Formal Modelling and Verification of GALS Systems Using GRL and CADP

Fatma Jebali, Frédéric Lang, Radu Mateescu

Under consideration for publication in Formal Aspects of Computing

Formal Modelling and Verification of GALS Systems Using GRL and CADP

Fatma Jebali, Frédéric Lang, and Radu Mateescu

Inria
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France

Abstract. A GALS (*Globally Asynchronous, Locally Synchronous*) system consists of several synchronous components that evolve concurrently and interact with each other asynchronously. The design of GALS systems is tedious and error-prone due to the high degree of synchronous and asynchronous concurrency present in complex architectures. In this paper, we present GRL (*GALS Representation Language*), a formal language designed to model GALS systems, for the purpose of formal verification of the asynchronous aspects. GRL combines the synchronous reactive model underlying dataflow languages and the asynchronous concurrent model underlying process algebras. We propose a translation from GRL to LNT, a value-passing concurrent language with classical process algebra flavour. This makes possible the analysis of GRL specifications using all the state-of-the-art simulation and verification functionalities provided by the CADP toolbox.

Keywords: GALS systems, asynchronous concurrency, formal description techniques, model-based verification, GRL, CADP

1. Introduction

A GALS (*Globally Asynchronous, Locally Synchronous*) system [Cha84] consists of several synchronous components that run concurrently and communicate altogether asynchronously. Each synchronous component consists of a deterministic and infinite loop, the iterations of which are cadenced by the component's own clock. The component is generally made of subcomponents composed synchronously, so that computations and data-flow communications within one iteration of the loop are assumed to be instantaneous.

Yet, in the general case, no *a priori* assumption can be made either on clock synchronization, relative clock paces, or asynchronous communication delays between components. Each GALS instance induces its

Correspondence and offprint requests to: Fatma Jebali, Inria Grenoble - Rhône-Alpes, Inovallée, 655, avenue de l'Europe, Montbonnot, F-38334 Saint Ismier Cedex, France. e-mail: fatma.jebali@inria.fr

own assumptions. In particular, although synchronous components have generally deterministic behaviour, asynchronous communication may introduce nondeterminism, thus complexity. Typical cases are unreliable communication media along which messages can be delayed, lost, duplicated, and/or reordered. Such inherent complexity entails a need to integrate formal verification in the design process of GALS systems. This helps to gain confidence in system correctness.

To formally model and verify GALS systems, synchronous languages and their dedicated verification frameworks have been intensively used. The reason is that the GALS paradigm takes its roots in 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 behaviour. However, some aspects related to asynchronous concurrency cannot be addressed in those frameworks.

Asynchronous languages and their dedicated verification frameworks have genuine benefits. First, they 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 of bisimulation relations w.r.t these operators. This allows efficient state-space reduction techniques (e.g., partial order reduction) and compositional verification to 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]).

Second, asynchronous verification frameworks 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. The approach promoted by most model checkers (e.g., [HLR93a], [Bou98]) for synchronous languages is *verification by observers*. Desired properties are expressed in terms of program invariants, also called *synchronous observers* [HLR93b]. Observers can be encoded directly in the synchronous language itself. This reduces the complexity of using full-fledged temporal logics for safety and liveness properties involving bounded future. However, general properties involving unbounded future cannot be expressed by means of observers and require more expressive formalisms, such as temporal logics.

However, asynchronous concurrent languages and temporal logics require a substantial learning effort, which narrows down their practical usability, in the context of GALS systems. We propose to alleviate the use of verification tools for asynchronous systems in the framework of GALS systems by using a DSL (*Domain Specific Language*) [vDKV00], which serves as intermediate form between collections of synchronous components composed asynchronously, and purely asynchronous concurrent languages. This DSL should provide a clear distinction between synchronous components, which can possibly be obtained from translation of existing synchronous languages, and the additional components used to define their asynchronous interaction, such as communication media. We consider such a distinction as necessary to enable the combined use of synchronous verification frameworks for synchronous components and asynchronous verification frameworks for their asynchronous composition. As regards the synchronous components, the language can therefore be seen as a (minimal) language intended to serve as target of back-end compilers for synchronous languages. An important factor of practical usability of the DSL is its ability to describe relevant aspects of GALS behaviours in a concise, natural, and integrated manner, so that it can be easily learnt and mastered by users.

In this paper, we propose GRL (*GALS Representation Language*) [JLM14a] as such a DSL for GALS systems. GRL is a rich description language that combines features of synchronous programming (determinism, atomicity) and process algebra (nondeterministic communication, asynchronous concurrency) in one unified language, while keeping homogeneous syntax and semantics.

GRL enables modular specification of synchronous programs (named *blocks*), asynchronous communication media (named *mediums*), and environmental constraints (named *environments*). Each block is a deterministic code fragment, built by synchronous composition of subblocks and standard algorithmic statements, which defines an iteration of the synchronous component loop. Each execution of this code fragment is called a *step*. GRL abstracts away the notion of component clock, so that blocks can perform their own steps at arbitrary instants by default. Environments serve to set constraints either on block inputs (e.g., to restrict the range of inputs that a block can read) or on block *activations* (permissions acquired by blocks to perform their steps). GRL is intended to be sufficiently expressive to enable modelling of various activation and communication policies, so that it can be used to address a large spectrum of GALS systems. In addition, GRL has formal semantics, which enables rigorous analysis of GRL descriptions.

To analyse GRL specifications, we take advantage of the CADP verification toolbox for concurrent asynchronous processes [GLMS13]. CADP provides various tools for interactive simulation, test case generation,

verification of temporal logic properties (model checking), equivalence checking, etc. The connection from GRL to CADP is done by translation from GRL to an input specification language of CADP named LNT [CCG+14], which combines the best of process algebraic languages and imperative/functional programming languages. This translation is fully automated by a tool named GRL2LNT[1]. This paper is an extended version of [JLM14a]. Our contribution can be summarized as follows:

- We give a detailed description of the GRL language and a subset of the formal semantic rules, which we illustrate with well-chosen examples, most of which are inspired from an FCS (*Flight Control System*). This description is more detailed than in [JLM14a] but due to space limitation, we cannot describe all the language details. The complete syntax and semantics of GRL can be found in an 82-page technical report available online [JLM14b].
  Compared to the GRL version presented in [JLM14a] and [JLM14b], we have revised and enhanced the syntax of the language. Enhancements encompass some keyword changes and the addition of a new construct, named *activation signals*, whose role is to control block activations.
- We propose a translation scheme between GRL and LNT, also using illustrative examples. In particular, we show how synchronous components with internal memory can be translated into LNT functions (which are stateless), and how the components of a GALS system translate into asynchronous concurrent processes. This contribution is new w.r.t. [JLM14a], which only addresses the GRL language.
- We illustrate some of the verifications that can be done on a GALS system modeled in GRL using GRL2LNT and CADP. In particular, we focus here on deadlock checking and on the verification of more involved properties using temporal logic formulas. Here again, verification was not addressed in [JLM14a].

Overview. Section 2 highlights some related work. Section 3 presents a simple FCS as a GALS instance. Section 4 presents the main features of the GRL language. Section 5 outlines the LNT language and the CADP toolbox. Section 6 describes the translation from GRL to LNT. Section 7 presents some verification scenarios for GALS systems, illustrated on the FCS. Finally, Section 8 summarizes the paper and indicates directions for future work.

2. Related work

In this section, we review the approaches addressing GALS systems in synchronous frameworks, then those addressing GALS systems in asynchronous frameworks.

GALS in synchronous frameworks. The intent of modelling GALS systems by using synchronous languages can be traced back to the early eighties, when Milner stated that asynchrony can be expressed in synchronous formalisms [Mil83]. Accordingly, several approaches emulated asynchrony by means of sporadic activation of synchronous components and external non-determinism (e.g., additional inputs, to which arbitrary values are assigned) [HB02, HM06, GG03, LGTLL03].

This gave birth to the multi-clock model (also called *polychrony*), which has proven adequate to compose several components whose clocks are loosely coupled [GG10, LGTLL03, GG07]. Multiclock Esterel [BS01], CRP (*Communicating Reactive Processes*) [BRS93] and CRSM (*Communicating Reactive State Machines*) [Ram98] (a visual language built upon CRP) are extensions of the Esterel language with CSP-like primitives to accommodate the multi-clock model. The Signal language [LGTLL03] allows the description of components, whose clocks are a priori unrelated while computations and communications are assumed to be bounded.

Further works have investigated how to synthesize semantic-preserving GALS systems from synchronous programs, foreseeing their distribution [BCLG99, PBCB06, PBDSST09, BBS12]. This approach (called *desynchronization*) favored correct-by-construction deployment of synchronous programs over GALS architectures. Several theoretical results on this concern are already supported by the Signal compiler. Our approach is different in the sense that GRL addresses directly the desynchronized system, so it can be used as back-end of the generated code.

All the aforementioned approaches address deterministic GALS instances in which communication media

[1] The GRL2LNT translator has been developed in the framework of an industrial project, named *Bluesky*, of the Minalogic French competitivity pole (`www.minalogic.com`). The Bluesky project addresses the verification of networks of PLCs (*Programmable Logic Controllers*).

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. In addition, modelling GALS systems in synchronous languages requires real-time guarantees on both the relative paces of synchronous components and communication delays. Such guarantees may be unknown in the general case, or at least difficult to synthesize in some distributed applications. Examples of this kind are the 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 specific to synchronous languages do not support logics with sufficient expressiveness to capture general liveness and fairness properties.

To address more general GALS systems, whose synchronous components evolve at different paces and communicate along unreliable media with no real-time guarantees, asynchronous verification frameworks are then required.

GALS in asynchronous frameworks. Several works have considered the modelling of GALS systems in asynchronous concurrent languages. A first approach consists in translating a GALS-specific language into a process language.

In [RSD+04], CRSM is translated into Promela, the input language of the SPIN model checker [Hol97]. Then, verification is achieved by means of distributed observers, to circumvent the complexity of using temporal logics. The reliance of CRSM on Esterel entails a lack of data-driven support in the language.

SystemJ [MSRG10] extends Java with Esterel-like synchronous model and CSP-like asynchronous model. Then, 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. However, nondeterminism is still difficult to verify. Efficient code can be automatically generated from SystemJ programs. The language is still unsuitable for systems with limited resources due to its reliance on Java virtual machines as target. Recently in [PMS15], a translation has been defined from a subset of SystemJ to LTL (*Linear Time Logic*) formulas, from which networks of Mealy automata are synthesized and translated into Promela, thus making possible the verification using Spin.

Both CRSM and SystemJ are design-oriented languages which are built upon the Esterel synchronous semantics. Contrarily, GRL has a minimal set of synchronous operators, which are enough to be used as (1) back-end of synchronous languages and (2) front-end of verification tools for asynchronous systems. Another key difference distinguishing GRL is its support of explicit nondeterministic statements, permitted only inside asynchronous components (environments and mediums). This enhances the expressiveness of GRL to handle a wide range of GALS instances. Moreover, nondeterminism induced by the asynchronous composition of synchronous components (blocks) can be controlled thanks to activation signals and verified by using CADP.

Another approach consists in combining synchronous languages and asynchronous process languages. Locally synchronous components are encapsulated in asynchronous processes (called *wrappers*) to interface with other components. Globally 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 iteration, all possible values of inputs are generated; then, the C program is invoked together with constraints on the component activation; finally, 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 formulas.

There are several differences between the Promela approach and ours. We use a single language to describe both synchronous and asynchronous components. Blocks in GRL are atomic by construction. Constraints on block inputs can be set by using environments, which lead to reduced state spaces. No equivalent to GRL environments has been mentioned in [DMK+06]. Our approach is more modular in the sense that several GRL environments can be composed with GRL blocks, either to control their activation or to constrain their inputs. GRL does not fix any communication protocol and is expressive enough to model general GALS systems, involving nondeterminism and arbitrary asynchrony between synchronous components. Finally, our verification is based on branching time temporal logic, adequate with bisimulation relations and compositional verification.

In the same vein as the Signal-Promela approach [DMK+06] is [GT09], in which SAM automata (extended Mealy machines) are translated into LNT functions, encapsulated into LNT wrapper processes. The execution

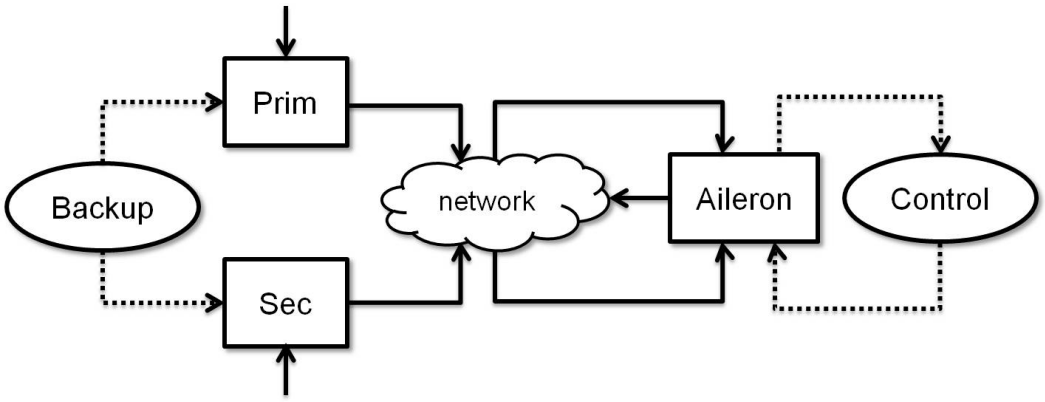


Fig. 1. Architecture of the FCS

of those processes is not atomic, contrarily to GRL blocks. Thus, inputs and outputs of different LNT wrappers can interleave arbitrarily. The asynchronous composition between processes is completely arbitrary, since no constraints are made on their execution paces. As such, the maximal degree of asynchrony is considered. Abstractions and compositional verification, available in CADP, are used to cope with state space explosion. Verification by model checking and performance evaluation are applied by using CADP.

The SAM-LNT approach addresses a specific GALS instance (a ground-plane communication protocol). Contrarily, GRL is intended to address more general GALS instances. To this aim, it provides activation constraints to fine tune the degree of asynchrony between concurrent blocks.

3. Running Example

Throughout this paper, we consider as example a part of an FCS, depicted in Figure 1, whose role is to control aircraft turning. The FCS is modeled at a high level of abstraction as a composition of:

- The controller of an aileron (a flap attached to the end of a wing), named *Aileron*, allowing to adjust the aircraft's flight turning.
- Two Fly-By-Wire computers, named *Prim* (for *primary*) and *Sec* (for *secondary*), commanding the movement of the aileron. They are implemented as redundant components, *Sec* being used as a backup of *Prim*, which provides the system with a level of fault tolerance.
- A Flight Control Data Concentrator, named *Backup*, scheduling the execution of *Prim* and *Sec*.
- A movement controller, named *Control*, ensuring a smooth movement of *Aileron*.

The behaviour of the system can be summarized as follows. *Prim* receives an order from the pilot cockpit; it determines whether the aileron should move up, move down, or not move; then it sends the decision to the aileron over the network. On the other side, *Aileron* receives the required movement from the network and a safety condition (determining whether it can perform the movement) from *Control*; it computes a new position, which depends on both the current position of the aileron and the required movement; then, it sends the position to *Control* and an acknowledgement to the network. Once a failure is detected in the behaviour of *Prim*, it is considered out of order and the control of *Aileron* is given to *Sec*.

Note that FCS components have been deeply studied in several works (see e.g., [MWO$^+$05, BÖM14]). We do not intend here to achieve yet another case study on FCS, but rather use this example to illustrate the main principles of the GRL approach.

4. GRL (GALS Representation Language)

In this section, we present GRL, whose syntax and semantics are fully described in the technical report [JLM14b]. Syntax is given in EBNF (*Extended Backus-Naur Form*), i.e., as a set of rules called *productions*. Each production has the form "$\chi ::= \xi$", where χ is a non-terminal symbol defined by the meta-expression ξ. ξ consists of non-terminal symbols and terminal symbols composed using the following meta-operators: concatenation (denoted by juxtaposition), alternative "|", bracketing "( )", option "[ ]", and repetition "...".

Non-terminal symbols and generic terminal symbols are written in italics. Their occurrences can be distinguished using subscripts. Terminal symbols are either keywords written in bold font or key symbols written in teletype font. Note therefore that "`[]`", "`()`", and "`|`" denote terminal symbols distinct from meta-operators "[]", "()", and "|".

GRL syntax is fully presented in Figures 3 (page 8) to 6 (page 16). Generic terminal symbols f, T, K, F, X, B, N, M, and S denote identifiers for, respectively, *record fields*, *types*, *literal constants*, *functions*, *variables*, *blocks*, *environments*, *mediums*, and *systems*. Non-terminals E and I denote respectively expressions and statements.

4.1. Overview

GRL is an imperative programming language with functional flavour. A GRL specification can be structured in several *modules*. Modules can import other modules, which allows single monolithic specifications to be split into reusable pieces of manageable size. Modules contain the following constructs:

1. *types*, which can be either predefined (such as Booleans and naturals) or defined by the user (such as arrays and records),
2. *named constants* of any type,
3. *blocks*, representing the synchronous components,
4. *mediums*, representing the asynchronous communication mediums,
5. *environments*, representing constraints of the external environment on blocks, and
6. *systems*, representing the composition and interaction of blocks, environments, and mediums.

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

Synchronous blocks. GRL is not intended to include a full-fledged synchronous language. Rather, it serves as an intermediate format mapping synchronous languages to verification tools dedicated to asynchronous systems. However, GRL is sufficiently rich to make possible the translation of synchronous programming constructs with reasonable effort.

The synchronous part of GRL provides a built-in definition of the notions of deterministic infinite loop and internal state. Synchronous parallelism and causality have to be resolved beforehand (e.g., by the compilers of synchronous languages), since synchronous composition in GRL is sequential. Time and clocks are not explicitly represented and are abstracted out.

An individual block performs a (potentially unbounded) sequence of discrete deterministic steps (also called *reactions*), and maintains an internal state (also called *memory*, represented by state variables). Each step consists in first reading inputs; then computing outputs and next internal state, both depending on inputs and the current internal state. These activities are performed simultaneously making the step atomic, as assumed in synchronous programming [Hal13].

Blocks can be composed synchronously to interact with each other inside higher-level blocks, in a modular way. This enables a textual description of hierarchical block compositions, as illustrated in Figure 2. Lower-level blocks are called *subblocks* and a block that is not a subblock of another block is called a *highest-level block*. Composition between subblocks is carried out by connecting inputs of some subblocks to outputs of preceding subblocks. Interactions between subblocks occur simultaneously, as assumed in synchronous programming [Hal13]. Accordingly, outputs produced by a subblock are consumed by other subblocks in the same step as the enclosing block. This way, data is processed along causal dependencies between subblocks, making the behaviour of blocks deterministic.

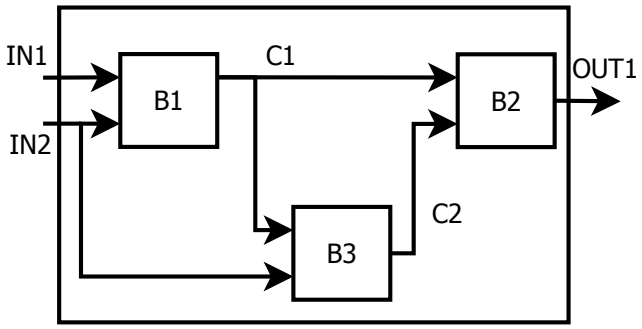


Fig. 2. Pictorial description of subblock composition

Asynchronous composition of blocks. Inside systems, highest-level blocks are composed in asynchronous concurrency, following the interleaving semantics underpinning process algebras [Mil89]. Atomic deterministic steps of concurrent blocks interleave arbitrarily, without causal dependency. By default, two consecutive steps of a block may occur arbitrarily far from each other, unless specified differently by using activation constraints. This abstraction makes GRL expressive enough to model and reason about general GALS systems.

Blocks can be connected to environments and mediums, but not to each other. Connections are made by means of *channels*, which are tuples of variables. Channels are unidirectional, i.e. they are used by a component either only for *reception* or only for *emission* of tuples of values. The underlying interaction model is message-passing rendezvous. Blocks are *active* components inside systems, i.e., they initiate interactions with other components. In this respect, environments and mediums are *passive* components, i.e., they only respond to block interactions.

Environments and mediums can be defined by the user, similarly to blocks. In addition, their behaviour may exhibit nondeterminism, a key feature for asynchronous system modelling and compositional specification. This provides descriptions with accuracy and high abstraction capability.

Environments enable constraints on block behaviours to be expressed at different levels of abstraction. 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.

Additionally, environments can adjust the degree of asynchrony in block composition by setting constraints on block activations. This allows to master the possible interleavings between blocks, e.g., a block cannot execute indefinitely to the detriment of the others. Activation policies can model, at a suitable level of abstraction, realistic situations such as constraints on the relative paces between synchronous components modeled as highest-level blocks.

Mediums enable blocks to communicate asynchronously. They receive messages from or send messages to blocks whenever requested. Messages can be stored in the internal state of the medium, thus enabling message buffering. In addition, nondeterminism allows behaviours such as message loss, duplication, or reordering to be described naturally. 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.

Composing a system from blocks, environments, and mediums provides the user with comfort and insight to fine tune GALS behaviour. With such a composition, we seek smooth and tight tailoring of complex network topologies, environment requirements and constraints, as well as communication protocols.

```
block  ::=  block B {vars_c}
              (in vars_i0 , ... , in vars_in , out vars_o0 , ... , out vars_on )
              [receive vars_r0 , ... , receive vars_rn , send vars_s0 , ... , send vars_sn ] is
               alias B_0 {args_0} as B'_0 , ... , B_m {args_m} as B'_m
               static var vars_p0 , ... ,static var vars_pn ,
               var vars_t0 , ... ,var vars_tn
                 I
            end block
       |    block B {vars_c}
              (in vars_i0 , ... , in vars_in , out vars_o0 , ... , out vars_on ) is
               !c string  |  !lnt string
            end block

vars   ::=  X_0 : T_0 [:= E_0] , ... , X_n : T_n [:= E_n]
arg    ::=  E | _ | ?X | ?_ | any T
args   ::=  arg_0 , ... , arg_n

I      ::=  null
       |    X := E
       |    X [E_0] := E_1
       |    X . f := E
       |    I_0 ; I_1
       |    if E_0 then 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
       |    for I_0 while E by I_1 loop I_2 end loop
       |    case E is K_0 -> I_0 | ... | K_n -> I_n | [any -> I_n+1] end case
       |    Bi(args)
            /* Statements forbidden in blocks and reserved to environments and mediums */
       |    enable B
       |    when <X_0 , ... , X_n> -> I_0
       |    when ?<X_0 , ... , X_n> -> I_0
       |    X := any T [where E]
       |    select I_0 [] ... [] I_n end select

E      ::=  K | X | F(E_1 , ... , E_n)
```

Fig. 3. The syntax of GRL blocks

4.2. Types

GRL provides a rich data structure. Predefined types consist of Boolean, naturals and integers which are represented on either 8 bits (**nat**, **int**), 16 bits (**nat16**, **int16**), or 32 bits (**nat32**, **int32**), and strings. Types can also be defined by the user. User-defined types consist of array, enumeration, record, and range types.

4.3. Blocks

Blocks are formally defined by the non-terminal *block* in Figure 3. A block specification consists of formal parameters, variables, subblock *instantiation*, and a statement I defining the block behaviour. This statement must be deterministic, i.e., use only the constructs described in the first 10 lines of the production defining I in Figure 3. It consists of subblock *invocations* combined with standard algorithmic control structures such as assignment, sequential composition, conditionals ("**if**" and "**case**") and loops ("**while**" and "**for**").

Formal parameters are declared with types and possibly default values. They are classified into *constant*, *input*, *output*, *receive*, and *send* parameters. Constant parameters, enclosed into braces, denote configuration data. Input and output parameters, preceded by keywords **in** and **out**, enable synchronous interaction either with other blocks (for subblocks) or with the environment (for highest-level blocks). Receive and send parameters, preceded by keywords **receive** and **send**, enable asynchronous interaction with other blocks along mediums. Blocks defined with receive and send parameters are necessarily highest-level blocks, thus cannot be used as subblocks inside other blocks.

Variables are either *temporary* or *static*. Temporary variables are preceded by the keyword **var** and are optionally initialized at declaration time. Once a step starts, each temporary variable is first assigned its

initialization value (if any), then used in computations within the step. Its updated value is lost at the end of the step, i.e, when returning from the block.

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

Example 4.1. The *Aileron* component of the FCS system, introduced in Section 3, is modeled in GRL as follows. Value of variable "pre_pos" read in lines 8 and 10 is the one computed in the previous step (line 17). In the first step of the block, the initialization value (line 3) is used.

```
1  block Aileron {delta:nat := 1}(in safe:bool, out pos:nat)
2                [receive move:move_type, send ok :bool] is
3    static var pre_pos:nat := 2 -- static variable
4    var new_pos:nat := 2 -- temporary variable
5    -- block behaviour
6    if (safe) then
7      if ((move == up) and (pre_pos < 4)) then
8        new_pos := pre_pos + delta
9      elsif ((move == down) and (pre_pos > 0)) then
10       new_pos := pre_pos - delta
11     end if
12   end if;
13   -- write outputs
14   pos := new_pos;
15   ok := safe;
16   -- update the internal state
17   pre_pos := new_pos
18 end block
```

A block can encapsulate subblock instances. Each subblock instance has a separate internal state. Instances can be *aliased*, i.e., assigned different names, by using the keyword **alias**. If a subblock has formal constant parameters, the corresponding actual parameters of the instance must be set at aliasing time. The underscore "_" can be used as actual constant parameter. It indicates that the instance should use the default value of the corresponding constant parameter in the block definition.

Example 4.2. In the code below, *Ail1* is an instance of *Aileron* (Ex. 4.1) with parameter *delta* set to 1, which is the default value associated to the formal parameter in the definition of *Aileron*. *Ail2* is another instance of *Aileron* with parameter *delta* set to 2.

```
1  alias Aileron {_} as Ail1, Aileron {2} as Ail2
```

Subblocks can be invoked inside the current block with actual input and output parameters. Actual output parameters are distinguished by a question mark. This means that the parameter will have a value assigned when returning from the block. Underscores are used to specify *unconnected* parameters. For each unconnected input parameter, the default value of the corresponding formal parameter in the block definition is used in each cycle. For each unconnected output parameter, the value assigned to the corresponding formal parameter when returning from the block is just ignored.

GRL does not provide any synchronous parallel composition operator. The synchronous composition of subblocks is therefore sequential. This requires a topological order between subblock invocations to be defined. If the GRL code was generated from a synchronous language, one would expect a correct order to be produced from the front-end compiler, automatically. This is reasonable since every synchronous language should be equiped with a compiler that determines such an order.

Example 4.3. The following GRL code corresponds to the block composition depicted in Figure 2. In accordance with the dependency between input-output links induced by the structure of *B0*, subblocks *B1*, *B3*, and *B2* should be invoked in this specific order. Subblocks are connected with each other using variables *c1* and *c2*. For example, the value of "?*c1*", an actual output of *B1*, is broadcasted to the subsequent subblocks *B2* and *B3*. Actual parameters *in1*, *in2*, and *out1* are declared as formal parameters of the enclosing block *B0*, since they are themselves inputs/outputs of *B0*. The other block parameters *c1* and *c2* are declared as temporary variables since they are input/output links, internal to *B0*.

```
1  block B0 (in in1, in2: nat, out out1: nat) is
2     alias B1 as B1, B as B2, B3
3     var c1, c2: nat -- input/output links
4        B1 (in1, in2, ?c1);
5        B3 (c1, in2, ?c2);
6        B2 (c1, c2, ?out1)
7  end block
```

Note that block $B0$ could be defined without aliasing as follows:

```
1  block B0 (in in1, in2: nat, out out1: nat) is
2     var c1, c2: nat -- input/output links
3        B1 (in1, in2, ?c1);
4        B  (c1, in2, ?c2);
5        B  (c1, c2, ?out1)
6  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 can be declared using pragmas "**!c**" and "**!lnt**", respectively.

Example 4.4. The C function *Shift* below applies a left-shift operation on a natural number. Type *GRL_NAT* is used in the function interface (line 7). Before using a parameter, it should be converted to the C domain by using the predefined function *GRL_NAT_TO_UNSIGNED_CHAR* (lines 8-9). Then, before returning from the function, the result is converted to the GRL domain by using the predefined function *GRL_UNSIGNED_CHAR_TO_NAT* (line 10).

The function is written in a file with the same prefix as the GRL file and extension ".c", which is imported in the current GRL module. So doing, it can be encapsulated inside block *C_Shift* (line 3).

```
1  -- GRL file importing the C file
2  block C_Shift (in numb:nat, in bits:nat, out result:nat) is
3     !c "Shift"
4  end block
5
6  -- C file
7  void Shift (GRL_NAT number, GRL_NAT bits, GRL_NAT * result){
8     unsigned char arg_number = GRL_NAT_TO_UNSIGNED_CHAR (number);
9     unsigned char arg_bits = GRL_NAT_TO_UNSIGNED_CHAR (bits);
10    * result = GRL_UNSIGNED_CHAR_TO_NAT (arg_number << arg_bits);
11 }
```

Although including external code enhances user convenience, 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 in different contexts should return the same result. In particular, blocks defined with external code have no static variables. 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.

Remarks. GRL blocks are synchronous reactive components with imperative-style. GRL blocks are somehow similar to Statecharts/Stateflow, since both languages define Mealy machines and explicitly represent state variables. Composition between GRL blocks is sequential. The absence of parallel and explicit delay operators in the synchronous model of GRL deserves a few comments.

- Our aim is to keep a minimal number of core constructs, but enough to use GRL as target language for synchronous programming. On the one hand, GRL can be used as back-end of synchronous language compilers, then it can benefit from optimized sequential code they generate. On the other hand, GRL can be smoothly integrated as back-end of various synchronous compilers, without interfering with the specific nonclassical algorithms they implement, such as causality analysis algorithms (e.g., conditional dependence graph in Signal, and computation of fixpoints in Esterel).
- Delay operators can be encoded in GRL by using static variables, with no difficulty.

  Example 4.5. The Lustre node *Counter*, defined on the left below, can be implemented by block *Counter*, defined on the right below. Variable *preN* implements the Lustre "pre (N)" expression, by storing the value to be used in the next step. The Boolean variable *first* implements the Lustre operator "→".

```
blocks  ::=  B_0 , ... , B_n
env     ::=  environment N {vars_c}
                        (in vars_{i_0} , ... , in vars_{i_n} , out vars_{o_0} , ... , out vars_{o_n} ,
                        block blocks_{b_0} , ... , block blocks_{b_n} ) is
               alias B_0 {args_0} as B'_0 , ... , B_n {args_n} as B'_n
               static var vars_{p_0} , ... , static var vars_{p_n} ,
               var vars_{t_0} , ... , var vars_{t_n}
                 I
               end environment
```

Fig. 4. The syntax of GRL environments

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

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

- GRL does not have to deal with causality problems owing to the absence of the synchronous parallel operator. GRL semantics ensure that: (1) variables are necessarily assigned values before being read and (2) static variables are necessarily initialized at declaration time. For instance, GRL semantics forbid the following program because variable x is used uninitialized 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:

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

```
1  -- program 2
2  var x: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)
```

Another difference with standard synchronous languages is the presence of explicit computation loops in GRL. For and while loops are useful if used as iterators. However, it is on the user's charge to ensure their boundedness, since GRL provides no static analysis in this concern. Computation loops can be combined with subblock invocations, in which case subblock invocations behave as procedure call.

4.4. Environments

Environments provide block inputs and react to block outputs, thus putting constraints on the data that blocks carry. Additionally, they define constraints on block activation, thus enabling to define execution policies.

Environments are formally defined by the non-terminal *env* in Figure 4. An environment specification consists of formal parameters encompassing constant, input, and output parameters; *activation* parameters, prefixed by the keyword **block** and denoting block identifiers; static and temporary variables; instantiations of subblocks used as routines; and a statement I defining the environment behaviour. This statement can be nondeterministic, i.e., use all the constructs of the production defining I in Figure 3. The same deterministic statements as blocks can be used, extended with nondeterministic assignment (line 14), nondeterministic choice (line 15), and *signals* (lines 11-13).

Data constraints. An environment interacts with a block either by reception (input parameters) or by emission (output parameters) of tuples of values, the parameters used in the same interaction being grouped in *channels*. Since environments are passive components, which may interact with several blocks, an environment

can be executed at several moments. Each of those executions is devoted to interaction on a given channel. The code to be executed by an environment depends on the channel on which the block-environment interaction occurs. GRL associates to each channel a *data signal* (introduced by keyword **when**) to guard the code part to be executed on interactions on that channel.

Example 4.6. The *Control* component of the FCS system is modeled in GRL as follows. It checks whether the current position of *Aileron* is still within a predefined interval (subblock *Interval*) and outputs a safety state. Details of the code will be explained in the sequel.

```
1  environment Control (in pos:nat, out safe:bool) is
2    alias Interval {5} as Interval
3    static var lastPos:nat := 0
4    select
5        when ?<pos> -> lastPos := pos        -- read ''pos''
6    [] when <safe> -> Interval (lastPos, ?safe) -- write ''safe''
7    end select
8  end environment
9
10 block Interval {thres:nat} (in pos:nat := 0, out ok :bool) is
11   var sup, inf : bool
12   sup := (thres < pos) and ((pos - thres) < 5);
13   inf := (pos <= thres) and ((thres - pos) < 5);
14   if (sup or inf) then
15     ok := true
16   else
17     ok := false
18   end if
19 end block
```

The signal associated to input channel "**in** $X_0 : T_0 , \ldots , X_n : T_n$" has the form "**when ?<**$X_0 , \ldots , X_n$**> ->** I_0" (Example 4.6, lines 1, 5). 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 4.6, the signal "**when ?<***pos*> $\rightarrow$" defined at line 5 is active each time the block connected to channel "**in** *pos*" outputs a value. In this case, value of *pos* is read and assigned to "*lastPos*", when returning from the environment. Signal "**when <***safe*> $\rightarrow$" defined at line 6 is not active during the execution.

The signal associated to each output channel "**out** $Y_0 , \ldots , Y_m$" has the form "**when <**$Y_0 , \ldots , Y_m$**> ->** I_0" (Example 4.6, lines 1, 6). 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 the statement I_0 to assign values to those variables, which are emitted on the channel. In Example 4.6, the signal "**when <***safe*> $\rightarrow$" defined at line 6 is active each time the block connected to channel "**out** *safe*" starts a step. In this case, output *safe* is assigned a value when returning from the environment. Signal "**when ?<***pos*> $\rightarrow$" defined at line 5 is not active during the execution. Angle brackets are optional if a signal contains a single variable.

Since interactions on a channel occur whenever requested by the connected block, there must be at least one reachable execution path, 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 4.6 (lines 4-7).

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, at most one signal is present in each execution path. Note, however, that the code associated to a given signal does not have to be 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 activations. This is illustrated in Example 4.6. The value emitted to a block on channel *safe* (line 6) depends on the last value carried by a block output received on channel *pos* (line 5). This information is stored in variable *lastPos* (line 3).

It is worth noticing that 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. In general, such constraints allow to fine tune the description of the system, and reduce the size of generated state spaces.

Activation constraints. The activation of blocks whose identifiers occur as activation parameters is intended to be controlled 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". That signal implements the permission for a block (named B) to start a step. A block whose identifier occurs as activation parameter of the current 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.

Example 4.7. Consider a system composed of highest-level blocks $B_1, \ldots, B_n$. 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 B_1, ..., block B_n) is
2     select
3       enable B_1 [] ... [] enable B_n
4     end select
5  end environment
```

Example 4.8. Environment *quasi_synchrony* implements the *quasi-synchrony* hypothesis [CMP01], stating that the connected blocks evolve at the same pace, although they can slightly drift. This means that between each two consecutive executions of a block, the other one can execute at most twice. This illustrates how to express relations between paces of different blocks.

```
1  environment quasi_synchrony (block B1, block B2) is
2     static var ok1, ok2 : bool := true -- B1 and B2 can execute
3     select
4       -- execute B1 once
5       if (ok1) then
6         enable B1; ok1 := not (ok2); ok2 := true
7       end if
8     []
9       -- execute B2 once
10      if (ok2) then
11        enable B2; ok2 := not (ok1); ok1 := true
12      end if
13    end select
14 end environment
```

Example 4.9. Environment *Backup* implements the activation policy for the FCS system composed of redundant blocks *Prim* and *Sec*, the latter being the backup of the former. This illustrates how to express constraints on the start off (initial delay before a block starts its first step) and crash of blocks.

```
1  environment Backup (block Prim, block Sec) is
2     static var p_alive, s_alive : bool := true
3        if (p_alive) then -- Prim executes
4           p_alive := any bool;
5           enable Prim
6        elsif (s_alive) then -- Sec executes
7           s_alive := any bool;
8           enable Sec
9        end if
10 end environment
```

In both examples, 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 discrete model. GRL allows to implement arbitrarily complex activation policies: regular or sporadic executions, priorities, or arbitrary relations between the paces of synchronous components.

Remark. The previous version of GRL provided no built-in mechanism to control block activations. If controlling block activation were necessary, the trick was to introduce additional data signals whose variables

```
med ::= medium M {vars_c}
          [receive vars_r0 , ... , receive vars_rn , send vars_s0 , ... , send vars_sn ] is
        alias B_0 {args_0} as B'_0 , ... , B_n {args_n} as B'_n
        static var vars_p0 , ... , static var vars_pn ,
        var vars_t0 , ... , var vars_tn
          I
        end medium
```

Fig. 5. The syntax of GRL mediums

implemented the permission of the block to perform a step. Then, the reachability of the introduced signals determined whether a block could perform a step. This solution led to potential confusion since the same construct (data signals) was used to express constraints having different levels of abstraction.

On the contrary, activation signals are syntactically and semantically more elegant. They allow a clear separation between data- and activation-oriented constraints, which makes the user intention clearer. Data signals allow to control data carried by block inputs and outputs, and cannot handle block activations. Thus, a data signal must be reachable whenever required by a connected block, which is now required using static semantics constraints. Activation signals allow to control the activation of blocks at specific moments in time, and cannot handle input and output data. Thus, the reachability of activation signals induces the activation policy of blocks. Moreover, both types of signals can be combined to describe complex situations. In particular, the description of scenarios, which was cumbersome and error-prone previously, is drastically simplified in the current version of GRL.

4.5. Mediums

Mediums are intended to model asynchronous communication between highest-level blocks. They are formally defined by the non-terminal *med* in Figure 5. Medium specification is described similarly to environments, except that input and output channels are replaced by receive and send channels, and activation parameters are not allowed. A medium behaviour is defined by a nondeterministic statement, in which activation signals are not allowed.

A medium interacts with highest-level blocks either by reception (receive channel) or by emission (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 with both blocks on separate channels. Data signals are associated to those channels, as explained in Section 4.4. Nondeterministic choice is appropriate for combining data signals and static variables are useful for message buffering.

Example 4.10. The following code implements a unidirectional one-place buffer of natural numbers. Channel *R_M* is devoted to receive messages, which are buffered using a static variable, waiting to be emitted on channel *S_M*. This model is used in *Loosely Time-Triggered Architectures* [BBC10], in which mediums behave as shared memories between writers and readers. Communication (called *communication by sampling*) is non blocking.

```
1  medium ComBySampling [receive R_M:nat, send S_M:nat] is
2     static var Buf_M:nat := 0
3        select
4              when ?<R_M> -> Buf_M := R_M -- reception of M
5           [] when <S_M> -> S_M := Buf_M -- emission of M
6        end select
7  end medium
```

Example 4.11. The following code implements a FIFO queue. The queue is encoded by using a static variable (line 8) of type *queue* which is an array of messages (lines 1-3). 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  type queue is
2       array [0 .. size_queue] of message
```

```
 3  end type
 4
 5  medium FIFO [receive rec_msg : message , send snd_msg : message] is
 6    static var queue : queue := queue (none)
 7      select
 8        when ?<rec_msg> -> enqueue (rec_msg, queue, ?queue)
 9      [] when <snd_msg> -> dequeue (queue, ?queue, ?snd_msg)
10      end select
11  end medium
```

4.6. Systems

Systems are formally defined by non-terminal *system* in Figure 6. A system specification consists of formal parameters, temporary variables, component instantiations, and a behaviour described as a composition of components.

The set of highest-level block instances, composed asynchronously in the system, is introduced by keyword **block list**. Actual parameters have the same forms as those in subblock invocation inside components. Additional parameters, called *wildcards*, of the form "**any** T" can be used, matching any value of type T. They are semantically equivalent to actual parameters, declared as temporary variables, and not used for interactions with other components. Actual parameters are grouped to compose channels. Parameters of the same channel are enclosed in angle brackets, which are optional if the channel has a single element. In each channel, parameters should have the same form, i.e., all of them 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 "?<_, … , _>").

Example 4.12. The following code implements the whole FCS system.

```
 1  system Main (order1, order2:move_type, safe:bool, pos:nat) is
 2    alias Computer as Prim, Computer as Sec,
 3          Aileron {_} as Aileron,
 4          Backup as Backup, Control as Control,
 5          Buffer as Buffer
 6
 7    var s_move1, s_move2, r_move: move_type,
 8        s_ok, r_ok1, r_ok2       : bool
 9
10      block list
11        Prim    (order1)[r_ok1, ?s_move1],
12        Sec     (order2)[r_ok2, ?s_move2],
13        Aileron (safe, pos)[r_move, ?s_ok]
14      environment list
15        Backup (Prim, Sec),
16        Control (pos, ?safe)
17      medium list
18        Buffer [s_move1, s_move2, ?r_move, ?r_ok1, ?r_ok2, s_ok]
19  end system
```

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 the associated signal in the component definition is never executed. Activation parameters of environments should belong to identifiers of the highest-level blocks composed in the current system. Those parameters should be pairwise distinct in all environments (Ex.4.12, line 15). This prevents undesirable interferences that may occur when a block activation is constrained by more than one environment.

Connections between components are necessarily *binary*, which is expressed by channels occurring in exactly one pair of components. A block and an environment can be connected using a set of variables "$X_0 : T_0$, … , $X_n : T_n$" 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 (Ex.4.12, lines 13 and 16). Connections between mediums and blocks on receive and send channels are carried out similarly (Ex.4.12, lines 11 and 19).

Alternatively, channels can occur in only one component. If such is the case of an input or receive channel of blocks, arbitrary values are assigned to its parameters. Such channels are useless in environments and mediums, since their associated signals will never be executed.

```
chan     ::=  <X_0 , ... , X_n>
            |  <_ , ... , _>
            |  <any T_0 , ... , any T_n>
            |  ?<X_0 , ... , X_n>
            |  ?<_ , ... , _>
            |  B_0 , ... , B_n

system   ::=  system S (vars) is
                alias B_0 {args} as B'_0 , ... , B_m {args} as B'_m ,
                      N_0 {args} as N'_0 , ... , N_n {args} as N'_n ,
                      M_0 {args} as M'_0 , ... , M_k {args} as M'_k
                var vars
                  block list
                    B'_0 (chan , ... , chan)[chan , ... , chan]
                    , ... ,
                    B'_m (chan , ... , chan)[chan , ... , chan]
                  environment list
                    N'_0 (chan , ... , chan)
                    , ... ,
                    N'_n (chan , ... , chan)
                  medium list
                    M'_0 [chan , ... , chan]
                    , ... ,
                    M'_k [chan , ... , chan]
                end system
```

Fig. 6. The syntax of GRL systems

Channels whose parameters are declared as formal parameters of the system are observable outside the system (Ex.4.12, line 1). Channels whose parameters are declared as temporary variables are not (Ex.4.12, lines 7-8). Distinction between observable and non observable channels is a key device for abstraction, inspired by process algebra [Mil82]. This is essential for verification.

Blocks cannot be directly connected to each other using channels. This ensures arbitrary interleavings between their executions. Environments and mediums cannot be connected to each other either. They are 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. 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 the component at the same moment is grouped in one single channel.

In Example 4.12, in the beginning of each step of *Aileron*, environment *Control* and medium *Buffer* execute, providing values to inputs *safe* and *r_move*. Then, at the end of the step, *Control* and *Buffer* execute once more, consuming values of outputs *pos* and *r_move*. 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.

Remarks. *Deterministic* GALS systems (e.g., [PBCB06]) can be described in GRL, despite the nondeterministic behaviour of mediums and environments. In asynchronous deterministic systems, messages are not lost and are delivered in the same order in which they have been received. To this aim, the buffering mechanisms specified in mediums should comply with the activation policies defined in environments. For example, in quasi-synchronous systems, two-place FIFOs can be sufficient to ensure the reliability of message transmission. This issue will be discussed in Section 7.

Several instances of GALS systems consist of three types of components: synchronous components, asynchronous interconnects, and an interface between the synchronous and asynchronous components. In GRL, those components correspond respectively to blocks which are pure synchronous components, mediums, and data signals inside mediums. Indeed, send and receive signals are executed respectively before and after each step of each connected block.

Example 4.13. *Prim*, *Sec*, and *Aileron* of the FCS example can be composed by using an AFDX[2] (*Avionics Full DupleX Switched Ethernet*) architecture. For conciseness, we represent only message transmission from *Prim* and *Sec* to *Aileron*. In addition to synchronous components, an AFDX architecture contains *end systems* and *AFDX interconnects*.

End systems represent communication interfaces between synchronous components and the asynchronous network. In GRL, they are represented by signals inside mediums.

AFDX interconnects are networks of switches that forward messages to their appropriate destination. We implement them by a network of mediums: *Switch_Prim*, *Switch_Sec*, and *Switch_Ail*. Those mediums are connected to each other by using "additional" blocks *Switch_Prim_to_Ail* and *Switch_Sec_to_Ail* since GRL mediums cannot be directly connected to each other. The introduced blocks can be used to implement transmission message delays, by setting constraints on their activation.

```
1   block list
2     Prim     (order1)[?s_move1],
3     Sec      (order2)[?s_move2],
4     Aileron (safe , ?pos)[r_move_prim_ail,  r_move_sec_ail ],
5     Switch_Prim_to_Ail    [s_move_prim_ail,  ?s_move_prim_ail],
6     Switch_Sec_to_Ail     [ s_move_sec_ail ,  ?s_move_sec_ail ]
7   medium list
8     Switch_Prim [s_move1, ?s_move_prim_ail],
9     Switch_sec   [s_move2, ?s_move_sec_ail ],
10    Switch_Ail   [s_move_prim_ail,  ?r_move_prim_ail,  s_move_sec_ail ,  ?r_move_sec_ail ]
```

4.7. Overview of the formal semantics

Following process algebraic languages (e.g., CCS, CSP, LNT), the formal semantics of GRL are given in terms of LTS (*Labelled Transition System*), using Plotkin-style structural operational semantic rules [Plo81]. An LTS is a quadruple $(S, L, \rightarrow, s_0)$ where S is a set of states, $s_0 \in S$ is the initial state, L is a set of labels, and $\rightarrow \subseteq S \times L \times S$ is the labelled transition relation. The contents of states are not observable. To define the dynamic semantics of GRL systems, we first introduce the following notions:

1. A *store*, written ρ, is a partial function from variables to their current values. We write $[X_1 \leftarrow e_1, ..., X_n \leftarrow e_n]$ for a store ρ mapping each variable X_i to the corresponding value e_i, where i belongs to the interval [1 .. n]. We write $dom\ (\rho)$ for the domain of store ρ, defined by $dom\ (\rho) = \{X_1, \ldots, X_n\}$. For $\{Y_1, \ldots, Y_p\} \subseteq dom\ (\rho)$, 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 write $\rho_1 \oplus \rho_2$ for the update of ρ_1 with ρ_2, which is a store defined 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 $\rho_1 \oplus \ldots \oplus \rho_n$.
2. A *stack*, written σ is a sequence of component identifiers. Formally, 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 identifier of a component instance. We write $\sigma_1.\sigma_2$ for the concatenation of stacks σ_1 and σ_2. We define function *prefix* determining whether a stack σ_1 is a prefix of stack σ_2 as follows: $prefix\ (\sigma_1,\ \sigma_2)\ =\ (\exists\ \sigma')\ \sigma_1.\sigma' = \sigma_2$.
   During the execution, a unique stack, called *instantiation stack*, can be associated to each component instance. This stack consists of the ordered list of all embedded component instances from the highest-level component (block, medium, or environment) down to the current component, transitively. Such instantiation stacks are similar to call stacks in ordinary programming languages. In GRL, a finite and unique instantiation stack can be associated to each component instance since both recursive and shared components are disallowed.

[2] http://www.afdx.com/

Example 4.14. Let "**alias** B **as** B'" be a highest-level block aliasing inside a system S. The stack associated to instance 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 system S. The stack associated to instance M' is $\epsilon.M'$ and the one associated to B' in this context is "$\epsilon.M'.B'$".

In the sequel, we omit the initial ϵ in non empty stacks, e.g., we write "$M'.B'$" instead of "$\epsilon.M'.B'$".

3. A *memory*, written μ, is a partial function from stacks to stores. Memories implement the internal state of components. We write $[\sigma_1 \leftarrow \rho_1, ..., \sigma_n \leftarrow \rho_n]$ for a memory mapping a stack σ_i to a store ρ_i (i $\in$ [0 .. n]). The notation $\sigma_i \leftarrow \rho_i$ means that ρ_i defines the static variable values of the component whose stack is σ_i. We write $\mu_1 \oplus \mu_2$ for the update of μ_1 with μ_2, which is a memory defined similarly as store update. The notation $\bigoplus_{i \in 1..n} \mu_i$ stands for the sum $\mu_1 \oplus \ldots \oplus \mu_n$.
4. We define a function *mem* which extracts from a memory μ the submemory corresponding to the component whose instantiation stack is σ as well as the memories of its subblocks.

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

Example 4.15. The internal state of block *Aileron* (Ex. 4.1, page 9) is composed of the static variable *pre_pos*. Its initial memory is thus $[Aileron \leftarrow [pre\_pos \leftarrow 2]]$. Block *Interval* (Ex. 4.6, page 12) does not have an internal state. Its memory is thus the empty memory [].

The LTS corresponding to GRL system is defined as follows. The set of states is the union of all component memories in the system. The initial state is the initial memory, in which each static variable is assigned its (mandatory) initial value. In a given state μ, the execution of a step of some block B, involving a combined execution of its connected mediums and environments, leads to a transition of the form:

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

Each ch_i (i $\in$ [1 .. m]) has the form "$X_1 = e_1, \ldots, X_p = e_p$", where X_i (i $\in$ [1 .. p]) are the actual parameters of input and output channels and e_i (i $\in$ [1 .. p]) their respective values. Similarly, each ch'_i (i $\in$ [1 .. n])) contains parameters and values of receive and send channels.

To formally define this relation, we need the semantics of expressions, statements, blocks, environments, mediums, and systems, which we describe below.

Semantics of expressions. The semantics of expressions are defined by a relation of the form $\{E\}\rho \rightarrow e$. In this relation, E denotes an expression, ρ denotes the store in which E is evaluated, and e is the resulting value of E in store ρ.

Semantics of statements. The semantics of statements are defined by a relation of the form $\{I\}\sigma, \rho, \mu \xrightarrow{\ell} \rho', \mu'$. In this relation, σ is the stack of the current component (inside which I is executed), ρ is the store defining parameters and variables of the current component, and μ is the memory defining static variables of the current component and of its subblocks. It means that the execution of statement I defines a transition, labelled ℓ, from memory state ρ, μ to ρ', μ'. Store ρ' and memory μ' represent the update of ρ and μ, respectively. Label ℓ has one of the following forms:

- ϵ means that the execution of I has terminated normally without encountering any signal.
- B_0 means that the execution of I has terminated and encountered an activation signal on the activation parameter B_0.
- $?\langle X_0, \ldots, X_n \rangle$ means that the execution of I has terminated and encountered a signal on the input or receive channel $X_0, \ldots, X_n$.
- $\langle X_0, \ldots, X_n \rangle$ means that the execution of I has terminated and encountered a signal on the output or send channel $X_0, \ldots, X_n$.

GRL has rules for every kind of statement. For instance, the following rule defines the semantics of deterministic assignment:

$$\frac{\{E\}\ \rho \rightarrow e}{\{X\ \texttt{:=}\ E\}\ \sigma, \rho, \mu \xrightarrow{\epsilon} \rho \oplus [X \leftarrow e], \mu}$$

Note that this rule updates the store but not the memory even if X is a static variable. Note also that μ is not used to evaluate E because the store ρ already contains a copy of the static variables local to the current component. Construction of this store and memory update are handled at the level of component invocation (see semantics of blocks below).

The definition of statements is standard (see [JLM14b]). Nondeterministic statements lead to a nondeterministic relation. Hereafter, we give the rules defining signals, which are inspired from communication action semantics in process algebra (LNT in particular). Signals are defined by the following three rules:

$$\frac{\{I_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{\epsilon}\ \rho',\ \mu'}{\{\textbf{when}\ \texttt{?<}X_0\texttt{,}\ldots\texttt{,}X_n\texttt{>}\ \texttt{->}\ I_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{?\langle X_0,\ldots,X_n\rangle} \rho',\ \mu'} \qquad \text{input or receive signal}$$

$$\frac{\{I_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{\epsilon}\ \rho',\ \mu'}{\{\textbf{when}\ \texttt{<}X_0\texttt{,}\ldots\texttt{,}X_n\texttt{>}\ \texttt{->}\ I_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{\langle X_0,\ldots,X_n\rangle} \rho',\ \mu'} \qquad \text{output or send signal}$$

$$\frac{}{\{\textbf{enable}\ B_0\}\ \sigma,\ \rho,\ \mu \xrightarrow{B_0} \rho, \mu} \qquad \text{activation signal}$$

Semantics of blocks. We consider a block invocation of the form "B' `(`$chan_0$`,` $\ldots$ `,`$chan_m$`)[`$chan'_0$`,` $\ldots$ `,`$chan'_n$`]`", where $chan_0, \ldots, chan_m$ (resp., $chan'_0, \ldots, chan'_n$) denote input/output (resp., receive/send) actual channels. We write $vars_0, \ldots, vars_m$ (resp., $vars'_0, \ldots, vars'_n$) for the formal channels corresponding to these actual channels as defined in block definition. We also write $vars_{stat}$ for the list of static variables of the block.

Given a store ρ, an actual channel $chan$, and the corresponding formal channel $vars$, we write $update\ (chan,\ vars,\ \rho)$ for the store which is empty if $chan$ is an unconnected channel, or which assigns to each parameter in $vars$ the value of the corresponding parameter in $chan$ available in store ρ.

Given the store ρ and memory μ in which this block invocation is executed, we can construct the store ρ_{init} in which the body I_0 of the block is to be executed as follows: Constant and input parameters are assigned their actual values, default values corresponding to unconnected parameters are fetched in the block definition, temporary variables are assigned their initial values, if any, and static variables are assigned the value they had in memory μ.

The following rule defines the semantics of such a block invocation:

$$\frac{\{I_0\}\ \sigma.B',\ \rho_{init},\ \mu\ \xrightarrow{\epsilon}\ \rho_{ret},\ \mu_{ret}}{\{B'\ \texttt{(}chan_0\texttt{,}\ldots\texttt{,}chan_m\texttt{)[}chan'_0\texttt{,}\ldots\texttt{,}chan'_n\texttt{]}\ \}\sigma, \rho, \mu\ \xrightarrow{\epsilon}\ \rho',\ \mu'}$$

where:

$$\begin{aligned} \rho' &= \rho_{ret} \oplus \bigoplus_{i \in 0..m} update\ (chan_i,\ vars_i,\ \rho_{ret}) \oplus \bigoplus_{i \in 0..n} update\ (chan'_i,\ vars'_i,\ \rho_{ret}) \\ \mu' &= \mu_{ret} \oplus [\sigma.B' \leftarrow \rho_{ret|vars_{stat}}] \end{aligned}$$

The execution of I_0 terminates by producing an updated store ρ_{ret} and an updated memory μ_{ret}. The execution of B' terminates by producing store ρ' and memory μ'. Store ρ' updates ρ_{ret} with the actual values of output and send parameters. Memory μ' updates μ_{ret} with the current values of static variables of B', which are available in store ρ_{ret}. Note that μ_{ret} already takes into account the memory updates done in the subblocks of B', if any. The label of the transition is necessarily ϵ since GRL static semantics prohibit the use of signals inside blocks.

Semantics of environments and mediums. The following rules define the invocation of environment (left-hand rule) and mediums (right-hand rule):

$$\frac{\{I_0\}\ \sigma.N',\ \rho_{init},\ \mu \xrightarrow{\ell_i} \rho_{ret},\ \mu_{ret} \quad i \in 0..m}{\{\ N'\ (chan_0, \ldots, chan_m)\ \}\ \sigma, \rho, \mu \xrightarrow{chan_i} \rho',\ \mu'} \qquad \frac{\{I_0\}\ \sigma.M',\ \rho_{init},\ \mu \xrightarrow{\ell_i} \rho_{ret},\ \mu_{ret} \quad i \in 0..m}{\{\ M'\ [chan_0, \ldots, chan_m]\ \}\ \sigma,\ \rho,\ \mu \xrightarrow{chan_i} \rho',\ \mu'}$$

where ℓ_i is the formal channel of M' that corresponds to the actual channel $chan_i$, and ρ' and μ' are identical to their definition in the semantics of block invocation.

Contrarily to block execution, labels of the above transition rules are necessarily different from ϵ as the execution of environments and mediums is triggered only on signal execution.

Semantics of systems. We note *block*, *environment*, and *medium* as shorthands of component invocations inside a system:

$$\begin{array}{rcl} block & ::= & B'\ (chan_0, \ldots, chan_m)\,[chan'_0, \ldots, chan'_n] \\ environment & ::= & N'\ (chan_0, \ldots, chan_m) \\ medium & ::= & M'\ [chan_0, \ldots, chan_m] \end{array}$$

Each component of the system is associated to a unique index. We write B'_i (resp., N'_i, M'_i) for the identifier of component instance $block_i$ (resp., $environment_i$, $medium_i$). Similarly, in each component invocation, each channel is also associated to a unique index.

Given a block B'_i, we note $\mathcal{A}$ the set, which is either singleton or empty, containing the index of the environment constraining its activation, if any. Similarly, we note $\mathcal{I}$, $\mathcal{O}$, $\mathcal{R}$, and $\mathcal{S}$ the sets (possibly empty but not necessarily singleton if not empty) of component indexes that have a channel connected to respectively an input, output, receive, and send channel of B'_i. Given a block index i, an environment or medium index j, and a channel type *mode* belonging to {*in*, *out*, *receive*, *send*}, we write *channel (i, j, mode)* the channel of type *mode* in the component indexed j that is connected to the block indexed i. Note that GRL static semantics ensure that there is exactly one such channel.

Given a block invocation, for actual channels that are not connected to any component, a store ρ_{any} is constructed to assign arbitrary values to their corresponding formal channels. Therefore, the construction of ρ_{any} yields nondeterminism.

The following rule defines the combined execution of block B'_i and its connected environments and mediums:

$$\begin{array}{llll} (\forall j \in \mathcal{A}) & \{environment_j\} & \epsilon,\ [],\ mem\ (\mu,\ N'_j) & \xrightarrow{B'_i} \rho'_{\mathcal{A}_j},\ \mu'_{\mathcal{A}_j} \\ (\forall l \in \mathcal{I}) & \{environment_l\} & \epsilon,\ [],\ \mu_{\mathcal{I}_l} & \xrightarrow{channel(i,\ l,\ out)} \rho'_{\mathcal{I}_l},\ \mu'_{\mathcal{I}_l} \\ (\forall k \in \mathcal{R}) & \{medium_k\} & \epsilon,\ [],\ mem\ (\mu,\ M'_k) & \xrightarrow{channel(i,\ k,\ send)} \rho'_{\mathcal{R}_k},\ \mu'_{\mathcal{R}_k} \\ & \{block_i\} & \epsilon,\ \rho_i,\ mem\ (\mu,\ B'_i) & \xrightarrow{\epsilon} \rho'_i,\ \mu'_i \\ (\forall m \in \mathcal{O}) & \{environment_m\} & \epsilon,\ \rho'_i,\ \mu_{\mathcal{O}_m} & \xrightarrow{channel(i,\ m,\ in)} \rho'_{\mathcal{O}_m},\ \mu'_{\mathcal{O}_m} \\ (\forall n \in \mathcal{S}) & \{medium_n\} & \epsilon,\ \rho'_i,\ \mu_{\mathcal{S}_n} & \xrightarrow{channel(i,\ n,\ receive)} \rho'_{\mathcal{S}_n},\ \mu'_{\mathcal{S}_n} \end{array}$$

$$\mu \xrightarrow{\text{label}(block_i,\ \rho'_i)} \mu \oplus \bigoplus_{j \in \mathcal{A}} \mu'_{\mathcal{A}} \oplus \bigoplus_{l \in \mathcal{I}} \mu'_{\mathcal{I}} \oplus \bigoplus_{k \in \mathcal{R}} \mu'_{\mathcal{R}} \oplus \bigoplus_{m \in \mathcal{O}} \mu'_{\mathcal{O}} \oplus \bigoplus_{n \in \mathcal{S}} \mu'_{\mathcal{S}}$$

where:

$$\begin{array}{rcl} \rho_i & = & \rho_{any} \oplus \bigoplus_{j \in \mathcal{A}} \rho'_{\mathcal{A}_j} \oplus \bigoplus_{l \in \mathcal{I}} \rho'_{\mathcal{I}} \oplus \bigoplus_{k \in \mathcal{R}} \rho'_{\mathcal{R}} \\ \mu_{\mathcal{I}_l} & = & mem\ (\mu,\ N'_l) \oplus \bigoplus_{j \in \mathcal{A}} mem\ (\mu'_{\mathcal{A}_j},\ N'_l) \\ \mu_{\mathcal{O}_m} & = & mem\ (\mu,\ N'_m) \oplus \bigoplus_{j \in \mathcal{A}} mem\ (\mu'_{\mathcal{A}_j},\ N'_m) \oplus \bigoplus_{l \in \mathcal{I}} mem\ (\mu'_{\mathcal{I}_l},\ N'_m) \\ \mu_{\mathcal{S}_n} & = & mem\ (\mu,\ M'_n) \oplus \bigoplus_{k \in \mathcal{R}} mem\ (\mu'_{\mathcal{R}_k},\ M'_n) \end{array}$$

where label($block_i$, ρ'_i) constructs the label of the transition as explained earlier.

First, the environment constraining the activation of the block is executed in the empty store and in its

```
process   ::= process Π [G_0:Γ_0, ..., G_m:Γ_m](param_0, ..., param_n) is
                B
              end process
function  ::= function F (param_0, ..., param_m)[:T] is I end function
param     ::= (in | out | in out) X_0:T_0, ..., X_n:T_n
channel   ::= channel Γ is tuple_0, ..., tuple_m end channel
tuple     ::= (T_0, ..., T_m)
arg       ::= !E | ?X | !?X
O         ::= [!]E | ?X
E         ::= X | F(E_1, ..., E_n) | C(P_1, ..., P_n)
P         ::= X | C(P_1, ..., P_n)
I         ::= null
            | X:=E
            | X[E_0]:=E_1
            | X.f:=E
            | eval F(arg_0, ..., arg_n)
            | I_0;I_1
            | if E_0 then 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
            | for I_0 while E by I_1 loop I_2 end loop
            | case E is P_0 -> I_0 | ... | P_n -> I_n | [any -> I_{n+1}] end case
            | var X_0:T_0, ..., X_n:T_n in I end var
              /* the following constructs are reserved for the control part */
            | X := any T [where E]
            | G(O_0, ..., O_n)
            | hide G_0:Γ_0, ..., G_n:Γ_n in I end hide
            | par G_0, ..., G_n in I_0 || ... || I_m end par
            | select I_0 [] ... [] I_n end select
            | Π[G_0, ..., G_n](arg_0, ..., arg_n)
```

Fig. 7. LNT syntax (excerpt)

own memory which is extracted from the current memory μ by using function *mem*. It produces the empty store ($\rho'_{\mathcal{A}_j} = []$) since no output channel is activated, and an updated memory $\mu'_{\mathcal{A}_j}$.

Similarly, all environments (resp., mediums) that are connected to input (resp., receive) channels of B'_i are executed in the empty store and in their own memories. In particular, if the environment constraining the execution of B'_i is also connected to an input channel of B'_i, its own memory is the one produced by its previous execution during the current step of B'_i, which is captured by the definition of $\mu_{\mathcal{I}_l}$.

Second, block B'_i is executed in the store ρ_i and its own memory. Store ρ_i assigns values to all input and receive parameters of B'_i which have been produced by other components. Those parameters are available in stores $\rho'_{\mathcal{R}_k}$ ($k \in \mathcal{R}$) and $\rho'_{\mathcal{I}_l}$ ($l \in \mathcal{I}$). Store ρ'_i, produced by the block execution, assigns values to all its output and send parameters.

Last, all mediums (resp., environments) that are connected to send (resp., output) channels of B'_i are executed in store ρ'_i and their own memories.

The execution of the system defines a transition updating the current memory μ with all memories produced by the executed components. This rule can be instantiated for any block of the system, which leads to an interleaving of block executions.

5. The LNT Language and the CADP Toolbox

LNT [CCG+14] is a simplified variant of the E-Lotos [ISO01] standard. LNT combines the best features of imperative programming languages and value-passing process algebras. The translation of GRL to LNT requires a subset of LNT constructs, whose syntax is given in EBNF in Figure 7.

The generic terminal symbols *T*, *X*, *F*, *Γ*, *G*, and *Π* denote identifiers for, respectively types, variables, functions, channels, gates, and processes. The non-terminals *E*, *arg*, *O*, *P*, and *I* denote respectively expressions, actual parameters, offers, patterns, and behaviours.

LNT provides a rich constructed data types (e.g., records, sets, and lists), statements built upon standard algorithmic control structures (lines 1-11 of non-terminal *I*), and functions (non-terminal *function*). Functions

are parametrized by data variables. Formal parameters (non-terminal *param*) can be either of mode **in** (call by value), **out** (call by reference, the function being in charge of producing a value for the parameter), or "**in out**" (call by reference, the function being allowed to read and update the parameter value). Their corresponding actual parameters (non-terminal *arg*) are preceded by symbols "`!`", "`?`", and "`!?`", respectively.

LNT processes (non-terminal *process*) are defined similarly to functions, with addition of the following behaviours (non-terminal *I*): nondeterministic assignment (line 12), nondeterministic choice (line 16), gate communication (line 13), gate hiding (line 14), parallel composition (line 15), and process instantiation (line 17). Communication takes place by rendezvous on gates, with bidirectional transmission of multiple values. Gates can be typed by channels (not to be confused with the GRL notion of channels), defining a set of type tuples (non-terminal *channel*). Processes can be parametrized by both data variables and gates. The parallel composition operator allows multiway rendezvous, which is a rendezvous involving several processes. The formal operational semantics are defined in terms of LTSs, described in detail in [CCG+14].

LNT specifications can be verified using the toolbox CADP (*Construction and Analysis of Concurrent Processes*[3]) [GLMS13]. The LNT.OPEN tool translates LNT specifications into LTSs suitable for on-the-fly exploration. CADP provides more than 42 tools for various kinds of analysis such as simulation, model checking, equivalence checking, compositional verification, test case generation, and performance evaluation. In particular, the EVALUATOR 4.0 model checker allows one to verify temporal properties written in MCL (*Model Checking Language*) [MT08], which extends the alternation-free μ-calculus with generalized regular expressions, data-based constructs, and fairness operators.

6. Translation from GRL to LNT

This section presents the translation from GRL to LNT. Synchronous components with internal memory are translated into LNT functions, which are deterministic and stateless. An LNT implementation of the notion of internal memory is proposed. Such LNT functions are encapsulated in asynchronous processes (called *wrapper processes*, following [GT09]) so as to implement asynchronous component composition. In addition, a locking mechanism is proposed to ensure the atomicity of component steps in wrapper processes. As regards asynchronous components, environments and mediums are naturally translated into LNT processes. In particular, the translation of activation signals complies with the locking mechanism by constraining the execution of wrapper processes. Finally, concurrent processes are composed to interact with each other inside a higher-level process, called *root process*.

GRL being a rich language, we cannot present the full translation function formally in this paper[4]. Instead, we give here the main principles of the translation, illustrated on well-chosen examples.

6.1. Translation of Variables, Types, Expressions, and Statements

GRL data types are translated with no difficulty into LNT, owing to the ability of LNT to handle complex data types. Each GRL type T is translated into an LNT type with the same identifier as T[5]. Each GRL variable X is translated into an LNT variable with the same identifier and type as X. Expressions and statements have a direct, one-to-one, correspondence with their LNT counterparts. The only exception concerns signals, which are translated into slightly more complex statements involving LNT communication actions. The imperative style of both GRL and LNT makes rather straightforward such a translation.

6.2. Translation of Blocks

For the sake of clarity and without loss of generality, we organize this section as follows. We first consider blocks without internal state (i.e., without static variables). We present the translation of subblocks, i.e., blocks intended to be used inside components. Then, we present the translation of highest-level blocks, i.e.,

[3] See the web page `http://cadp.inria.fr` for more information.
[4] A complete translation from GRL to LNT is defined formally in `http://convecs.inria.fr/doc/grl2lnt.pdf`.
[5] In practice, the translation must ensure that the GRL identifier is not an LNT keyword. This is handled by our translator but we skip such low-level details in this presentation for brevity.

blocks intended to be composed asynchronously inside systems. Finally, we present the translation of static variables, which represent the internal states of block instances.

Translation of subblocks without internal state. A GRL block definition is systematically translated into an LNT function having the same identifier, called *definition function*. The definition function implements one block step, i.e., computes outputs from inputs. Such a translation is straightforward if GRL blocks have no internal state, as illustrated in the following example.

Example 6.1. The following block *Interval* is translated into the LNT definition function below.

```
-- GRL code
block Interval {thres:nat} (in pos: nat := 0, out ok: bool) is
  var sup, inf: bool
  sup := (thres < pos) and (pos - thres) < 5);
  inf := (pos <= thres) and (thres - pos) < 5);
  if (sup or inf) then
    ok := true
  else
    ok := false
  end if
end block
```

```
-- LNT code
function Interval (in thres:Nat8, in pos:Nat8, out ok:Bool) is
   var sup:Bool, inf:Bool in
      sup := (thres < pos) and (pos - thres) < 5 of Nat8);
      inf := (pos <= thres) and (thres - pos) < 5 of Nat8);
      if (sup or inf) then
         ok := true
      else
         ok := false
      end if
   end var
end function
```

Each GRL constant and input parameter (resp. output parameter) is translated into an LNT input parameter (resp. output parameter). In Example 6.1, input parameters *thres* and *pos* and output parameter *ok* in function *Interval* correspond to, respectively constant, input, and output parameters of block *Interval*. GRL temporary variables are translated into LNT local variables as illustrated by *sup* and *inf*.

If a block is defined with external C code, it is translated into both an LNT function and into an interface C function. The LNT function encapsulates the generated C code, by using pragmas "**!implementedby**" and "**!external**". The interface C function calls the C code defined by the user. This is required by the front-end compiler of LNT to properly handle the external C code.

Example 6.2. The following block *C_Shift* is translated into LNT as below.

```
-- GRL file importing the C file
block C_Shift (in numb:nat, in bits:nat, out result:nat) is
   !c "Shift"
end block
-- C file
void Shift (GRL_NAT number, GRL_NAT bits, GRL_NAT * result){
   unsigned char arg_number = GRL_NAT_TO_UNSIGNED_CHAR (number);
   unsigned char arg_bits = GRL_NAT_TO_UNSIGNED_CHAR (bits);
   * result = GRL_UNSIGNED_CHAR_TO_NAT (arg_number << arg_bits);
}
```

```
-- generated LNT file
function GRL_C_Shift (in GRL_numb : Nat8, in GRL_bits : Nat8, out GRL_result : Nat8) is
   !implementedby "Shift%1"
   !external
   null
end function
-- generated interface C file
GRL_NAT Shift1 (GRL_NAT numb, GRL_NAT bits){
```

```
 9      GRL_NAT result;
10      Shift (numb, bits, &result);
11      return result;
12  }
```

Each subblock aliasing "**alias** $B\{\dots\}$ **as** B'" is translated into an LNT function, called *aliasing function*, which encapsulates a call to the definition function of B with appropriate parameters, corresponding to the constant parameters of block B. If the block aliasing uses actual constant parameters of the form "_", default values of the corresponding formal parameter are fetched in the block definition and passed to the LNT function call. Each aliasing function has a unique name[6]. This prevents naming conflicts, since GRL subblocks with the same identifier can be aliased in different components.

Example 6.3. Subblock aliasing "**alias** *Interval* {5} **as** *Interval*" of block *Interval* inside a component is translated into the following aliasing function.

```
1  function Subblock_Interval (in pos:Nat8, out ok:Bool) is
2      Interval (5, pos, ?ok)
3  end function
```

The invocation of a GRL subblock is translated into a call to the corresponding LNT aliasing function. GRL actual parameters of the form "E" or "?X" are directly translated into LNT actual parameters of the same form. Unconnected input parameters (of the form "_") are translated similarly to actual constant parameters. For unconnected output parameters (of the form "?_"), "dummy" variables are declared, then passed to the LNT function call.

Example 6.4. The invocation "*Interval* (*lastPos*, ?*ok*)" of subblock *Interval* inside environment *Control* (Section 4.4) is translated as follows:

```
1  eval Interval (lastPos, ?ok)
```

The invocation "*Interval* (_, ?*ok*)" having an unconnected input parameter is translated as follows:

```
1  eval Interval (0, ?ok)
```

The invocation "*Interval* (*lastPos*, ?_)" having an unconnected output parameter is translated as follows:

```
1  var dummy_ok:bool in
2      eval Interval (lastPos, ?dummy_ok)
3  end var
```

Behaviour preservation. The synchronous assumptions are granted for free in the translation of GRL subblocks. Indeed, LNT functions are deterministic and execute atomically, no transition being produced. This coincides with the assumption that computations and data processing are instantaneous in synchronous components.

Translation of highest-level blocks without internal state. A block used as a highest-level block is also translated into an LNT definition function as explained previously. The only addition is that GRL receive and send parameters (if any) are translated into LNT input and output parameters, respectively.

Highest-level blocks are intended to be composed asynchronously and communicate with environments and mediums. Thus, their aliasing is translated into an LNT wrapper process encapsulating the corresponding LNT definition function. The wrapper process implements the cyclic behaviour of the block (input reading, computations, output update) inside an infinite loop.

Example 6.5. Aliasing "**alias** *Interval* {5} **as** *Interval*" of block *Interval* inside a system is translated into the following wrapper process. Details of the translation will be explained in the sequel.

```
1  process Wrapper_Interval [Pos:Nat, Ok:Bool,
2                            Start:Block, Finish:None] is
3    var pos:Nat, ok:Bool in
4      loop
5        Start (Interval);
```

[6] For simplicity of the present article, we consider that LNT functions and processes corresponding to GRL component instances have the same identifiers as their GRL components.

```
6         Pos (?pos);                  -- reading input parameter
7         eval Interval (pos, ?ok);    -- calling definition function
8         Pos (pos);                   -- writing output parameter
9         Finish
10      end loop
11    end var
12  end process
```

Example 6.6. Aliasing "**alias** *Aileron* {_} **as** *Aileron*" of block *Aileron* inside a system is translated into the following wrapper process. Details of the translation will be explained in the sequel.

```
1   process Wrapper_Aileron [Safe:Bool, R_Move:Move_Type,
2                            S_Ok:Bool, Start:Block, Finish:None] is
3     var safe:Bool, move:move_type, ok:Bool, ... in
4     ...
5       loop
6         Start (Aileron);
7         Safe (?safe);                                -- reading input parameter
8         R_Move (?move);                              -- reading receive parameter
9         eval Aileron (safe, move, ?pos, ?ok, ...) ;  -- calling definition function
10        Pos (pos);                                   -- writing output parameter
11        S_Ok (ok);                                   -- writing send parameter
12        Finish
13      end loop
14    end var
15  end process
16
17  channel Bool is
18    (Bool)
19  end channel
```

Asynchronous communication between processes is performed by exchanging data on gates. An LNT gate is generated for each GRL channel (i.e., tuple of parameters) of a block. Parameters composing the GRL channel are translated into LNT local variables, to be exchanged on the corresponding gate. Those variables are passed as actual parameters to the LNT function encapsulated inside the process. In Example 6.6, variable *safe* (line 3), corresponding to a GRL input channel, is received on gate *Safe* (line 7) and passed as actual input parameter to the definition function (line 9).

LNT gates are typed by LNT channels. An LNT channel is generated for each LNT gate defining its communication profile, i.e., number and types of exchanged data. Our translation ensures that generated gates are pairwise distinct. For example, gate *Safe* (Ex.6.6, line 1) is typed by channel *Bool* (Ex.6.6, lines 17-19).

The intrinsic infinite sequence of steps of GRL highest-level blocks is implemented in LNT by an infinite loop. Its body consists of a sequential composition of:

1. Reception of data from other components using LNT gates, which we call *reception gates*. Reception gates correspond to GRL receive and input channels, on which data is received from environments and mediums. For example, in lines 7-8 of Example 6.6, gates *Safe* and *R_move* are used to receive the input parameter *safe* and receive parameter *move*.
2. Call to the LNT definition function of the GRL block, received data being passed as actual **in** parameters. The function returns (through actual **out** parameters) the data to be emitted to other components (Ex. For example, in line 9 of Example 6.6, input actual parameters *safe* and *move* passed to function *Aileron* correspond to data received on gates *Safe* and *R_move*. Output actual parameters *pos* and *ok* are returned by the function.
3. Communication with environments and mediums using LNT gates, which we call *emission gates*. Emission gates correspond to GRL output and send channels, on which data is sent to other environments and mediums. For example, in lines 10-11 of Example 6.6, gates *Pos* and *S_Ok* are used to emit the output parameter *pos* and send parameter *ok*.

Note that since there is no dependency relation between reception gates nor between emission gates (i.e., their sets of variables are disjoint in GRL semantics), the order in which receptions (resp., emissions) are performed is irrelevant.

The invocation of a highest-level block inside a system is translated into a call to its LNT wrapper process. An illustration is given in Example 6.7.

Example 6.7. The invocation "*Aileron* `(`*safe*`, ?`*pos*`)[`*r_move*`, ?`*s_ok*`]`" of the highest-level block *Aileron* is translated into LNT as follows:

```
1  Wrapper_Aileron [Safe, R_move, Pos, S_ok, Start, Finish]
```

GRL actual channels that are either unconnected ("`<_, ..., _>`") or wildcard ("**<any** T_0**, ...,any** T_n**>**") are unused in communications. Therefore, no corresponding gate is generated. This helps reducing the size of the state space corresponding to the block.

Atomicity. Each LNT gate communication generates a transition in the LTS. Following the translation of wrapper processes, each GRL transition (i.e., one atomic block step) maps to a sequence of LNT transitions. Such a transition sequence should be atomic, i.e., interleaving of individual transitions in LNT transition sequences corresponding to different blocks should be prevented. To this aim, a locking mechanism is implemented by using two additional gates *Start* and *Finish* and a *Mutex* process. Gate *Start* starts the transition sequence by acquiring the lock (Ex. 6.6, line 6). Gate *Finish* finishes the transition sequence by releasing the lock (Ex. 6.6, line 12). Gates *Start* and *Finish* are common to process *Mutex* and all wrapper processes of blocks. Each wrapper process must synchronize individually with *Mutex*, defined as follows:

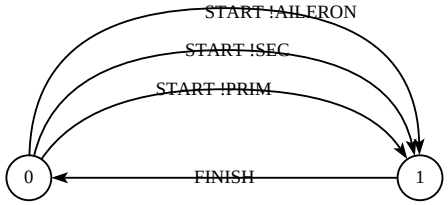
```
1  process Mutex [Start:Block, Finish:None] is
2     var block:block in
3        loop
4           Start (?block);
5           -- Only the process of the GRL block named ''block''
6           -- can execute
7           Finish
8        end loop
9     end var
10 end process
11
12 channel None is () end channel
13
14 channel Block is (Block) end channel
```

Type *Block* is an enumerated type generated systematically by the translation. Its values consist of the identifiers of all highest-level blocks.

Example 6.8. For the FCS example comprising blocks *Prim*, *Sec*, and *Aileron*, the following type *Block* is generated.

```
1  type Block is
2     Prim, Sec, Aileron with "=="
3  end type
```

Accordingly, the LTS of process *Mutex* is the following.



Such a locking mechanism introduces no deadlock. A transition *Start* is necessarily followed by a transition *Finish* in both the Mutex and wrapper processes. Specifically, gates occurring between *Start* and *Finish* in the wrapper processes correspond to data exchange with mediums and environments. Mediums and environments are not atomic and always accept data exchange with different blocks according to GRL static semantics. Therefore, once a sequence of LNT transitions has started, it is guaranteed to finish.

Example 6.9. A transition in GRL semantics corresponding to one step of block *Aileron* can be:

$$S_0 \xrightarrow{AILERON\ (SAFE{=}TRUE,\ POS{=}2)[MOVE{=}UP,\ OK{=}TRUE]} S_1$$

The corresponding LNT transition sequence obtained by our translation is:

$$S_0 \xrightarrow{START\ !AILERON} S_1 \xrightarrow{SAFE\ !TRUE} S_2 \xrightarrow{MOVE\ !UP} S_3 \xrightarrow{POS\ !2} S_4 \xrightarrow{OK\ !TRUE} S_5 \xrightarrow{FINISH} S_6$$

Behaviour preservation. In general, a GRL transition of the form: $S_0 \xrightarrow{B\ (ch_1,...,ch_m)[ch'_1,...,ch'_n]} S_1$ corresponds to the following sequence of LNT transitions:

$$S_0 \xrightarrow{START\ !B} S_1 \xrightarrow{gate_1} S_2 \rightarrow ... \rightarrow S_m \xrightarrow{gate_m} S_{m+1} \xrightarrow{gate'_1} S_{m+2} \rightarrow ... \rightarrow S_{m+n} \xrightarrow{gate'_n} S_{m+n+1} \xrightarrow{FINISH} S_{m+n+2}$$

The LNT transitions are constructed as follows. To each GRL channel ch_i ($i \in [1..m]$) corresponds an LNT gate $gate_i$ and to each GRL channel ch'_i ($i \in [1..m]$) corresponds an LNT gate $gate'_i$. For a GRL channel of the form "$X_1 = e_1, \ldots, X_p = e_p$", the corresponding LNT gate has the form "$\mathsf{gate}\ (\mathsf{X_1}, \ldots, \mathsf{X_p})!\ e_1!\ ...!\ e_p$", where gate is a function returning a gate identifier built upon the identifiers X_1, ..., X_p.

Note that the expansion caused by the transformation is linear in the number of transitions, since the transitions composing an LNT transition sequence cannot interleave, owing to the locking mechanism. Such transformation is bijective, i.e., there is one-to-one correspondence between a GRL transition and an LNT transition sequence. Accordingly, the label of the GRL transition can be reconstructed from the labels of the LNT transition sequence, by merging them together and renaming the resulting transition. Strong equivalence between the LTS of GRL and the one generated by our translation is still preserved modulo the compression of the resulting LTS.

Translation of blocks with internal state. The internal state of a block is built upon the static variables of the block itself and those of its subblocks, transitively. Each block instance has its own copy of the block internal state and can read and update only the variables of that copy. For example, the internal state of block *Aileron*, defined in Example 4.1 (page 9), consists of its static variable *pre_pos*.

The translation of static variables is a main difficulty of the translation, since LNT functions enable only local variables, the values of which are lost between subsequent invocations. Therefore, we implement static variables of each block using LNT local variables declared in the wrapper process, and initialized once and for all just before the infinite loop. This is illustrated by variable *Aileron_pre_pos* (lines 3-4, LNT code) in Example 6.10 below, which implements the internal state of block *Aileron*. Variables implementing the internal state are then propagated through "**in out**" parameters to functions corresponding to blocks, transitively (Ex.6.10, line 17, LNT code). Therefore, variables implementing the internal state of a block are synthesized in a bottom-up manner in the LNT functions corresponding to its subblocks. This is possible because all block instances can be statically determined. 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 6.10. The definition function of block *Aileron* is defined at lines 13-20 of the LNT code below. The "**in out**" variable *pre_pos* implements the internal state of the block. The wrapper process corresponding to block *Aileron* is defined at lines 1-11. Variable *Aileron_pre_pos* implements the internal state of the block instance.

```
1  -- GRL code
2  block Aileron ... is
3     static var pre_pos:nat := 2 -- static variable
4      ...
5     pre_pos := new_pos
6  end block
```

```
1  -- LNT code
2  process Wrapper_Aileron [...] is
3    var Aileron_pre_pos: Nat8 in     -- internal state declaration
4      Aileron_pre_pos := 2 of Nat8; -- internal state initialization
5        loop
6         ...
7        eval Aileron (..., !?Aileron_pre_pos);  -- internal state can be updated inside Aileron
8         ...
```

```
 9      end loop
10    end var
11  end process
12
13  function Aileron (in safe:Bool,  ...,  out ok:Bool,
14                    in out pre_pos:Nat8) is -- internal state of block Aileron
15    var new_pos:Nat8 in
16        ...
17        ok := safe and (pre_pos != new_pos);
18        pre_pos := new_pos -- changing the internal state
19    end var
20  end function
```

6.3. Translation of Environments and Mediums

GRL environments have nondeterministic behaviour and support signals, which are used for asynchronous communication. Thus, a GRL environment is translated into an LNT process, called *definition process*. In the sequel, we present first the translation of environments with only data constraints, then that of environments with only activation constraints. Constraints on data and on activation are orthogonal, i.e., there is no additional complexity to translate environments containing both data and activation constraints.

Environments with data signals. The definition process is specified using gate declarations corresponding to GRL input and output channels, "**in out**" parameters corresponding to the environment internal state, and a statement translating the environment behaviour.

Data signals are translated into LNT communication actions to describe interactions with blocks. The gates introduced in block wrapper processes are used. In statement "**when** ?<X_0 , ... , X_n> -> I_0", values of variables $X_0, \ldots, X_n$ can be read in I_0. Such a signal is translated into a sequential composition of the form "G (?X_0 , ... , ?X_n) ; I_0", where G denotes the gate corresponding to the GRL signal and I_0 denotes the behaviour translating the GRL statement I_0.

In statement "**when** <X_0 , ... , X_n> -> I_0", statement I_0 should assign values to variables $X_0, \ldots, X_n$. Such a signal is translated into a sequential composition of the form "I_0 ; G (X_0 , ... , X_n)", where I_0 denotes the behaviour translating the GRL statement I_0 and G denotes the gate corresponding to the GRL signal.

Example 6.11. The following GRL environment *Control* is translated into the LNT process below. Lines 6-7 of the LNT code correspond to the translation of the GRL signal "**when** ?<*pos*> → *lastPos* := *pos*" (line 6, GRL code). Lines 10-12 correspond to the translation of the GRL signal "**when** <*safe*> → *interval* (*lastPos* , ?*ok*) ; *safe* := *ok*" (line 7, GRL code).

```
1  -- GRL code
2  environment Control (in pos:nat, out safe:bool) is
3    alias Interval {5} as Interval
4    static var lastPos:nat := 0
5    select
6        when ?<pos> -> lastPos := pos
7    [] when <safe> -> Interval (lastPos, ?safe)
8    end select
9  end environment
```

```
1  -- LNT code
2  process Control [Pos:Nat, Safe:Bool] (in out lastPos:Nat8) is
3    var pos:Nat8,  safe:Bool, ok:Bool in
4      select
5        -- signal ''on pos''
6        Pos (?pos);
7        lastPos := pos
8      []
9        -- signal ''on ?safe''
10       eval Interval (lastPos, ?ok);
11       safe := ok;
12       Safe (!safe)
13     end select
```

```
14    end var
15  end process
```

Similarly to highest-level blocks, environment aliasing is translated into an LNT wrapper process. This process encapsulates the definition process of the environment inside an infinite loop.

Example 6.12. Environment aliasing "**alias** *Control* **as** *Control*" is translated into the following wrapper process:

```
1   process Wrapper_Control [Pos:Nat, Safe:Bool] is
2       var Control_lastPos:Nat8 in
3         Control_lastPos := 0 of Nat8
4         loop
5            Control [Pos, Safe] (!?Control_lastPos)
6         end loop
7     end var
8   end process
```

Environments with block activation signals. In the LNT definition process of environments with block activation signals, each activation parameter is translated into an LNT input parameter. Those parameters have type *Block* (See section 6.2, page 26). The process definition is parametrized by gate *Start*, introduced by the locking mechanism.

Each environment definition process synchronizes on gate *Start* with both the *Mutex* and the wrapper processes of the blocks it constrains. As such, synchronizations on gate *Start* are multiway involving three processes. Therefore, a block B, whose activation is constrained by an environment N, can acquire the Mutex only if (1) it is not acquired by any other block and (2) environment N authorizes its activation by enabling the action *Start (B)*, corresponding to the activation signal. The GRL statement "**enable** B_0", enabling the execution of block B_0, is translated into gate communication "*Start* (B_0)".

Example 6.13. The following GRL environment *Backup* is translated into the LNT definition process below. Lines 7 and 10 of the LNT code is the translation of GRL signals "**enable** *Prim*" and "**enable** *Sec*", respectively.

```
1   -- GRL code
2   environment Backup (block Prim, block Sec) is
3      static var p_alive, s_alive : bool := true
4          if (p_alive) then -- Prim executes
5             p_alive := any bool;
6             enable Prim
7          elsif (s_alive) then -- Sec executes
8             s_alive := any bool;
9             enable Sec
10         end if
11  end environment
```

```
1   -- LNT code
2   process Backup [Start:Block]
3                  (in Prim, Sec:Block, -- activation parameters
4                  in out s_alive:Bool, in out p_alive:Bool) is
5     if (p_alive) then
6       p_alive := any Bool;
7       Start (Prim)   -- signal ''on Prim''
8     elsif (s_alive) then
9       s_alive := any Bool;
10      Start (Sec)    -- signal ''on Sec''
11    end if
12  end process
```

Behaviour preservation. Note that deadlocks in GRL should occur only if all block activations are prevented by environments, i.e., all activation signals are unreachable. If an activation signal is reachable in GRL, its respective gate *Start* is necessarily reachable, since GRL statements have one-to-one direct correspondence with their LNT counterpart. Consequently, a deadlock in GRL leads to a deadlock in the translated LNT

code because all actions *Start* in the different process definitions of environments are unreachable. Then, none of the wrapper processes of different blocks can execute. Conversely, the translation of activation signals introduces no additional deadlocks because wrapper processes of blocks and the Mutex can always synchronize on gate *Start*.

The LNT wrapper process corresponding to environment aliasing with activation signals is parametrized by gate *Start*. It encapsulates the definition process in the same way as explained previously.

Example 6.14. The aliasing "*Backup* **as** *Backup*" of environment *Backup* inside a system is translated into the following LNT process, where actual parameters *Prim* and *Sec* are highest-level block identifiers of the system.

```
process Wrapper_Backup [Start : BLOCK] is
  -- Static variable declarations
  var Backup_s_alive : Bool, Backup_p_alive : Bool in
    -- Static variable  initializations
    Backup_s_alive := true;
    Backup_p_alive := true;
    -- Main loop
    loop
      Backup [Start] (Prim, Sec,
                      !?Backup_s_alive, !?Backup_p_alive)
    end loop
  end var
end process
```

Mediums. Similarly to environments, a medium definition is translated into an LNT definition process. GRL receive and send channels together with their respective signals are translated in the same way as input and output channels and data signals in environments. A medium aliasing is translated into an LNT wrapper encapsulating the LNT definition process inside an infinite loop.

6.4. Translation of Systems

A GRL system is translated into an LNT process, called *root process*, having the same identifier as the GRL system.

Example 6.15. The following GRL system *Main* is translated into the root process below. Details of the translation will be explained in the sequel.

```
-- GRL code
system Main (order1, order2:move_type, safe:bool, pos:nat) is
  alias Computer as Prim, Computer as Sec,
        Aileron {_} as Aileron,
        Backup as Backup, Control as Control,
        Buffer as Buffer

  var s_move1, s_move2, r_move: move_type,
      s_ok, r_ok1, r_ok2        : bool

    block list
      Prim    (order1)[r_ok1, ?s_move1],
      Sec     (order2)[r_ok2, ?s_move2],
      Aileron (safe, pos)[r_move, ?s_ok]
    environment list
      Backup (Prim, Sec),
      Control (pos, ?safe)
    medium list
      Buffer [s_move1, s_move2, ?r_move, ?r_ok1, ?r_ok2, s_ok]
end system
```

```
-- LNT code
process Main [Start:Block, Order1, Order2:Move_type, Safe:Bool, Pos:Nat8] is
  hide Finish:None, R_move:Move_type, S_ok:Bool, ... in
    par Start, Safe, Pos, R_move, S_ok, ... in
      par Start, Finish in
```

```
 6          Mutex [Start, Finish]
 7          ||
 8          -- Block wrappers
 9          par
10            Wrapper_Aileron [Safe, Pos, R_move, S_ok, Start, Finish]
11            || Wrapper_Sec [..., Start, Finish]
12            || Wrapper_Prim [..., Start, Finish]
13          end par
14        end par
15        ||
16        -- Environment and medium wrappers
17        par
18          Wrapper_Buffer [..., R_move, S_ok]
19          || Wrapper_Control [Pos, Safe]
20          || Wrapper_Data [Order1, Order2]
21          || Wrapper_Backup [Start]
22          || Activation [Start]
23        end par
24      end par
25    end hide
26  end process
27
28  process Activation [Start:Block] is
29      var Block:Block in
30        loop
31          Start (?Block) where (Block == Aileron)
32        end loop
33      end var
34  end process
```

The root process is parametrized by gates corresponding to the GRL channels whose variables are declared as formal parameters of the GRL system (line 2, LNT code). Those gates are therefore visible in the LTS. Gates corresponding to the GRL channels whose variables are declared as temporary variables of the GRL system are declared inside the root process using the **hide** construct (line 3, LNT code). Those gates are not visible in the LTS. The gate *Finish* is by default hidden since it is used only to release the *Mutex* and does not contain useful information about block execution.

Inside the root process, wrapper processes corresponding to GRL components are composed using parallel composition (**par**). The set of gates (called *synchronization set*) on which processes should synchronize has to be explicitly specified.

Wrapper processes of highest-level blocks are composed in pure interleaving, i.e., inside a parallel composition without synchronization set (lines 9-13, LNT code). This reflects the fact that blocks cannot interact directly with each other. In particular, they do not synchronize with each other on their common gates *Start* and *Finish*. This parallel composition is itself encapsulated inside a higher-level parallel composition to synchronize with process *Mutex* on gates *Start* and *Finish* (lines 5-14, LNT code).

Similarly to blocks, wrapper processes of mediums and environments are composed in pure interleaving (lines 17-23, LNT code). Then, all components are composed in parallel with synchronizations on gates corresponding to GRL common channels (lines 4-24). In particular, gates corresponding to unconnected channels of environments and mediums belong to the synchronization set. This way, no synchronization can happen on those gates, since they are not used in other components. To enable the activation of blocks that are not constrained by environments, we introduce an additional process *Activation* which proposes synchronizations on gate *Start* for those blocks. In our example, only *Aileron* is considered in process *Activation* (lines 28-34) since *Prim* and *Sec* are constrained by environment *Backup*.

Although both GRL and LNT have formal semantics, we have not yet proven formally the correctness of the translation. This would be a useful but long task due to the size of the language, which is far from a toy language. For this reason, we leave this for future work. However, we gave a number of arguments in this section about the intuition why the translation is correct. This intuition has been complemented by tests using the tools that will be presented in the next section.

6.5. Tool Support

A tool named GRL2LNT has been 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. Additionally, a second tool named GRL.OPEN has been developed, which encapsulates GRL2LNT and calls the LNT.OPEN tool, to connect GRL to all the on-the-fly verification tools of CADP.

GRL2LNT and GRL.OPEN have been tested on a benchmark of about 120 GRL specification files totalizing about 7,000 lines of code. Some specifications include external C and LNT code, as supported by GRL. The generated files consist of about 18,000 lines of LNT code, each LNT file being on average 2.5 times larger (in lines of code) than the GRL file. This linear expansion is mainly caused by the translation of GRL constructs into more than one LNT construct. This illustrates the fact that the level of abstraction at which GRL is specified is closer to the user's view as opposed to writing it straight in LNT.

During the development of the GRL2LNT translator, an extensive amount of GRL examples were written to perform "unit testing" of the language construct. At least two examples were written for each GRL syntactic and static semantic rule. The first example violates the rule in order to check that GRL2LNT captures the error. The second example, which is a corrected version of the first one, aims to check that no error is raised by GRL2LNT. More elaborated examples were written to cover very different aspects of the language. First, the generated LNT programs are analysed manually to check their conformance with the defined translation scheme. 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). Besides, our industrial partners also tested the GRL2LNT translator by generating GRL models automatically from their synchronous programming software and checking properties on the resulting LTSs, which is another form of validation of 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 execution and the occurrence of inputs before outputs in each execution.

7. Functional Verification by Model Checking

In this section, we illustrate briefly how GRL programs can be analysed by model checking, using the CADP verification toolbox. We use the GRL.OPEN and the GENERATOR tool of CADP to build state spaces of GRL programs. To illustrate different verification scenarios, we take advantage of SVL (*Script Verification Language*)[GL02], a high-level interface to all CADP tools, enabling an easy description and automatic execution of complex verification scenarios. We identify some correctness properties and specify them in the MCL language.

MCL enables a concise formulation of temporal properties, especially when these properties are parameterized by data values, such as the data carried by block channels. MCL is built from three kinds of formula. First, an *action formula* A characterizes actions (transition labels) of the LTS, which contain a gate name G followed by a list of values $v_1, ..., v_n$ exchanged during the rendezvous on G. An action formula is built from action patterns and the usual boolean connectors. An action pattern of the form "$\{G\ ?x{:}T\ !e$ **where** $b(x)\}$" matches every action of the form "$G\ v_1\ v_2$" where v_1 is a value of type T that is assigned to variable x, v_2 is the value obtained by evaluating the expression e, and the boolean expression $b(v_1)$ evaluates to true. Arbitrary combinations of value matchings ("$!e$") and value extractions ("$?x{:}T$") are allowed for matching actions containing several values. All variables assigned by value extraction are exported to the enclosing formula. Gate names G can also be extracted and manipulated as ordinary values of type String.

Second, a *regular formula* R characterizes sequences of transitions in the LTS. A regular formula is built from action formulas and (extended) regular expression operators such as concatenation ("$R_1.R_2$"), choice ("$R_1|R_2$"), and unbounded iterations ("R^*" and "R^+").

Third, a *state formula* F characterizes 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 operator ("$< R >$ `@`").

Deadlock analysis. The absence of deadlock ensures that the system continues to progress, i.e., its various components continue to execute. A deadlock can be either *global* or *local*:

- A global deadlock involves all blocks and is defined by a state from which no block is able to execute anymore. In terms of LTSs, this is a sink state, i.e., a state without any successor.
- A local deadlock involves one block B and is defined by a state from which the block cannot execute anymore. In terms of LTSs, this is a state from which no action labeled *Start !B* is possible.

Example 7.1. The following SVL script describes a verification scenario for deadlock analysis in the FCS model. Since we focus on the activation of blocks rather than the data handled by block channels, one can reason only on actions *Start*. We first generate the LTS of the system (line 1), which contains 439 states and 555 transitions. Then, we hide all actions, except actions *Start* (line 2). We reduce the LTS modulo branching bisimulation [vGW96] to remove hidden actions while preserving the branching structure of the LTS (line 4). Finally, we rename all transitions of the obtained LTS by removing the prefix "*Start !*" to simplify the presentation (line 5). The final LTS, depicted in Figure 8, corresponds to the activation policy of our system.

Property *Global_Deadlock* (lines 7-13) checks for the absence of global deadlock in the behaviour of the FCS. The text of the property construct is a statement calling the *EVALUATOR4.0* model checker. It states that the model given in "FCS.bcg" should satisfy the MCL property "$[true*] < true > true$", which verifies that each state in the LTS has a successor (line 11). The verification result is expected to be true (line 12). If the property is not satisfied, a counterexample is given in file "Global_Deadlock.bcg" (line 10). In the FCS model, the property evaluates to true, meaning that the system does not contain sink states.

Property *Prim_Deadlock* (lines 15-21) checks for the absence of local deadlock in block *Prim*, ensuring that from each state, there exists a sequence of actions leading to a *Prim* action. The property is not satisfied and a counterexample is given in file "Prim_Deadlock.bcg" (line 18), illustrating a local deadlock in block *Prim*.

```
1  % grl.open FCS.grl generator FCS.bcg
2  ...
3  "FCS.bcg" = partial hide all but ".*Start.*" in "FCS.bcg";
4  "FCS.bcg" = branching reduction of "FCS.bcg";
5  "FCS.bcg" = total rename "Start !\(.*\)" -> "\1" in "FCS.bcg";
6
7  property Global_Deadlock
8       "Check global deadlock freedom in the FCS behaviour"
9  is
10     "Global_Deadlock.bcg" =
11     "FCS.bcg" |= [true*] <true> true ;
12   expected TRUE
13 end property
14
15 property Prim_Deadlock
16     "Check deadlock freedom in Prim behaviour"
17 is
18     "Prim_Deadlock.bcg" =
19     "FCS.bcg" |= [true*] <true*.Prim> true ;
20   expected FALSE
21 end property
22
23 property Control
24     "Check Aileron is always under control of Prim or Sec"
25 is
26     "Prim_Deadlock.bcg" =
27     "FCS.bcg" |= not <(not (Prim or Sec))*.Aileron> @ ;
28   expected FALSE
29 end property
```

More properties on the activation policy can be expressed. For example, property *Control* (Ex. 7.1, lines 23-29) checks the absence of "unfair" sequences in which *Aileron* executes indefinitely with the control of neither *Prim* nor *Sec*. Abstracting out data in the system and reasoning about block activations is useful when debugging large specifications. This helps to keep the state space small, making easy the comprehension of the system behaviour and helps bug detection in early stage of the verification process.

Inactivity. Although the system has no deadlocks (global or local), it is possible that one or more blocks execute indefinitely without doing anything useful. We call *block inactivity* (also called *livelock*) a state from

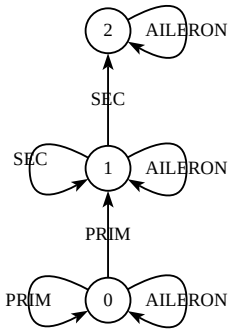


Fig. 8. Activation policy of the FCS

which all input/output and receive/send channels of the block continue to carry the same values, indefinitely. If all blocks are inactive, the whole system becomes inactive.

Example 7.2. The following SVL script describes a strategy to check the activity of block *Aileron* in the FCS system. Since we focus only on the behaviour of *Aileron*, we hide all actions relative to other blocks (lines 1-3). Then, we reduce the LTS modulo branching bisimulation (line 4). The reduced LTS contains 207 states and 270 transitions.

Property *Inactivity* (lines 6-29) checks that from each state, there exists a sequence of actions in which at least one channel of block *Aileron* continues to carry different values. The property is not satisfied in the FCS, and a counterexample, depicted in Figure 9, is given in file "Inactivity.bcg" (line 9). The cyclic sequence $10 \rightarrow 9 \rightarrow 16 \rightarrow 14$ illustrates an inactivity in the behaviour of *Aileron*.

```
1   "FCS.bcg" = partial hide all but
2                 ".*SAFE.*", ".*POS.*", ".*R_MOVE.*", ".*S_OK.*"
3               in "FCS.bcg";
4   "FCS.bcg" = branching reduction of "FCS.bcg";
5
6   property Inactivity
7       "Check inactivity in block Aileron"
8   is
9       "Inactivity.bcg" =
10      "FCS.bcg" |=
11      [true*]
12        ( <true*.{Safe ?safe1:bool}.
13          true*.{Safe ?safe2:bool where safe1 <> safe2}
14        > true
15        or
16        <true*.{Pos ?pos1:nat}.
17          true*.{Pos ?pos2:nat where pos2 <> pos1}
18        > true
19        or
20        <true*.{R_Move ?move1:string}.
21          true*.{R_Move ?move2:string where move2 <> move1}
22        > true
23        or
24        <true*.{S_Ok ?ok1:bool}.
25          true*.{S_Ok ?ok2:bool where ok1 <> ok2}
26        > true
27      ) ;
28    expected TRUE
29  end property
30
31  "Inactivity.bcg" = total branching reduction of "Inactivity.bcg";
```

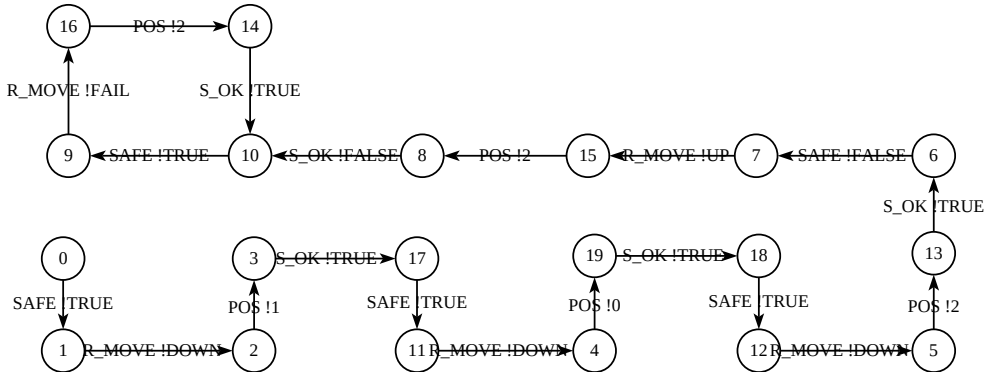


Fig. 9. Inactivity of block Aileron

Correct Message Transmission. To ensure the correctness of message transmission, one should check that each message emission over a send channel of a block is followed by the reception of the message over a receive channel of another block.

Example 7.3. Property *Correct_Transmission* ensures that all messages sent by *Prim* on channel *s_move1* are transmitted to *Aileron* on channel *r_move1*. The property is not satisfied and a counterexample is provided. It shows that block *Prim* can perform an arbitrary number of steps (at least two) sending value *Up*, before *Aileron* executes to receive the value.

```
1  property Correct_Transmission
2      ``Correct Transmission Message''
3  is
4      ``Correct_Transmission.bcg'' =
5      ``FCS.bcg'' |=
6      [true* . {S_MOVE1 ?msg:string}]
7          mu X . (<true> true and [not ({R_MOVE1 !msg})] X) ;
8    expected TRUE
9  end property
```

The main reason due to which the property is not satisfied is that blocks *Aileron* and *Prim* execute arbitrarily, i.e., no constraints have been put on their relative activation paces. This has induced discrepancies between the rates of message submission by *Prim* and message delivery to *Aileron* inside medium *Buf1* (Example 4.12, page 15), which has caused a loss of messages. A way to palliate such discrepancy, if divergence between submission and delivery rates (i.e., between block paces) is proved bounded, would be a buffering mechanism with well-chosen dimensions. In the simple case of quasi-synchrony (Example 4.9, page 13), at most two message submissions may occur between two message deliveries in each transmission, and conversely. A double-place FIFO is then sufficient to ensure message transmission without loss. More generally, FIFOs are very often used in interfacing synchronous components to obtain GALS systems, making the behaviour of the system deterministic.

However, the rates of message submission and delivery cannot always be deduced from the activation policy of the system. In such cases, a robust communication protocol should be implemented. GRL is sufficiently expressive to model various nondeterministic behaviours (e.g., unreliable mediums) and complex communication protocols. On the other hand, MCL is sufficiently expressive to capture complex properties of GALS systems, involving the succession of events in time (arbitrarily far from each other), the branching of execution, and the cycles denoting infinite executions.

8. Conclusion

We proposed an approach to the formal modelling and verification of GALS systems, intended to enhance industrial design process with asynchronous verification frameworks. Our approach is based on a DSL which

serves as an intermediate formal model from synchronous languages to asynchronous process calculi. We address a lack of approaches in industry dealing with asynchronous concurrency of GALS systems. This lack is at least twofold: (1) asynchronous concurrency is intrinsically more complex than synchronous concurrency and (2) asynchronous verification frameworks are expensive to integrate [Gar08]. We aim at keeping design flows as they are and enhancing them with both automatic connections to asynchronous verification frameworks and user-friendly analysis interfaces.

Our approach is based on GRL, a new textual language intended to model GALS systems with a focus on the asynchronous concurrency they involve. GRL combines synchronous features of dataflow imperative languages and asynchronous features of process algebras, and makes possible a modular description of synchronous components, environment constraints, and asynchronous communications. This makes the language sufficiently expressive and general-purpose to model a wide range of GALS architectures, possibly nondeterministic. The semantics of GRL are action-based, defined in terms of LTSs, in which transitions are labeled by events (interactions with the physical environment and the network). Such a representation is appropriate to specify possible interleavings between synchronous component executions.

We proposed a translation algorithm from GRL to LNT, one of the input languages of the CADP toolbox. The translation has been automated in the GRL2LNT tool, which has been tested on a large number of examples. This makes possible the analysis of GRL descriptions using the rich functionalities of CADP (e.g., simulation, verification, performance evaluation), focusing on the asynchronous behaviour of GALS systems. In particular, hardware/software co-simulation is possible by using the EXEC/CAESAR framework [GVZ01] of CADP, which enables the C code generated from a GRL description to be integrated with a physical platform. For various reasons (e.g., debugging, efficiency, abstraction), it would be difficult to automatically generate reliable and concise specifications in full-fledged process algebra, such as LNT.

The GRL approach has begun to be used in industry in the framework of the Bluesky project[7]. The project addresses the validation of networks of PLCs (*Programmable Logic Controllers*). Our industrial partner provides a software for the design of PLCs. The software is built upon a synchronous dataflow language with graphical syntax based on function block diagrams. Basically, after performing static analysis, encompassing causality analysis, the compiler generates executable code to be embedded on the PLC. The 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 for data constraints are automatically generated. Still, GRL mediums together with activation constraints should be encoded by hand by the engineers, at the time of writing. The reason is that the software does not yet support the design of multi-diagrams, enabling GALS design. The aim is to develop a catalogue of generic environments and mediums, that can be automatically generated from the software. The engineers of our industrial partner provided us with positive feedback concerning the use-friendliness of both the synchronous and asynchronous parts of GRL. Indeed, the GRL synchronous model is smoothly extended with asynchronous features fine-tuned for GALS, making asynchronous concurrency easy to learn and control.

Regarding ongoing work, we have defined a property specification language for GALS systems. The language builds upon a set of temporal logic patterns. The proposed patterns capture frequently encountered behaviours in the scope of GALS systems. This reduces the complexity of using full-fledged temporal logics, such as the MCL language of the CADP toolbox. Branching-time semantics in an action-based setting are considered. As interpretation model, the LTSs corresponding to GRL programs are considered. In addition, we are working to apply the GRL approach to real-life applications from the avionics industry based on GALS architectures.

Regarding future work, an interesting direction is the use of GRL as the target from other synchronous environments used in industry, such as Scade. GRL can also be used as an intermediate target language for GALS systems modeled in AADL-like architectural languages. Another direction is to investigate the connection of GRL as front-end of verification frameworks based on the synchronous paradigm so as to enable the analysis of GRL synchronous blocks. Moreover, we plan to prove formally the correctness of the translation from GRL to LNT.

Acknowledgements. We are grateful to Éric Léo for implementing the GRL2LNT translator and for valuable discussions. We are also grateful to our industrial partners in the Bluesky project for useful feedback

[7] `www.minalogic.com`

on the usage of GRL for describing GALS systems. Acknowledgements are due to the anonymous reviewers for their relevant comments which were helpful in improving the present article.

References

[BBC10] A. Benveniste, A. Bouillard, and P. Caspi. A Unifying View of Loosely Time-triggered Architectures. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, pages 189–198, New York, NY, USA, 2010. ACM.

[BBS12] Y. Bai, J. Brandt, and K. Schneider. Preservation of LTL properties in desynchronized systems. In *MEMOCODE*, pages 53–64. IEEE, July 2012.

[BCLG99] A. Benveniste, B. Caillaud, and P. Le Guernic. From Synchrony to Asynchrony. In JosC. M. Baeten and Sjouke Mauw, editors, *CONCUR'99*, volume 1664 of *LNCS*, pages 162–177. Springer, 1999.

[BCMW15] J. Backes, D. D. Cofer, S. P. Miller, and M. Whalen. Requirements Analysis of a Quad-Redundant Flight Control System. *CoRR*, abs/1502.03343, 2015.

[BÖM14] K. Bae, P.C. Ölveczky, and J. Meseguer. Definition, Semantics, and Analysis of Multirate Synchronous AADL. In C. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014*, volume 8442 of *LNCS*, pages 94–109, 2014.

[Bou98] A. Bouali. Xeve, an Esterel verification environment. In AlanJ. Hu and MosheY. Vardi, editors, *CAV*, volume 1427 of *LNCS*, pages 500–504. Springer Berlin Heidelberg, 1998.

[BRS93] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating Reactive Processes. In *Proc. of POPL*, pages 85–98. ACM Press, 1993.

[BS01] G. Berry and E. Sentovich. Multiclock Esterel. In *Proc. of CHARME*, volume 2144 of *LNCS*, pages 110–125. Springer, 2001.

[CCG+14] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.1). INRIA/VASY and INRIA/CONVECS, 131 pages, August 2014.

[Cha84] D. M. Chapiro. Globally-Asynchronous Locally-Synchronous Systems. Technical report, DTIC Document, October 1984.

[CMP01] P. Caspi, C. Mazuet, and N. Paligot. About the Design of Distributed Control Systems: The Quasi-Synchronous Approach. In Udo Voges, editor, *Computer Safety, Reliability and Security*, volume 2187 of *LNCS*, pages 215–226. Springer Berlin Heidelberg, 2001.

[DMK+06] F. Doucet, M. Menarini, I.H. Krüger, R. Gupta, and J.-P. Talpin. A verification approach for GALS integration of synchronous components. *ENTCS*, 146(2):105–131, 2006.

[Gar08] H. Garavel. Reflections on the future of concurrency theory in general and process calculi in particular. *ENTCS*, 209:149–164, 2008.

[GG03] A. Gamatié and T. Gautier. The signal approach to the design of system architectures. In *10th IEEE International Conference on Engineering of Computer-Based Systems, ECBS 2003, Huntsville, AL, USA*, pages 80–88. IEEE, 2003.

[GG07] M. K. Ganai and A. Gupta. Efficient BMC for multi-clock systems with clocked specifications. In *Design Automation Conference*, pages 310–315. IEEE, 2007.

[GG10] A. Gamatié and T. Gautier. The signal synchronous multiclock approach to the design of distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 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).

[GL02] H. Garavel and F. Lang. SVL: a scripting language for compositional verification. In *Formal Techniques for Networked and Distributed Systems*, IFIP Conference Proceedings, pages 377–392. Springer, 2002.

[GLM02] H. Garavel, F. Lang, and R. 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 *LNCS*, pages 9–13, April 2002.

[GLM15] H. Garavel, F. Lang, and R. Mateescu. Compositional Verification of Asynchronous Concurrent Systems Using CADP. *Acta Informatica*, 52(4):337–392, April 2015.

[GLMS13] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107, 2013.

[GT09] H. Garavel and D. 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'2009 (Grenoble, France)*, volume 5578 of *LNCS*, pages 241–260, June 2009.

[GVZ01] H. Garavel, C. Viho, and M. Zendri. System design of a CC-NUMA multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. *STTT*, 3(3):314–331, 2001.

[Hal13] N. Halbwachs. *Synchronous programming of reactive systems*, volume 215. Springer Science & Business Media, 2013.

[HB02] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, volume 2491 of *LNCS*, pages 240–251, Grenoble, October 2002. Springer.

[HLR93a] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous Observers and the Verification of Reactive Systems. In *AMAST'93, Twente*, pages 83–96. Springer, June 1993.

[HLR93b] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HM06] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Proc. of ACSD*, pages 3–14. IEEE, June 2006.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[ISO01] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.

[JLM14a] F. Jebali, F. Lang, and R. Mateescu. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems. *Proc. of ICFEM*, 8829:219–234, 2014.

[JLM14b] F. Jebali, F. Lang, and R. Mateescu. GRL: A specification language for Globally Asynchronous Locally Synchronous systems (syntax and formal semantics). Research report RR-8527, INRIA, 2014.

[LGTLL03] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(03):261–303, 2003.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical computer science*, 25(3):267–310, 1983.

[Mil89] R. Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[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] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *Proc. of FM*, LNCS, pages 148–164. Springer, 2008.

[MWO+05] S.P. Miller, M.W. Whalen, D. O'Brien, M.P. Heimdahl, and A. Joshi. A methodology for the design and verification of globally asynchronous/locally synchronous architectures. *National Aeronautics and Space Administration, Langley Research Center*, 2005.

[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.

[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.

[Ram98] S. Ramesh. Communicating reactive state machines: Design, model and implementation. In *IFAC Workshop on Distributed Computer Control Systems*, 1998.

[RSD+04] S. Ramesh, Sampada Sonalkar, Vijay Dsilva, Naveen Chandra R., and B. Vijayalakshmi. A Toolset for Modelling and Verification of GALS Systems. In R. Alur and D. A. Peled, editors, *Proc. of CAV*, volume 3114 of *LNCS*, pages 506–509. Springer, 2004.

[Sme13] G. Smeding. *Verification of Weakly-Hard Requirements on Quasi-Synchronous Systems*. Theses, Université de Grenoble, December 2013.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[vGW96] R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM*, 43(3):555–600, May 1996.