, Cmd_P12, Cmd_P21, Cmd_P22, Exit_P1, Exit_P2: bool,
66                      Green, Yellow, Red : bool,
67                      Park_Open, Open1, Open2, Out_Open : bool,
68                      Err1, Err2 : bool)
69   is
70       alias Storey {true, false}  as Storey1,
71             Storey {false, true}  as Storey2,
72             Entrance  {Park_Size} as Entrance,
73             Exit as Exit,
74             Quasisynch_4 as Env_Act,
75             Env_Cmd as Env_Data,
76             Buffer_Bit as Med1; Med2; Med3; Med4
77       var  S_Out1, S_Out2, R_Out1, R_Out2     : bool,
78            S_Full1, S_Full2, R_Full1, R_Full2: bool
79       block list
80          Exit     (<Exit_P1, Exit_P2>, ?Out_Open) [?S_Out1, ?S_Out2],
81          Storey1  (<Cmd_P11, Cmd_P12>, ?<Open1, Err1>) [R_Out1, ?S_Full1],
82          Storey2  (<Cmd_P21, Cmd_P22>, ?<Open2, Err2>) [R_Out2, ?S_Full2],
83          Entrance (Cmd_Park, ?<Green, Yellow, Red>,
84                                ?<Park_Open, Park_P1, Park_P2>)
85                   [R_Full1, R_Full2]
```

```
86        environment list
87          Env_Act    (Exit, Storey1, Storey2, Entrance),
88          Env_Data   (<Park_Open, Park_P1, Park_P2>, Out_Open, ?Cmd_Park,
89                      ?<Exit_P1, Exit_P2>, ?<Cmd_P11, Cmd_P12>, ?<Cmd_P21, Cmd_P22>)
90        medium list
91           Med1 [S_Out1, ?R_Out1],
92           Med2 [S_Out2, ?R_Out2],
93           Med3 [S_Full1, ?R_Full1],
94           Med4 [S_Full2, ?R_Full2]
95    end system
96    ------------------------------------------------------------------------------
```

Bibliography

[AB84] Didier Austry and Gérard Boudol. Algèbre de Processus et Synchronisation. *Theor. Comput. Sci.*, 30:91–131, 1984.

[AFFSV01] M. Antoniotti, A. Ferrari, A. Flesca, and A. Sangiovanni-Vincentelli. *System-on-Chip Methodologies & Design Languages*, chapter Jester, pages 205–214. Springer US, Boston, MA, 2001.

[AL03] Rajeev Alur and Insup Lee, editors. *Embedded Software, Third International Conference, EMSOFT 2003, Philadelphia, PA, USA, October 13-15*, volume 2855 of *Lecture Notes in Computer Science*. Springer, 2003.

[Amj04] Hasan Amjad. *Combining model checking and theorem proving*. Technical report, University of Cambridge, 2004.

[And95] Charles André. SyncCharts: A visual representation of reactive behaviors. *Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis*, 1995.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.

[BBC10] Albert Benveniste, Anne Bouillard, and Paul Caspi. A unifying view of loosely time-triggered architectures. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 189–198. ACM, 2010.

[BBJ14] Pierre-Alain Bourdil, Bernard Berthomieu, and Eric Jenn. Model-checking real-time properties of an auto flight control system function. In *25th IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Naples, Italy, November 3-6, 2014*, pages 120–123, 2014.

[BBP15] Guillaume Baudart, Timothy Bourke, and Marc Pouzet. Soundness of the Quasi-Synchronous Abstraction. Research Report RR-8755, INRIA Paris-Rocquencourt ; INRIA, August 2015.

[BBS12] Yu Bai, Jens Brandt, and Klaus Schneider. Preservation of LTL properties in desynchronized systems. In *Tenth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMCODE 2012, Arlington, VA, USA, July 16-17, 2012*, pages 53–64, 2012.

[BCG99] Albert Benveniste, Benoît Caillaud, and Paul Le Guernic. From Synchrony to Asynchrony. In *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999*, pages 162–177, 1999.

[BCMW15] John Backes, Darren D. Cofer, Steven P. Miller, and Mike Whalen. Requirements analysis of a quad-redundant flight control system. *CoRR*, abs/1502.03343, 2015.

[Ber89] Gérard Berry. Real Time Programming: Special Purpose or General Purpose Languages. In *IFIP Congress*, pages 11–17, 1989.

[BG92] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[BHS91] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with Applications from Protocol Specification*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

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

[BMY+14] S. Bhattacharyya, S. Miller, J. Yang, S. Smolka, B. Meng, C. Sticksel, and C. Tinelli. Verification of quasi-synchronous systems with Uppaal. In *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, pages 8A4–1–8A4–12, Oct 2014.

[Bou98] Amar Bouali. XEVE, an ESTEREL Verification Environment. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998*, pages 500–504, 1998.

[BRS93] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 85–98, 1993.

[BS01] Gérard Berry and Ellen Sentovich. Multiclock Esterel. In *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001*, pages 110–125, 2001.

[Cas00] Paul Caspi. The quasi-synchronous approach to distributed control systems. Technical Report CMA/009931, VERIMAG, 2000.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ Analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Edinburgh, UK, April 4-8*, pages 21–30, 2005.

[CCG+16] 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 6.4). INRIA/VASY and INRIA/CONVECS, 131 pages, April 2016.

[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 *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Rome, Italy, March 16-24, 2013*, pages 199–213, 2013.

[CGP00] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[Cha84] Daniel M Chapiro. Globally-Asynchronous Locally-Synchronous Systems. Technical report, DTIC Document, 1984.

[CLS00] Rance Cleaveland, Tan Li, and Steve Sims. The Concurrency Workbench of the New Century, Version 1.2 - User's Manual, 2000.

[CPS89] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench. In *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989*, pages 24–37, 1989.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice, March 4-5, 1998, Clearwater Beach, Florida, USA*, pages 7–15, 1998.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420, 1999.

[DFGR93] Rocco De Nicola, Alessandro Fantechi, Stefania Gnesi, and Gioia Ristori. An Action-Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems. *Computer Networks and ISDN Systems*, 25(7):761–778, 1993.

[Dij65] Edsger Wybe Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 8(9):569–, September 1965.

[Dij72] Edsger Wybe Dijkstra. Information streams sharing a finite buffer. *Information Processing Letters*, 1(5):179 – 180, 1972.

[Dij82] Edsger Wybe Dijkstra. On the role of scientific thought (EWD447). *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[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. *Electr. Notes Theor. Comput. Sci.*, 146(2):105–131, 2006.

[DNV90] Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Berlin Heidelberg, 1990.

[DO-11] DO-333. Formal Methods Supplement to DO-178C and DO-278A, 2011.

[EC82] E Allen Emerson and Edmund M Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming*, 2(3):241–266, 1982.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[FFG14] Alessandro Fantechi, Francesco Flammini, and Stefania Gnesi. Formal methods for railway control systems. *STTT*, 16(6):643–646, 2014.

[FHB+97] Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Samuel Boutin, Eduardo Giménez, Gérard Huet, César Muñoz, Cristina Cornes, Judicaël Courant, Chetan Murthy, Catherine Parent, Christine Paulin-mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof

Assistant Reference Manual Version 6.1. Technical report, INRIA-Rocquencourt-CNRS-ENS Lyon, December 1997.

[GABR14] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Grenoble, France, April 5-13*, pages 187–201, 2014.

[GG03] Abdoulaye Gamatié and Thierry Gautier. The SIGNAL Approach to the Design of System Architectures. In *10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003), 7-10 April 2003, Huntsville, AL, USA*, pages 80–88, 2003.

[GG07] M. K. Ganai and A. Gupta. Efficient BMC for multi-clock systems with clocked specifications. In *Design Automation Conference, 2007. ASP-DAC'07. Asia and South Pacific*, pages 310–315. IEEE, 2007.

[GG10] Abdoulaye Gamatié and Thierry Gautier. The Signal Synchronous Multiclock Approach to the Design of Distributed Embedded Systems. *IEEE Trans. Parallel Distrib. Syst.*, 21(5):641–657, 2010.

[GGTG10] Y. Glouche, P. Le Guernic, J.-P. Talpin, and T. Gautier. A boolean algebra of contracts for assume-guarantee reasoning. *Electronic Notes in Theoretical Computer Science*, 263:111 – 127, 2010. Proceedings of the 6th International Workshop on Formal Aspects of Component Software (FACS 2009).

[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.

[GLM02] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compiler Construction using LOTOS NT. In Nigel Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction (CC'02), Grenoble, France*, volume 2304 of *Lecture Notes in Computer Science*, pages 9–13. Springer, April 2002.

[GLM15] Hubert Garavel, Frédéric Lang, and Radu Mateescu. Compositional Verification of Asynchronous Concurrent Systems Using CADP. *Acta Informatica*, 52(4):337–392, April 2015.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107, 2013.

[GM03] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 257–266. ACM, 2003.

[GT09] Hubert Garavel and Damien Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In Corina Pasareanu, editor, *Model Checking Software, 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.

[Hal10] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer-Verlag, Berlin, Heidelberg, 2010.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HB02] Nicolas Halbwachs and Siwar Baghdadi. Synchronous Modelling of Asynchronous Systems. In *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002*, pages 240–251, 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, sep 1991.

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous Observers and the Verification of Reactive Systems. In *Algebraic Methodology and Software Technology (AMAST '93), Proceedings of the Third International Conference on Methodology and Software Technology, University of Twente, Enschede, The Netherlands, 21-25 June, 1993*, pages 83–96, 1993.

[HM06] Nicolas Halbwachs and Louis Mandel. Simulation and Verification of Asynchronous Systems by means of a Synchronous Model. In *Sixth International Conference on Application of Concurrency to System Design (ACSD 2006), 28-30 June 2006, Turku, Finland*, pages 3–14, 2006.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[HP85] David Harel and Amir Pnueli. *Logics and Models of Concurrent Systems*, chapter On the Development of Reactive Systems, pages 477–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[HPR97] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of Real-Time Systems using Linear Relation Analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[HR99] Nicolas Halbwachs and Pascal Raymond. Validation of Synchronous Reactive Systems: From Formal Verification to Automatic Testing. In *Advances in Computing Science - ASIAN'99, 5th Asian Computing Science Conference, Phuket, Thailand, December 10-12, 1999*, pages 1–12, 1999.

[HS96] Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Oxford, UK, March 18-22, 1996*, pages 662–681, 1996.

[Hun93] Hardi Hungar. Combining Model Checking and Theorem Proving to Verify Parallel Processes. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993*, pages 154–165, 1993.

[ISO01] ISO/IEC. Enhancements to lotos (e-lotos). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.

[JLB+15] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A Formally-Verified C Static Analyzer. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 247–259, New York, NY, USA, 2015. ACM.

[JLM14a] Fatma Jebali, Frédéric Lang, and Radu Mateescu. GRL: A specification language for globally asynchronous locally synchronous systems. In *16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, November 3-5, 2014*, pages 219–234, 2014.

[JLM14b] Fatma Jebali, Frédéric Lang, and Radu Mateescu. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems (Syntax and Formal Semantics). Research Report RR-8527, INRIA, April 2014.

[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, April 2016.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[KM97] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma, editor, *Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), University of Twente, Enschede, The Netherlands*, volume 1217 of *Lecture Notes in Computer Science*, Berlin, April 1997. Springer. Extended version with proofs available as Research Report VERIMAG RR97-01.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *Software Engineering, IEEE Transactions on*, SE-3(2):125–143, March 1977.

[Lam80] Leslie Lamport. "Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 174–185, New York, NY, USA, 1980. ACM.

[Lan02] Frédéric Lang. Compositional Verification using SVL Scripts. 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 465–469. Springer, April 2002.

[Lan06] Frédéric Lang. Refined Interfaces for Compositional Verification. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Viguié Donzeau-Gouge, editors, *Proceedings of the 26th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'06), Paris, France*, volume 4229 of *Lecture Notes in Com-*

puter Science, pages 159–174. Springer, September 2006. Full version available as INRIA Research Report RR-5996.

[LcW06] Stefan Leue, Alin Ştefănescu, and Wei Wei. A Livelock Freedom Analysis for Infinite State Asynchronous Reactive Systems. In *Proceedings of the 17th International Conference on Concurrency Theory*, CONCUR'06, pages 79–94, Berlin, Heidelberg, 2006. Springer-Verlag.

[LDG+03] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.07: Documentation and user's manual. INRIA. Available at http://pauillac.inria.fr/ocaml/htmlman, 2003.

[LGGLBLM91] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming Real-Time Applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[LS99] Luciano Lavagno and Ellen Sentovich. ECL: A Specification Environment for System-Level Design. In *DAC*, pages 511–516, 1999.

[LT89] N Lynch and Mark Tuttle. An introduction to input/output automate. *CWI quarterly*, 2(3):219–246, 1989.

[Mar91] Florence Maraninchi. The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In *In IEEE Workshop on Visual Languages*, 1991.

[McG82] James R. McGraw. The VAL Language: Description and Analysis. *ACM Trans. Program. Lang. Syst.*, 4(1):44–82, January 1982.

[MGS12] Avinash Malik, Alain Girault, and Zoran Salcic. Formal semantics, compilation and execution of the GALS programming language DSystemJ. *Parallel and Distributed Systems*, 2012.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil83] Robin Milner. Calculi for Synchrony and Asynchrony. *Theor. Comput. Sci.*, 25:267–310, 1983.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MK06] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.

[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.

[Mon92] Ugo Montanari. True Concurrency: Theory and Practice. In *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June 29 - July 3, 1992*, pages 14–17, 1992.

[MRBS01] Hervé Marchand, Éric Rutten, Michel Le Borgne, and Mazen Samaan. Formal verification of programs specified with signal: application to a power transformer station controller. *Sci. Comput. Program.*, 41(1):85–104, 2001.

[MSRG10] A. Malik, Z. Salcic, P.S. Roop, and A. Girault. SystemJ: A GALS language for system level design. *Comput. Lang. Syst. Struct.*, 36(4):317–344, December 2010.

[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.

[MW14] R. Mateescu and A. Wijs. Property-dependent Reductions Adequate with Divergence-sensitive Branching Bisimilarity. *Sci. Comput. Program.*, 96(P3):354–376, December 2014.

[Par81] David Michael Ritchie Park. Concurrency and Automata on Infinite Sequences. In *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981*, pages 167–183, 1981.

[Pau89] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *J. Autom. Reasoning*, 5(3):363–397, 1989.

[PBCB06] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *FMSD*, 28(2):111–130, 2006.

[PBDSST09] D. Potop-Butucaru, R. De Simone, Y. Sorel, and J.-P. Talpin. From concurrent multi-clock programs to deterministic asynchronous implementations. In *ACSD '09*, pages 42–51. IEEE, July 2009.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, Germany, 1962.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[PMS15] Heejong P., Avinash M., and Zoran S. Compiling and Verifying SC-SystemJ Programs for Safety-critical Reactive Systems. *Comput. Lang. Syst. Struct.*, 44(PC):251–282, December 2015.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[Ram98] S Ramesh. Communicating reactive state machines: Design, model and implementation. In *IFAC Workshop on Distributed Computer Control Systems*, 1998.

[RdS90] Valérie Roy and Robert de Simone. Auto/Autograph. In *Computer-Aided Verification, Proceedings of a DIMACS Workshop 1990, New Brunswick, New Jersey, USA, June 18-21, 1990*, pages 477–492, 1990.

[RSD+04] S. Ramesh, Sampada Sonalkar, Vijay D'silva, Naveen Chandra R., and B. Vijayalakshmi. A Toolset for Modelling and Verification of GALS Systems. In Rajeev Alur and DoronA. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 506–509. Springer Berlin Heidelberg, 2004.

[Sme13] Gideon Smeding. *Verification of Weakly-Hard Requirements on Quasi-Synchronous Systems*. Theses, Université de Grenoble, December 2013.

[Sta96] E. S. S. Standard. EBNF: ISO/IEC 14977: 1996 (E). *URL http://www.cl. cam. ac. uk/mgk25/iso-14977. pdf*, 1996.

[Thi11] Damien Thivolle. *Modern languages for modeling and verifying asynchronous systems*. Theses, Université de Grenoble, April 2011.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[vGW89] Rob J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics (Extended Abstract). In *IFIP Congress*, pages 613–618, 1989.

[vGW96] Rob J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM*, 43(3):555–600, 1996.

[VW86] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)*. IEEE Computer Society, 1986.

[WA85] William W. Wadge and Edward A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.