HAL
open science

GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems

Fatma Jebali, Frédéric Lang, Radu Mateescu

To cite this version:

Fatma Jebali, Frédéric Lang, Radu Mateescu. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems. Proceedings of the 16th International Conference on Formal Engineering Methods (ICFEM'14), Nov 2014, Luxembourg, Luxembourg. pp.219-234, 10.1007/978-3-319-11737-9_15 . hal-01082348

HAL Id: hal-01082348

https://inria.hal.science/hal-01082348v2

Submitted on 8 Feb 2016

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

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

GRL: a Specification Language for Globally Asynchronous Locally Synchronous Systems*

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 subsystems that evolve concurrently and interact with each other asynchronously. Most formalisms and design tools support either the synchronous paradigm or the asynchronous paradigm but rarely combine both, which requires an intricate modeling of GALS systems. In this paper, we present a new language, called GRL (*GALS Representation Language*) designed to model GALS systems in an abstract and versatile manner for the purpose of formal verification. GRL has formal semantics combining the synchronous reactive model underlying dataflow languages and the asynchronous concurrent model underlying process algebras. We present the basic concepts and the main constructs of the language, together with an illustrative example.

1 Introduction

Computer science has led to new generations of heterogeneous systems called GALS (*Globally Asynchronous, Locally Synchronous*). A GALS system is composed of several synchronous subsystems, executing and interacting in asynchronous concurrency: no assumption is made, neither on the relative frequency of each subsystem, nor on the communication delays between subsystems. Each subsystem is composed of several components running together synchronously, all governed by a single clock and encompassing the zero-delay assumption: computations and communications between components are instantaneous (these are called the synchronous assumptions). As such, GALS systems involve a high degree of synchronous and asynchronous concurrency (introducing nondeterminism), which requires tedious effort to design and debug. Formal modeling and verification is then a crucial part in the design process of such usually safety-critical systems.

Many different approaches have been proposed for GALS modeling and verification. Some propose to model GALS systems in synchronous frameworks (such as Signal [26]) directly or to extend synchronous languages with an asynchronous layer (Multiclock Esterel [4], CRSM [29]). Such representations are well-suited

* This work was partly funded by the French *Fonds unique interministériel* (FUI), Pôle Minalogic (project "Bluesky for I-Automation").

to hardware-based subsystems distributed on one single hardware platform, each with its own clock.

Other approaches, conversely, extend asynchronous languages to incorporate synchronous features. The most common approach is to surround each locally synchronous subsystem by an asynchronous wrapper, which provides an asynchronous interface to other subsystems. This way, GALS systems can be modeled and verified in asynchronous frameworks. In [9], Signal modules are translated to Promela, the input language of the SPIN model checker [20]. In [13], Sam synchronous programs are represented by Mealy functions without internal state and are encapsulated into wrappers modeled in LNT [7], a language of asynchronous concurrent processes inheriting process algebraic concepts and extended with data and the control structures of classical algorithmic programming. LNT is equipped with the CADP verification toolbox [11], which comprises tools for visual checking, model checking, and equivalence checking. Previously, LNT had been used successfully for the analysis of other GALS systems [8, 12, 22].

All the aforementioned approaches adopt specific techniques of a specific paradigm (synchronous or asynchronous) to accommodate GALS systems. On the one hand, synchronous frameworks are deterministic by nature, and not appropriate to model asynchrony. On the other hand, asynchrony and nondeterminism are granted for free in asynchronous frameworks, but those lack built-in constructs dedicated to synchronous components, which guarantee that system models fulfill the synchronous assumptions.

Moreover, most existing approaches depend strongly on the considered application field (e.g., distributed control systems [31], FPGA and ASIC digital designs [6, 28], or networks on chip), the target platform (e.g., software [24], hardware [9], or heterogeneous), and the preferred specification methods (e.g., based on Petri nets [27], automata [16], process algebras [13]). This narrows down the range of systems that can be addressed. On the other hand, the surge in complexity of GALS systems forces designers to tackle (among others) design concepts, synchronous and asynchronous computations, deterministic and nondeterministic behaviour, and verification approaches, which makes the design of such systems increasingly challenging.

To circumvent this complexity, an appealing trend has been to design new languages dedicated to GALS system modeling [3, 24, 5], which enforce the assumptions of the GALS paradigm. In this paper, we propose GRL (*GALS Representation Language*), a new specification language with textual syntax and formal semantics, targeting systems consisting of a network of distributed synchronous systems (called blocks) that interact with their environments and exchange data asynchronously via communication mediums. The design of GRL has originally been driven by the need of general-purpose, designer-friendly, and formal representation of GALS systems suitable for efficient verification.

Our approach draws mainly from two semantic foundations. As regards synchrony, GRL holds a dataflow-oriented model based on the block-diagram model, widely used in industry: synchronous components are modeled by blocks connected together hierarchically to build higher-level blocks. Therefore, the GRL

synchronous model inherits from the simplicity and modularity of the block-diagram model. As regards asynchrony, GRL was inspired by process algebras, and more particularly by LNT: blocks exchange data by (implicit) rendezvous synchronization with communication mediums connected to other blocks, and the interactions between blocks and their environments work similarly. Thereby, GRL leverages process algebra expressiveness, versatility, and verification efficiency, with a specialization to the GALS paradigm.

GRL was designed with several concerns in mind. First, it provides a sufficiently high abstraction level to fit a wide range of applications, independently from both the target platform (hardware, software, or heterogeneous), the architecture (single or distributed platforms), and the application domain.

Second, GRL is aimed at being a pivot language between industrial design tools (in particular those based on function block diagrams for the synchronous part) and verification tools for both synchronous and asynchronous systems, which guarantee system reliability and correctness. This way, we hope that formal verification methods – claimed traditionally to require high level expertise in theoretical issues – are easier to learn by industrial users, without requiring companies to shift from their actual tools and languages to entirely new production approaches. Indeed, although some approaches seem efficient [15], the high cost of such a shift makes it unlikely to happen in the near future.

Last but not least, GRL is intended to have a user-friendly syntax as it does not require users to have solid background in neither synchronous programming (e.g., clocks are not modeled explicitly), asynchronous concurrent programming (e.g., parallel composition and synchronization), nor formal verification methods. All features are smoothly and tightly integrated to form a language with homogeneous syntax and semantics.

In this paper, we introduce GRL as a first step towards fully-automated verification of GALS systems. It is organized as follows. Section 2 presents some related work. Section 3 presents the language, its formal semantics, and the current status of software tools. Section 4 gives an illustrative GRL model of an aircraft flight control system used in the avionics industry. Finally, Section 5 summarizes the paper and indicates directions for future work.

2 Related work

In this section, we review the languages combining synchronous and asynchronous features. CRP [3] combines the Esterel [2] synchronous language and the CSP [19] asynchronous language. Despite its mathematical elegance, CRP is still rarely used in industry since it requires the user to have expertise in both Esterel and CSP. Such expertise is not required for GRL, which was designed to facilitate industrial GALS design. A language close to CRP is SystemJ [24], which extends Java with Esterel-like synchronous model and CSP-like asynchronous model. SystemJ allows efficient code to be generated automatically. However, it lacks rigorous support for fully-automated formal verification and is not suitable for systems with limited resources because of its reliance on Java virtual

machine as target. To the contrary, GRL is intended to be general-purpose and verification-oriented. Action Language [5] is a state-based approach, which aims at bridging the gap between high specification languages (Statecharts [17], SCR [18], and RSML [23]) and the SPIN model checker. A key difference between this approach and ours is that Action Language adopts a low-level condition/action model whereas GRL is equipped with high-level control structures making GRL models clearer and more structured.

3 The GRL Language

The syntax and semantics of GRL are formally described in a research report [21] (76 pages). In this section, we present them briefly and informally. Figure 1 is a simplified presentation[1] described in EBNF (*Extended Backus-Naur Form*), where square brackets denote optional syntactic parts and vertical bars denote alternatives. The symbols K, X, and E denote respectively literal constants, variables, and expressions (built upon constants, variables, and function applications). The symbols S, B, N, M, T, and f denote respectively system, block, environment, medium, type, and record field identifiers.

3.1 Overview

GRL specifications are structured in modules, called *programs*. Each program can import other programs, which promotes code organization and reuse. A GRL program contains the following constructs:

1. *types*, ranging from predefined types (such as Booleans and naturals) to user-defined types (such as arrays and record types),
2. *named constants*, visible by all other constructs,
3. *blocks*, representing the synchronous components,
4. *mediums* and *environments*, representing respectively communication mediums and physical or logical constraints on block inputs, and
5. *systems*, representing the composition and interactions of blocks, environments, and mediums.

In the sequel, these five constructs are called *entities*, and blocks, environments, and mediums are called *actors*.

As regards synchronous behaviours, blocks are the synchronous composition of one or several subblocks, all governed by the clock of the highest level block. A block performs a sequence of discrete deterministic steps and preserves an internal state, hereafter called *memory*. At each step (each cycle of the clock), it consumes a set of inputs, computes a reaction instantaneously, produces a set of outputs, and updates its memory. Within one block (i.e., at *actor level*), connections between subblocks are carried out using parameters in modes "**in**" (input) and "**out**" (output). Every output parameter can be connected to several

[1] 70 EBNF productions were necessary to present the full language in [21]

```
system ::= system S [(X0 : T0 , ... , Xm : Tm)] is
             allocate actor0 , ... , actorn [temp X'0 : T'0 , ... , X'l : T'l]
             network block_call0 , ... , block_callp
             [constrainedby env_call0 , ... , env_callq]
             [connectedby med_call0 , ... , med_callr]
           end system
block ::= block B [[const_param]][(inout_param0 ; ... ; inout_paramm)]
                                 [{com_param0 ; ... ; com_paramn}] is
             [allocate sub_block0 , ... , sub_blockp] [local_var0 , ... , local_varl]
             I
           end block
        | block B [[const_param]][(inout_param0 ; ... ; inout_paramm)] is
             !c string | !lnt string
           end block
env ::= environment N [[const_param]][(inout_param0 | ... | inout_paramm)] is
             [allocate sub_block0 , ... , sub_blockn] [local_var0 , ... , local_varl]
             I
           end environment
med ::= medium M [[const_param]][{com_param0 | ... | com_paramm}] is
             [allocate sub_block0 , ... , sub_blockn] [local_var0 , ... , local_varl]
             I
           end medium
const_param ::= const X0 : T0 [:= E0] , ... , Xn : Tn [:= En]
inout_param ::= (in | out) X0 : T0 [:= E0] , ... , Xn : Tn [:= En]
com_param ::= (send | receive) X0 : T0 , ... , Xn : Tn
local_var ::= (perm | temp) X0 : T0 [:= E0] , ... , Xn : Tn [:= En]
sub_block ::= B [[arg0 , ... , argn]] as Bi
actor ::= B[[arg0 , ... , argn]] as Bi | N[[arg0 , ... , argn]] as Ni
        | M[[arg0 , ... , argn]] as Mi
block_call ::= Bi [(arg(0,0) , ... , arg(0,m0) ; ... ; arg(n,0) , ... , arg(n,mn))]
             | Bi [(arg(0,0) , ... , arg(0,m0) ; ... ; arg(n,0) , ... , arg(n,mn))]
                  [{arg'(0,0) , ... , arg'(0,p0) ; ... ; arg'(q,0) , ... , arg'(p,pq)}]
env_call ::= Ni (arg(0,0) , ... , arg(0,m0) | ... | arg(n,0) , ... , arg(n,mn))
med_call ::= Mi {arg(0,0) , ... , arg(0,m0) | ... | arg(n,0) , ... , arg(n,mn)}
signal ::= on [?]X0 , ... , [?]Xn -> I
I ::= null | X :=E | X [E0] :=E1 | X .f :=E | I0 ; I1 | Bi(arg0 , ... , argn)
    | if E0 then I0 elsif E1 then I1 ... elsif En then In else In+1 end if
    | while E loop I0 end loop | for I0 while E by I1 loop I2 end loop
    | case E is K0 -> I0 | ... | Kn -> In | [any -> In+1] end case
    | X := any T [where E] | select I0 [] ... [] In end select | signal
```

Fig. 1: The syntax of GRL (excerpts)

input parameters of different blocks; however, an input parameter can be connected to only one output parameter of another block. Such connections describe synchronous communication by instantaneous broadcasting.

As regards asynchronous behaviours, blocks are composed, together with environments and mediums, within systems to form networks of distributed connected synchronous subsystems. Within a GRL system (i.e., at *system level*), the separate blocks execute asynchronously, i.e., each block evolves cyclically at its own frequency (blocks have independent clocks). Blocks interact with each other across mediums, which allows separate blocks to be loosely coupled so that communication is performed asynchronously (i.e., takes an arbitrary amount of time). Connections between blocks and mediums are carried out using parameters in modes "**receive**" and "**send**". Receive parameters of mediums can be connected to send parameters of blocks, and conversely. Such connections describe synchronisation and communication by message-passing rendezvous be-

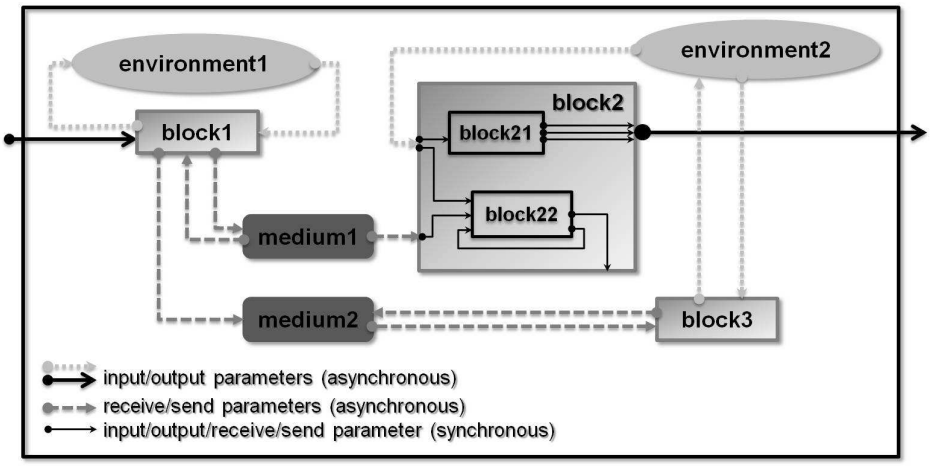

Fig. 2: Schematic representation of a GRL system

tween blocks and mediums. Mediums may exhibit nondeterministic behaviour, a key feature for asynchronous systems modeling and compositional specification; this provides descriptions with accuracy and high abstraction capability.

Blocks behaviour can also be constraint-driven by a collection of user-defined environments. In essence, environments exhibit a similar behaviour as mediums, except that their connections to blocks are carried out using modes "**in**" and "**out**". They have been introduced in GRL to explicitly separate the specification of communication mediums from external constraints imposed by the environment; this contributes to provide more comfort and insight about the system composition. With such a composition, we seek enhanced user-convenience to smoothly and tightly tailor complex network topologies, environment requirements and constraints, as well as communication protocols.

As an example, GRL can be used to model the network topology depicted in Figure 2: *block1* is constrained by *env1* and communicates with *block2* and *block3* respectively across *med1* and *med2*; *block2* and *block3* are both constrained by *env2*; *block2* is the synchronous composition of two subblocks.

3.2 Block

Blocks, defined by the non-terminal *block* in Figure 1, are the central building unit of the language. At actor level, a block is defined by the following elements:

- a list "*const_param*" of typed constant parameters (read-only variables).
- a list "$inout_param_0$ **,** ... **,** $inout_param_m$" of typed input and output parameters preceded by their mode ("**in**" or "**out**").
- a list "$local_var_0$ **,** ... **,** $local_var_l$" of local variables. Temporary variables are declared using the keyword "**temp**"; their values are lost when the current block execution terminates. Permanent variables are declared using the

keyword "**perm**"; their values are kept until the next execution cycle. The *memory* of the block is the list of values assigned to its permanent variables.
- a list "sub_block_0 **,** ... **,** sub_block_p" of subblock *allocations*, which enables subblock instances to be created, each maintaining its own memory. This concept is inherited from the block-diagram model, which is similar to, but simpler than, the class-instance paradigm in object-oriented languages.
- a body "I" expressed as a deterministic statement defined by combination of high-level control structures (bounded loops, if-then-else, sequential composition, etc.) and synchronous subblock invocations. The scheduling of subblock executions is inherently specified by the order in which the subblocks are invoked, using the sequential composition operator "**;**". Nondeterministic statements such as "X **:= any** T **where** E" and "**select**" (arbitrary choice of one statement among a set) are forbidden within block bodies.

Alternatively, blocks can be specified in an external language. Their body consists of a pragma denoting the language in which the external function implementing the block is written, followed by the name of the function in the external code. So far, the supported external languages are C and LNT: an external function identifier in C (resp. LNT) is preceded by the pragma "**!c**" (resp. "**!lnt**"). Although C external blocks provide more flexibility for the user, they should be defined to comply with the GRL block semantics (in particular, side effects in external C code are prohibited to enable model checking). LNT external blocks, however, have formal semantics and can thus be used safely.

In a block invocation, actual parameters have different forms according to their modes. A question mark precedes both output and send actual parameters, meaning that the parameter will have a value assigned when returning from the block. An underscore ("_") is used for unconnected parameters (i.e., unused inputs or outputs). An output parameter X_i declared in "**out** $X_0 : T_0$ **,** ... **,** $X_n : T_n$" of a block Bi and an input parameter Y_j declared in "**in** $Y_0 : T_0$ **,** ... **,** $Y_m : T_m$" of a block Bi' can thus be connected synchronously using a variable "$Z : T_i$" by passing "**?**Z" to Bi and "Z" to Bi' in a subsequent invocation.

Additional elements can be used to define a block that can only be invoked at system level, namely lists "com_param_0 **,** ... **,** com_param_n" of typed receive and send parameters preceded by their mode "**receive**" or "**send**". They enable blocks to interact asynchronously within a network of blocks via mediums. Such a block cannot be allocated nor invoked inside another actor since communication between blocks within actors is necessarily synchronous. At system level, actual input parameters of blocks can have the additional form "**any** T", meaning that an arbitrary value of type T is passed as input to the block.

The behaviour of a block is the following. In each cycle of its clock, (1) the block consumes data received over input and receive parameters, (2) the block computes by executing its body, then (3) the block produces data sent over output and send parameters. During computation, its memory is assigned the updated values of permanent variables so as to keep them stored up to the next cycle. As usual in the synchronous paradigm, all these steps are performed in zero-delay, i.e., instantaneously and atomically.

3.3 Medium

Mediums, defined by the non-terminal *med* in Figure 1, are dedicated to the modeling of communications and asynchronous interactions within a network of synchronous blocks. A medium is defined by the following elements:

- a list "*const_param*" of constant parameters.
- a list "com_param_0 **,** ... **,** com_param_m" of send and receive parameters.
- a list "sub_block_0 **,** ... **,** sub_block_p" of subblock allocations enabling blocks to be used in mediums in the same way as functions in programming languages.
- a list "$local_var_0$ **,** ... **,** $local_var_l$" of local (temporary and permanent) variable declarations.
- a body "I" expressed as a statement (not necessarily deterministic) defined as a combination of high-level control structures, subblock compositions, and nondeterministic statements.

A medium sends and receives messages to and from several blocks. When a block wants to send a message to or receive a message from a medium, it triggers the execution of the medium, which we call medium *activation*. Therefore, the invocation of mediums is demand-driven by different blocks at unpredictable instants. In this respect, mediums are *passive actors*, whereas blocks are *active actors*. Each medium is activated during a block execution cycle at most once to send messages to the block, then at most once to receive messages from the block. Since several messages may have to transit via one medium, those messages are grouped in tuples, called *channels*, all messages of a channel being exchanged within a single block-medium interaction. The channel under consideration is then called *activated*. The activations of a given medium are thus guided by the separate activations of its channels, as is suggested by the pipe symbol ("`|`") used to delimit formal and actual channel parameters, each channel activation leading to a separate execution of the medium.

To control medium activations, we introduce *signal statements*, whose syntax is defined by the non-terminal *signal* in Figure 1. A signal guards the part of the medium code that needs to be executed upon the activation of a particular channel. When a channel of the form "**receive** X_0 **,** ... **,** X_n" is activated, the signal statement "**on** X_0 **,** ... **,** X_n `->` I" can be executed and the values of variables X_0 **,** ... **,** X_n passed to the channel can be read within the statement I. When a channel of the form "**send** Y_0 **,** ... **,** Y_m" is activated, the signal statement "**on** `?`Y_0 **,** ... **,** `?`Y_m `->` I" can be executed and the statement I must assign values to the variables $Y_0, \ldots, Y_m$. Static semantics prohibit sequential composition of signals, loop statements containing signals, and nested signals, so that at most one signal is present on each execution path.

Mediums introduce flexibility in system models since they provide an accurate design of complex network topologies (e.g., bus, star, ring, mesh), connection modes (e.g., point-to-point, multi-point), as well as communication protocols. This way, we address a lack identified in existing languages confined to rigid topologies and point-to-point communications between separate synchronous subsystems, such as those based on CSP rendezvous [24, 3, 9]. Limita-

tions of adopting point-to-point communications in GALS models are considered as drastically restrictive to design complex networks of arbitrary topologies [30].

3.4 Environment

Since synchronous systems are often recognized to be outside-aware, GRL allows the user to model explicitly, yet abstractly, the behaviour of the environment. There are two major roles the environment can play: impose outside physical and logical requirements that block inputs may undergo, and put constraints on the scheduling of blocks executing in parallel. Additionally, an environment may be local, i.e., connected only to one block, or global, i.e., connected to several blocks at the same time. Environments, defined by the non-terminal *env* in Figure 1, are syntactically and semantically very similar to mediums, except that send and receive parameters are replaced by input and output parameters.

3.5 System

Systems, defined by the non-terminal *system* in Figure 1, are the top level entities in GRL programs, within which actors are invoked and connected to each other. A system is defined by the following elements:

- a list "$X_0 : T_0 , \ldots , X_m : T_m$" of parameters, which can be used in actual channels to connect blocks to environments and mediums. They are the visible parameters of the system, observable from the outside world.
- a list "$actor_0 , \ldots , actor_n$" of actor instance declarations.
- a list "$X'_0 : T'_0 , \ldots , X'_l : T'_l$" of temporary variables, which can be used in actual channels to connect blocks to environments and mediums. They are the invisible parameters of the system, not observable from the outside world.
- a list "$block_call_0 , \ldots , block_call_p$" of block invocations.
- a list "$env_call_0 , \ldots , env_call_q$" of environment invocations that constrain the blocks.
- a list "$med_call_0 , \ldots , med_call_r$" of medium invocations that ensure the traffic inside the network of blocks.

All interactions between actors within a system are built on message-passing synchronisations (rendezvous). Since blocks are the active actors of systems, the scheduling of the whole system is focused around their executions. Blocks execute cyclically, each at its own frequency, and force environments and mediums to perform some operations by sending messages (requesting or providing data) through channels. Environments and mediums, consequently, are passive actors responding to arisen demands from different blocks. At system level, GRL prohibits blocks to be connected directly to each other in order to preserve an independent behaviour of each block and an asynchronous behaviour of the network. Thus, blocks communicate only indirectly across mediums.

Each interaction between two actors is performed through exactly one channel. Namely, an output channel of the form "**out** $X_0 : T_0 , \ldots , X_n : T_n$" of an

actor Ai and an input channel of the form "**in** $Y_0 : T_0$ **,** $\ldots$ **,** $Y_n : T_n$" of another actor Ai' can thus be connected using a set of variables "$Z_0 : T_0$ **,** $\ldots$ **,** $Z_n : T_n$" by passing $?Z_0, \ldots, ?Z_n$ to Ai and $Z_0, \ldots, Z_n$ to Ai', in their respective invocations within a system. Send and receive channels can be connected similarly.

The semantics of a system are the following. Blocks execute arbitrarily often and cyclically. Each time a block begins its execution cycle, all environments and mediums connected respectively to input and receive channels of the block are activated to provide the needed input and receive values. Unconnected input and receive channels are assigned arbitrary values. Then, the block executes its body, and thus updates its output and send channels as well as its memory. Finally, all environments and mediums connected to respectively output or send channels of the block are activated.

The combined execution cycle of a block, and its related environments and mediums is performed instantaneously according to the synchronous assumptions. As a consequence, a block is executed only if all its connected environments and mediums are able to respond to all input, output, receive, and send signals of the block.

3.6 Formal Semantics

The semantics of GRL are formally defined in [21]. They consist in 145 rules of static semantics and 24 rules of Plotkin-style structural operational semantics for the dynamic part. In this paper, we only sketch briefly the principles of the dynamic semantics, defined in terms of LTSs (*Labelled Transition Systems*). 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 memory of an actor, denoted by μ, is a partial function mapping all permanent variables of the actor and its subblocks to their current values. A state S of the system is the union of memories μ_i of all actors composing the system, and the initial state s_0 maps all permanent variables to their initialization values. Each label has the form $Bi(a_0, \ldots, a_n)\{a'_0, \ldots, a'_m\}$ with Bi a block identifier, $a_0, \ldots, a_n$ the visible actual parameters of input and output channels, and $a'_0, \ldots, a'_m$ the visible actual parameters of receive and send channels. A transition $\mu \xrightarrow{Bi(a_0,\ldots,a_n)\{a'_0,\ldots,a'_m\}} \mu'$ expresses the combined execution of one cycle of the block instance Bi, together with its connected environments and mediums. The semantics of the system are obtained by interleaving all possible block executions. Verification (e.g., visual checking, equivalence checking, model checking) can be done by inspection of the LTS.

3.7 Tools for GRL

There are currently two software tools for handling GRL models. The first one is a parser for GRL (2000 lines), developed using the SYNTAX and Lotos NT compiler construction technology [10], which performs lexical and syntax analysis, type checking, binding analysis, and variable initialisation analysis of GRL programs.

The second one, named GRL2LNT (8000 lines), is an automated translator from GRL to LNT. Each block is mapped to an LNT function that takes inputs and produces outputs. Its permanent variables are mapped to *inout* parameters, i.e., parameters whose values are updated during function invocation. Synchronous block composition is mapped to sequential composition. Additionally, each actor invoked within a system is also mapped to a *wrapper* process, which contains communication actions to exchange data with its connected actors. The whole system is mapped to the LNT parallel composition of the wrapper processes with appropriate synchronizations of the communication actions.

GRL and GRL2LNT play the role of an intermediate and appropriate layer of abstraction and compositionality to provide generated LNT code with accuracy and conciseness, since scalability of automated model checking is limited. We can take advantage from the CADP toolbox available for LNT to build state spaces and apply visual, equivalence, and model checking techniques.

4 Example: Flight Control System

Our aim here is not to present a full case study, but rather to illustrate the main concepts of GRL via a feature rich example: the aircraft Flight Control System (FCS)[2], whose role is to control the aircraft turning and which is one of the most critical systems inside new generations of Airbus aircraft designs. Subsets of FCS have been studied at different levels of abstraction. In [25], the Flight Guidance System component of the FCS has been studied as a composition of synchronous systems following a single-platform GALS architecture. In [1], control systems have been studied as synchronous systems following a distributed-platforms PALS (*Physically Asynchronous, Locally synchronous*) architecture. For the sake of simplicity, we model the global behaviour, at a very high level of abstraction, of an FCS containing the following subsystems:

- Flight Control Surfaces adjust and control the aircraft's flight turning. We consider only one aileron (a flap attached to the end of a wing) controller.
- Fly-By-Wire Computers command the movement of the Flight Control Surfaces. We consider only two Fly-By-Wire Computers commanding the position of the aileron, one being used as a backup in case the other fails.
- A Flight Control Data Concentrator schedules the execution of Fly-By-Wire Computers and allows interaction with pilot displays.

The GRL model. The FCS system depicted in Figure 3 (see the GRL code below) consists of four block instances, whose cyclic behaviour is as follows:

- The *Ail* (for *aileron*) block instance receives movement requirements from the network, computes the next position of the aileron depending on its current position, then sends it to the network.

[2] `http://www.skybrary.aero/index.php/Flight_Control_Laws`

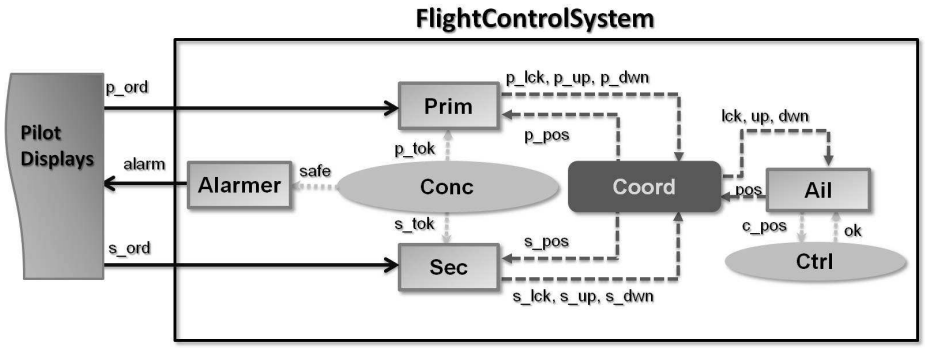

Fig. 3: Architecture of the Flight Control System

- The *Prim* (for *primary*) and *Sec* (for *secondary*) instances of block *FBW-Comp* (for *fly-by-wire computer*) receive: tokens from the environment indicating whether *Prim* or *Sec* should control *Ail*; an order from the pilot displays; and the current position of the aileron from the network. They compare the two latter values, and then send to the network the decision about whether the aileron should move up, move down, or not move (if the order matches the position).
- The *Alarmer* block instance checks whether the system is evolving safely by receiving from the environment a message indicating whether *Ail* is still controlled by either *Prim* or *Sec*, then informs the pilot about the safety state of the system.

Communications within the network of blocks are modeled by the medium *Coord* (for *coordinator*) as follows. *Prim* provides *Ail* with move requirements, then *Ail* achieves the required computations and sends its new position to *Prim*. *Sec* and *Ail* communicate similarly.

The environment constraints are modeled by two environments. The first environment *Conc* (for *concentrator*) ensures that either *Prim* or *Sec*, but not both, can control *Ail*, the priority being given to *Prim* by activating its token. *Conc* determines whether *Prim* is in a safety state (i.e., able to control *Ail*). Once *Prim* is not in safety state, it is considered out of order (not alive). Then, *Conc* blocks the execution of *Prim* by inactivating its token and gives the control of *Ail* to *Sec* by activating its token. Once *Sec* is not in safety state anymore, which means that neither *Prim* nor *Sec* is in safety state, then *Ail* is considered out of control and *Alarmer* warns the pilot.

The second environment *Ctrl* (for *controller*) constrains the execution of *Ail* as follows. *Ail* sends cyclically its new position to *Ctrl*. *Ctrl* verifies whether the position is still within a predefined interval, which means that *Ail* moves smoothly; if not, the execution of *Ail* is blocked. The GRL specification of the system follows.

```
system FlightControlSystem (p_ord:nat, s_ord:nat, alarm:bool) is
  allocate FBWComp as Prim, FBWComp as Sec, Ail as Ail, Alarmer as Alarmer,
           Conc as Conc, Ctrl[3] as Ctrl, Coord as Coord
  temp p_tok:bool, p_pos:nat, p_lck, p_up, p_dwn:bool, s_tok:bool, s_pos:nat,
       s_lck, s_up, s_dwn:bool, c_pos, pos:nat, lck, up, dwn:bool, safe, ok:bool
  network
    Prim (p_tok;p_ord){p_pos;?p_lck,?p_up,?p_dwn},
    Sec (s_tok;s_ord){s_pos;?s_lck,?s_up,?s_dwn},
    Ail (ok;?c_pos){lck,up,dwn;?pos}, Alarmer(safe;?alarm)
  constrainedby
    Conc (?p_tok | ?s_tok | ?safe), Ctrl (c_pos | ?ok)
  connectedby
    Coord {pos | ?lck, ?up, ?dwn | p_lck, p_up, p_dwn | ?p_pos | s_lck, s_up, s_dwn | ?s_pos}
end system

block FBWComp (in tok:bool; in ord:nat){receive pos:nat; send lck, up, dwn:bool} is
  if tok then
      if ord > pos then lck := false; up := true; dwn := false
      elsif ord < pos then lck := false; up := false; dwn := true
      else lck := true; up := false; dwn := false
      end if
  else lck := false; up := false; dwn := false
  end if
end block

block Ail (in ok:bool; out c_pos:nat){receive lck:bool, up, dwn:bool; send pos:nat} is
  perm pos_buf:nat := 0
  if not(lck) and ok then
      if up then pos_buf := pos_buf + 1
      elsif dwn then pos_buf := pos_buf - 1
      end if
  end if;
  c_pos := pos_buf; pos := pos_buf
end block

block Alarmer (in safe:bool; out alarm:bool) is
  if safe then alarm := false else alarm := true end if
end block

environment Ctrl [const threshold:nat] (in pos:nat | out ok:bool) is
  perm lastPos:nat := 0
  select
     on pos -> lastPos := pos
  [] on ?ok -> if (((lastPos > threshold) and ((lastPos - threshold) < 5))
                 or ((lastPos <= threshold) and ((threshold - lastPos) < 5))) then
                    ok := true
                 else ok := false
               end if
  end select
end environment

environment Conc (out p_tok:bool | out s_tok:bool | out safe:bool) is
  perm p_alive, s_alive:bool := true
  if p_alive then
    select
       on ?p_tok -> p_tok := true -- primary responds
    [] p_alive := false -- primary fails
    end select
  elsif s_alive then
    select
       on ?s_tok -> s_tok := true -- secondary responds
    [] s_alive := false -- secondary fails
    end select
  else on ?safe -> safe := false
  end if
end environment

medium Coord {receive pos:nat | send lck, up, dwn:bool |
                receive p_lck, p_up, p_dwn:bool | send p_pos:nat |
                receive s_lck, s_up, s_dwn:bool | send s_pos:nat} is
```

```
  perm lckBuff:bool := true, upBuff, dwnBuff:bool := false, posBuff:nat := 0
  select
     on p_lck, p_up, p_dwn -> lckBuff := p_lck; upBuff := p_up; dwnBuff := p_dwn
  [] on s_lck, s_up, s_dwn -> lckBuff := s_lck; upBuff := s_up; dwnBuff := s_dwn
  [] on pos -> posBuff := pos
  [] on ?p_pos -> p_pos := posBuff
  [] on ?s_pos -> s_pos := posBuff
  [] on ?lck, ?up, ?dwn -> lck := lckBuff; up := upBuff; dwn := dwnBuff
  end select
end medium
```

LTS generation. The GRL model has been translated into an LNT specification using the GRL2LNT tool, yielding a code that is 2.5 times larger than the input GRL model. Using CADP [11], the LTS of the model has been generated ($2,653$ states, $7,406$ transitions) then reduced modulo branching bisimulation (5 states, $1,287$ transitions), naturals being represented on 8 bits. This apparently small LTS size can be explained by the following facts. Different states represent different values of permanent variables whereas inputs and outputs only appear on transitions (temporary variables are not stored but only used in intermediate computations). Only variables *p_ord*, *s_ord*, and *alarm* are visible on the LTS. Other variables (17 inputs and outputs) are hidden and thus do not occur in transition labels. Environment *Ctrl* constraints the range of possible positions to which *Ail* can move, thus drastically reducing the LTS. Reduction modulo branching bisimulation also helps in keeping the LTS small.

5 Conclusion and Future Work

We gave an overview of GRL, a new language with user-friendly syntax and formal semantics for modeling GALS systems, intended to enhance their design process. GRL combines synchronous features of dataflow languages and asynchronous features of process algebras, and makes possible a versatile, modular description of synchronous subsystems, environment constraints, and asynchronous communications. We designed GRL initially as a pivot language intended to facilitate the connection of industrial environments for designing PLCs (*Programmable Logic Controllers*) to formal verification tools. However, the language appears to be sufficiently expressive and general-purpose to model a wide range of GALS architectures (possibly nondeterministic), implemented on single or distributed platforms, and involving point-to-point or multi-point communications. Moreover, its user-friendly syntax and abstraction level, which is close to the dataflow model used in industry, makes GRL easier to learn and employ than a full-fledged process algebraic language like LNT.

GRL can independently be connected to verification frameworks based on either the synchronous or the asynchronous paradigms. The language is currently equipped with an automated translator to LNT, which makes possible the analysis of GRL descriptions using the rich functionalities of the CADP toolbox (e.g., simulation, verification, performance evaluation), focusing on the asynchronous behaviour of the GALS. GRL and the GRL2LNT translator start

to be used in the Bluesky industrial project[3], which addresses the validation of PLC networks. After a positive feedback received from our industrial partners, we are investigating an automated connection between their PLC design software (based on function block diagrams) and GRL, which would provide a complete analysis chain having CADP as verification back-end. We also develop reusable GRL programs describing basic function blocks and mediums corresponding to communication protocols used in PLC networks.

We plan to continue our work by applying equivalence checking and model checking techniques to industrial GALS systems described in GRL. Hardware/-software co-simulation is also possible using the EXEC/CAESAR framework [14] of CADP, which enables the C code generated from a GRL description to be integrated with a physical platform. We also plan to investigate the connection of GRL to verification frameworks based on the synchronous paradigm to analyse the behaviour of individual blocks corresponding to synchronous subsystems.

References

1. K. Bae, P.C. Ölveczky, and J. Meseguer. Definition, semantics, and analysis of multirate synchronous aadl. In *Proc. of FM*. Springer, 2014.
2. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.
3. G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating reactive processes. In *Proc. of POPL*, pages 85–98. ACM Press, 1993.
4. G. Berry and E. Sentovich. Multiclock Esterel. In *Proc. of CHARME*, volume 2144 of *LNCS*, pages 110–125. Springer, 2001.
5. T. Bultan. Action language: A specification language for model checking reactive systems. In *Proc. of ICSE*. ACM, 2000.
6. J. Carlsson, K. Palmkvist, and L. Wanhammar. Synchronous design flow for Globally Asynchronous Locally Synchronous systems. In *Proc. of ICC*. WSEAS, 2006.
7. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference manual of the LOTOS NT to LOTOS translator (version 5.4). INRIA/VASY, September 2011.
8. N. Coste, H. Hermanns, E. Lantreibecq, and W. Serwe. Towards Performance Prediction of Compositional Models in Industrial GALS Designs. In *Proc. of CAV*, LNCS. Springer, 2009.
9. F. Doucet, M. Menarini, I.H. Krüger, R.K. Gupta, and J.-P. Talpin. A verification approach for GALS integration of synchronous components. *Electr. Notes Theor. Comput. Sci.*, 146(2):105–131, 2006.
10. H. Garavel, F. Lang, and R. Mateescu. Compiler Construction Using LOTOS NT. In *Proc. of CC*, volume 2304 of *LNCS*. Springer, 2002.
11. 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.
12. H. Garavel, G. Salaun, and W. Serwe. On the Semantics of Communicating Hardware Processes and their Translation into LOTOS for the Verification of Asynchronous Circuits with CADP. *Science of Computer Programming*, 2009.

[3] `www.minalogic.com`

13. H. Garavel and D. Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In *Proc. of SPIN*, volume 5578 of *LNCS*, pages 241–260. Springer, 2009.
14. 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.
15. A. Girault and C. Ménier. Automatic Production of Globally Asynchronous Locally Synchronous Systems. In *Proc. of EMSOFT*, volume 2491 of *LNCS*, pages 266–281. Springer, 2002.
16. H. Günther, S. Milius, and O. Möller. On the Formal Verification of Systems of Synchronous Software Components. In *Proc. of SAFECOMP*, volume 7612 of *LNCS*. Springer, 2012.
17. D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.
18. C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. on Software Engineering and Methodology*, 5(3):231–261, 1996.
19. C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.
20. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.
21. F. Jebali, F. Lang, and R. Mateescu. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems. Research Report 8527, Inria, April 2014. Available online at `http://hal.inria.fr/hal-00983711`.
22. E. Lantreibecq and W. Serwe. Model Checking and Co-simulation of a Dynamic Task Dispatcher Circuit using CADP. In *Proc. of FMICS*, 2011.
23. N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Trans. on Software Engineering*, 20(9):684–707, 1994.
24. 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.
25. S. Miller, E. Anderson, L. Wagner, M. Whalen, and M. Heimdahl. Formal verification of flight critical software. In *Proc. of the AIAA Guidance, Navigation and Control Conference and Exhibit*, 2005.
26. M.R. Mousavi, P. Le Guernic, J.-P. Talpin, S.K. Shukla, and T. Basten. Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks. In *Proc. of DATE*. IEEE Computer Society, 2004.
27. F. Moutinho and L. Gomes. State space generation for Petri nets-based GALS systems. In *Proc. of ICIT*, 2012.
28. J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proc. of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2000.
29. S Ramesh. Communicating reactive state machines: Design, model and implementation. In *IFAC Workshop on Distributed Computer Control Systems*, 1998.
30. M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Proc. of DATE*, volume 2. IEEE, 2004.
31. L.H. Yoong, G. Shaw, P.S. Roop, and Z. Salcic. Synthesizing Globally Asynchronous Locally Synchronous Systems With IEC 61499. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 42(6):1465–1477, 2012.