HAL
open science

Formal Modeling and Verification of GALS Systems Using GRL and CADP

Fatma Jebali, Frédéric Lang, Eric Léo, Radu Mateescu

To cite this version:

Fatma Jebali, Frédéric Lang, Eric Léo, Radu Mateescu. Formal Modeling and Verification of GALS Systems Using GRL and CADP. 2014. hal-01082950

HAL Id: hal-01082950

https://inria.hal.science/hal-01082950v1

Preprint submitted on 14 Nov 2014

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.

Formal Modeling and Verification of GALS Systems Using GRL and CADP

Fatma Jebali, Frédéric Lang, Éric Léo, and Radu Mateescu

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

Abstract. The GALS (*Globally Asynchronous, Locally Synchronous*) paradigm is a prevalent approach to design distributed synchronous subsystems that communicate with each other asynchronously. The design of GALS systems is tedious and error-prone due to the complexity of architectures and high synchronous and asynchronous concurrency involved. This paper proposes a model-based approach to formally verify such systems. Specifications are written in GRL (*GALS Representation Language*), dedicated to model GALS systems with homogeneous syntax and formal semantics. We present a translation from GRL to LNT, a value-passing process algebra with imperative flavour. The translation is automated by means of the GRL2LNT tool, making possible the analysis of GRL specifications using the CADP toolbox. We illustrate our approach with an access management system for smart parking based on distributed software systems embedded in programmable logic controllers.

1 Introduction

Synchronous specification languages have for long been a valuable approach in the design of large scale hardware and software systems. Their easy-to-learn syntax, their sound semantics suitable for efficient verification, and their underlying assumptions such as determinism and atomicity, are among the key features making synchronous languages and frameworks well-supported by industrial design flows. However, an increasing number of applications such as sensor networks and multiprocessor architectures are distributed by nature. It follows that the ideal assumptions of zero-time computations and instantaneous broadcasting communications may be unrealistic [1, 3], since distribution introduces asynchrony and nondeterminism in the global behaviour of these systems.

This observation gave birth to the GALS (*Globally Asynchronous, Locally Synchronous*) paradigm [5], which makes a good compromise between synchronous systems having each its own clock structure and asynchronous communication schemes. Interestingly enough, the GALS concept has been implemented in a wide spectrum of applications: System-on-Chip [21] and Network-on-Chip [21] are instances of hardware-based systems distributed on one single architecture; flight control systems [22] and automotive central lock systems [6] are

instances of hardware-based systems distributed on several architectures; distributed coordination of web services [19] and security surveillance systems [15] are instances of distributed software systems.

The design and verification of such systems can be extremely cumbersome due to the heterogeneous nature of computations and the complexity of architectures, in particular if used in safety-critical applications. Model-based design flows and formal analysis are then required to help designers master that complexity and build confidence on the correctness of systems. The traditional approaches consist either in using synchronous languages and frameworks as they are (e.g., [18]) or in extending them to encompass some asynchronous features and nondeterministic behaviour (e.g., [12] and [20]). Those approaches are well suited to specify latency-insensitive designs when synchronous components are distributed in hardware architectures and run correctly in the presence of interconnection delays (messages are delivered in order and cannot be lost). However, asynchrony and nondeterminism still are not encompassed as main concepts and powerful verification techniques dealing with asynchronous concurrency such as compositional reduction are missing [10].

This makes the aforementioned approaches insufficient when addressing completely desynchronized software embedded in several distributed sites and/or architectures in which there is no restriction on communication delays and messages can be lost or delivered in an arbitrary order. In addition, most machines and embedded devices are interconnected due to the massive use of connectivity technologies such as Internet and ubiquitous computing. This enforces the degree of asynchrony and nondeterminism, thus making network topologies and communication mechanisms more and more complex.

We believe that new modeling and verification approaches that inherently combine synchronous and asynchronous features are likely to become central in the integration of formal analysis in the design process of GALS systems. Towards this objective, GRL has been proposed [13] as an attempt to combine the main features of synchronous programming and process algebra in one unified language, while keeping the syntax homogeneous for better acceptance by industrial users. GRL allows synchronous systems (blocks), environmental constraints (environments), and asynchronous communication mechanisms (mediums) to be described in a modular way and at the appropriate level of abstraction. To analyse GRL specifications, our aim is to benefit of the widely-used CADP toolbox [9] offering a wide range of the state-of-the-art verification approaches.

In this paper, we propose a verification approach to bridge the gap between design frameworks of GALS systems and formal verification tools using GRL and CADP. Our approach can be summarized as follows. We first propose a translation of GRL descriptions to LNT [4], one of the input languages of CADP. LNT is a specification language for asynchronous concurrent processes, which aims at combining the best of process algebraic languages and imperative functional programming languages. GRL blocks are represented by LNT functions encapsulated into wrapper processes as proposed in [10]. GRL environments and mediums are represented by LNT processes. The asynchronous execution of

blocks, mediums, and environments is modeled using the LNT parallel composition operator. This translation is fully automated by the GRL2LNT tool we have developed. We illustrate our approach with an example we designed with Crouzet Automatismes (a branch of Schneider Electric group) in the context of Bluesky[1]. The application is an AMS (*Access Management System*) to be used in a smart parking, which consists of a network of distributed software systems embedded in PLCs (*Programmable Logic Controllers*).

The paper is organized as follows. Section 2 describes the AMS. Section 3 presents GRL. Section 4 presents LNT and CADP. Section 5 outlines the translation from GRL to LNT and the corresponding tools. Section 6 shows the model checking of some properties of the AMS. Section 7 concludes the paper.

2 AMS (Access Management System) for Smart Parking

Throughout this article, we consider an example based on distributed software embedded in em4 controllers provided by Crouzet. These are a new generation of PLCs embedding synchronous software supported by *em4soft* tools, which are based on graphical block diagrams and enable to create and implement smart automation applications. The system we consider is part of a smart parking platform to make parking easier and more efficient. It consists of a network of communicating em4 applications that control and manage the vehicle access through automatic vehicle detection. A user can interact with the system either in a local mode using physical sensors or in a remote mode using mobile applications. Mobile applications enable also to pay and get the exit clearance.

For the sake of simplicity, we consider two em4 controllers. The first one manages an entry gate giving the access to the parking over the reception of an order. The controller checks cyclically whether there still are unoccupied parking spots, in which case a green light is maintained on, otherwise a red light is turned on. Once an order is detected, depending on whether the red or the green light is on, a request to open the gate may be denied or granted, respectively. In the latter case, the entry gate remains open for a fixed amount of time and a yellow light is maintained on until the closure of the gate. One light at the same time should be on. The second controller manages an exit gate allowing to leave the parking over the reception of an order. Each time an order is detected, the exit gate remains open for a fixed amount of time. The two controllers communicate through a network to keep informed about the entrance and exit flow.

3 GRL (GALS Representation Language)

An excerpt of the syntax of GRL is given in Figure 1. Symbols K, T, X, E, S, B, M denote respectively literal constants, type identifiers, variables, expressions (built upon constants, variables, and function applications), system identifiers, block identifiers, and medium identifiers. GRL consists of a synchronous part made of

[1] project of the Minalogic "pole de compétitivité" `http://www.minalogic.com`

blocks, connected asynchronously through *mediums*. Environmental constraints on blocks are described in *environments*. For the sake of conciseness and for lack of space, we do not present environments, which have a lot in common with mediums. Blocks, mediums, and environments are assembled together in *systems*. In the sequel, blocks, mediums, and environments are called *actors*.

```
system ::= system S
               [(X_0 : T_0 , ... , X_m : T_m)] is
           allocate actor_0 , ... , actor_n
           [temp X'_0 : T'_0 , ... , X'_l : T'_l]
           network
           block_call_0 , ... , block_call_p
           [connectedby
           med_call_0 , ... , med_call_r]
        end system

block ::= block B
              [[const_param]]
              [(inout_param_0 ; ... ; inout_param_m)]
              [{com_param_0 ; ... ; com_param_n}] is
          [allocate sub_block_0 , ... , sub_block_p]
          [local_var_0 , ... , local_var_l]
            I
        end block

med ::= medium M [[const_param]][{com_param_0 | ... | com_param_m}] is
          [allocate sub_block_0 , ... , sub_block_n]
          [local_var_0 , ... , local_var_l]
            I
        end medium
const_param ::= const X_0 : T_0 [:= E_0] , ... , X_n : T_n [:= E_n]
inout_param ::= (in | out) X_0 : T_0 [:= E_0] , ... , X_n : T_n [:= E_n]
  com_param ::= (send | receive) X_0 : T_0 , ... , X_n : T_n
    local_var ::= (perm | temp) X_0 : T_0 [:= E_0] , ... , X_n : T_n [:= E_n]
    sub_block ::= B [[arg_0 , ... , arg_n]] as Bi
        actor ::= B[[arg_0 , ... , arg_n]] as Bi | M[[arg_0 , ... , arg_n]] as Mi
   block_call ::= Bi [(ch_0 ; ... ; ch_m)][{ch'_0 ; ... ; ch'_n}]
     med_call ::= Mi {ch_0 | ... | ch_m}
       signal ::= on [?]X_0 , ... , [?]X_n -> I
I ::= null | X:=E | X[E_0]:=E_1 | X.f:=E | I_0 ; I_1 | Bi(arg_0 , ... , arg_n) | X := any T [where E]
    | case E is K_0 -> I_0 | ... | K_n -> I_n | [any -> I_{n+1}] end case
    | while E loop I_0 end loop | for I_0 while E by I_1 loop I_2 end loop
    | if E_0 then I_0 elsif E_1 then I_1 ... elsif E_n then I_n else I_{n+1} end if
    | select I_0 [] ... [] I_n end select | signal
```

Fig. 1: Excerpts of the GRL syntax

Blocks. A block is defined as the synchronous composition of one or several other blocks (called *subblocks*), together with the deterministic statements available in standard programming languages (assignments, if-then-else, while and for loops, etc.)[2]. The current state of a block is defined by the values of permanent variables declared using the keyword **perm**. During each synchronous cycle, a block operates as follows: (1) it consumes its inputs; (2) it evaluates its function, which depends on both the block inputs and the current state; (3) it produces its outputs and computes the state of the system in the next cycle. These computations are assumed to be done in zero-delay, following the standard abstraction of synchronous programming. In addition to input and output parameters, a block can be parameterized with constant parameters.

Instances can be created from block definitions using the keyword **allocate**. Each block instance has its own memory consisting of a copy of its permanent

[2] The nondeterministic statements "X := **any** T [**where** E]" and "**select** I_0 [] ... [] I_n **end select**" are not allowed in block definitions, but only in mediums and environments.

variables and those of its subblocks transitively. Block instances can be invoked and composed to interact with each other by instantaneous broadcasting, i.e., outputs produced by a block in a cycle can be consumed by other blocks in the same cycle. In this way, data is processed along causal dependencies between subblocks, making the behaviour of blocks deterministic.

Excerpts of the GRL code of the AMS introduced in Section 2 are given in Figure 2. Block *Block_Timer_BW* (lines 1-7, Fig. 2) ensures that the output is set to true if the input is true during several consecutive cycles of the block. Permanent variable *Pre_Input* stores the value of *Input* from one execution to the next (lines 3 and 6, Fig. 2). A parameterized instance *B02* of block *Block_Timer_BW* is allocated (line 11, Fig. 2) then invoked (line 15, Fig. 2) inside block *Out_controller*. Block *Out_controller* is defined by a synchronous composition of subblock instances *B01*, *B02*, and *B05* (lines 14-16, Fig. 2). Data flow from *B01* to *B02* is ensured through variable *c1* whose value is produced by *B01* (actual parameter ?*c1*) and consumed by *B02* (actual parameter *c1*). In *B05* invocation (line 16, Fig. 2), symbols "_" and "?_" denote unconnected inputs and outputs respectively. For unconnected inputs, the corresponding formal parameters must be assigned default values in the definition of block *Block_Timer_AC*, similarly to parameter *Open_Cmd* in block *Out_controller* (line 9, Fig. 2).

Asynchronous composition of blocks. A block can communicate asynchronously with other blocks in a network through communication mediums. Each communicating block has send and receive parameters (line 10, Fig. 2), which are connected to receive and send parameters of mediums, respectively. Unlike blocks, mediums support nondeterminism, which makes the description of communication protocols at a high level of abstraction possible. Instances of mediums can be created and invoked in systems, similarly to blocks.

Interactions between a block and several mediums are carried out as follows. When it starts its execution cycle, the block triggers the execution of all mediums connected to its receive parameters, so that mediums produce the data required by the block. When it finishes its execution cycle, the block triggers the execution of all mediums connected to its send parameters, so that mediums can consume data produced by the block. Following this execution model, at most two interactions are possible between a given pair of block and medium. Exchanged data is therefore grouped in *channels* (tuples of data) so that to each block-medium interaction is associated one single channel. Such a channel is called *activated* during the interaction. Depending on the activated channel, different fragments of a medium code have to be executed, possibly separated using a nondeterministic "**select**" statement. To each channel is thus associated a *signal* statement, which guards the fragment of the medium code that should be executed over the channel activation.

For example, consider medium *Medium_Buffer_Bit* (lines 19-29, Fig. 2). It represents a non-blocking single-place buffer that allows communication between a sender block (through channel *Input*) and a receiver block (through channel *Output*). The code executed upon activation of the channels *Input* and *Output* is on lines 22-23 and lines 25-27 of Figure 2, respectively.

```
1: block Block_Timer_BW [const Rising_Edge : bool := true, Falling_Edge : bool := false]
2:                          (in Input : bool; out Output : bool) is
3:   perm Pre_Input : bool := false
4:   Output := (((Input and not (Pre_Input)) and Rising_Edge)
5:              or ((not (Input) and Pre_Input) and Falling_Edge));
6:   Pre_Input := Input
7: end block
8:
9: block Out_Controller (in Open_Cmd : bool := false; out Door_Open : bool)
10:                        {receive Open_Distant_Cmd : bool; send Decrease_Counter : bool} is
11:   allocate Block_Or as B01, Block_Timer_BW [true, false] as B02,
12:            Block_Timer_AC [0, 5, Cycle] as B05
13:   temp c1 : bool
14:   B01 (Open_Cmd, Open_Distant_Cmd, ?c1);
15:   B02 (c1, ?Decrease_Counter);
16:   B05 (Decrease_Counter, _, ?Door_Open, ?_ ,?_, ?_, ?_)
17: end block
18:
19: medium Medium_Buffer_Bit {receive Input : bool | send Output : bool} is
20:   perm Buffer : bool := Cst_Bool_Empty_Buffer
21:   select
22:     on Input → if (Buffer == Cst_Bool_Empty_Buffer) then Buffer := Input
23:                else null end if
24:   []
25:     on ?Output → if (Buffer == Cst_Bool_Empty_Buffer) then
26:                    Output := Cst_Bool_Default_Value
27:                  else Output := Buffer; Buffer := Cst_Bool_Empty_Buffer end if
28:   end select
29: end medium
30:
31: system Main (Out_Open_Cmd : bool, Out_Door_Open : bool, ...) is
32:   allocate Out_Controller as Out_Controller, In_Controller as In_Controller,
33:            Medium_Buffer_Bit as Med, ...
34:   temp Decrease_Counter_1 : bool, Decrease_Counter_2 : bool
35:   network
36:     Out_Controller (Out_Open_Cmd; ...){...; ?Decrease_Counter_1},
37:     In_Controller (In_Open_Cmd; ...){...; Decrease_Counter_2} ...
38:   connectedby
39:     Med {Decrease_Counter_1 | ?Decrease_Counter_2}, ...
40: end system
```

Fig. 2: GRL code of the AMS

Actors are composed inside systems to build networks of blocks that execute independently and communicate asynchronously with each other across mediums. Connections between blocks and mediums are carried out through parameters that are either observable or not from the outside world. In our example, the network consists of two blocks *In_Controller* and *Out_Controller* communicating across medium *Med* to keep informed of the actual number of available spots in the parking (lines 35-39, Fig. 2). Using channel *Decrease_Counter_1*, *Out_Controller* informs through medium *Med* whether a vehicle has left or not the parking in its current execution cycle. In turn, *In_Controller* receives the information using channel *Decrease_Counter_2*. Parameters *Out_Open_Cmd* and *Out_Door_Open* (line 31, Fig. 2) are observable from the outside world, whereas variables *Decrease_Counter_1* and *Decrease_Counter_2* (line 34, Fig. 2) are not.

Semantics. The formal dynamic semantics of GRL systems are defined using structural operational semantic rules in terms of LTS (*Labelled Transition System*), the complete definition being available in [14]. States consist of the union

of actor memories, the initial state being the initial memory. Labels represent the block invocations, each label corresponding to a block instance identifier B and its corresponding channels $\overline{ch}$ (input/output channels) and $\overline{ch'}$ (send/receive channels). A transition $\mu \xrightarrow{B(\overline{ch})\{\overline{ch'}\}} \mu'$ means that the combined execution of block instance B together with its connected mediums (and environments) instances in the memory μ produces the new memory μ'. The resulting LTS represents the interleaving of all block instance executions.

4 The LNT language and the CADP toolbox

An excerpt of the syntax of LNT is given in Figure 3. Symbols T, X, E, P, B, π, F, G, and Γ denote respectively type identifiers, variables, expressions, patterns, behaviours, process identifiers, function identifiers, gate identifiers, and channel identifiers. LNT consists of two parts. The data part is defined by means of rich data type structures, statements built upon standard algorithmic control structures, and functions. The control part is defined by means of behaviours and processes. Behaviours are extensions of statements with process instantiation, parallel composition, gate communication, and nondeterministic statements. Functions are parameterized with data variables and processes can be parameterized with both data variables and gates. Formal parameters can be of type in (call by value), out (call by reference, the function being in charge of producing a value for the parameter), or inout (call by reference, the function being allowed to read and update the parameter value) and their respective actual parameters are preceded by symbols "`!`", "`?`", and "`!?`", respectively. Communication takes place by rendezvous on gates, with bidirectional transmission of multiple values. Gates are typed with channels (not to be confused with the GRL notion of channels).

```
 process ::= process Π [G0:Γ0, ..., Gm:Γm] is B end process
function ::= function F (param0, ..., paramm):T is B end function
   param ::= (in | out | inout) X0:T0, ..., Xn:Tn
 channel ::= channel Γ is gate0, ..., gatem end channel
    gate ::= (T0, ..., Tm)
       B ::= null | X:=E | X[E0]:=E1 | X.f:=E | B0;B1 | eval F(arg0, ..., argn)
           | if E0 then B0 [[elsif E1 then B1 ... elsif En then Bn] else Bn+1 ]end if
           | while E loop B0 end loop | for B0 while E by B1 loop B2 end loop
           | case E is P0 -> B0 | ... | Pn -> Bn | [any -> Bn+1] end case
           | var X0:T0, ..., Xn:Tn in B end var
             /* the following constructs are reserved for the control part */
           | hide G0:Γ0, ..., Gn:Γn in B end hide | X := any T [where E]
           | par G0, ..., Gn in B0 || ... || Bm end par | select B0 [] ... [] Bn end select
           | Π[G0, ..., Gn](arg0, ..., argn) | G(O0, ..., On)
     arg ::= !E | ?X | !?X          O ::= [!]E | ?X          P ::= X | C(P1, ..., Pn)
```

Fig. 3: Excerpts of the LNT syntax

LNT programs can be verified using the toolbox CADP (*Construction and Analysis of Concurrent Processes*[3]) [9]. The LNT.OPEN tool translates LNT specifications into LTS 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 [16] allows one to verify temporal properties written in MCL (*Model Checking Language*), which extends the alternation-free μ-calculus with generalized regular expressions, data-based constructs, and fairness operators.

5 Model Transformation from GRL to LNT

This section is devoted to the translation from GRL to LNT. Due to space limitations, we do not give the complete formal translation [4], but we outline its main principles. The correspondence between the main GRL constructs and how they are translated in LNT is summarized in Table 1. Data types, expressions, and algorithmic control structures (including nondeterministic choice and assignment) of GRL have a direct, one-to-one correspondence with their LNT counterparts.

GRL constructs	Translation into LNT
statements	
block invocation	function invocation
signal	statement guarded by a gate (sequential composition)
parameters and variables	
input parameter	in parameter
output parameter	out parameter
constant parameter	in parameter
permanent variable	inout parameters and local variables
temporary variable	local variable
channel	local variables and gate
actors and systems	
block definition	function definition
block instance	function call (if inside an actor) process definition and instantiation (if inside a system)
medium definition	process definition
medium instance	process instantiation
system	process

Table 1: Correspondence between GRL and LNT

[3] `http://cadp.inria.fr`

[4] the formal description of the translation from GRL to LNT is available in `http://convecs.inria.fr/doc/grl2lnt.pdf`

Translation of Blocks. Each block in GRL has one definition and several instances, each instance having its own memory. Each block definition is systematically translated into an LNT function implementing one execution cycle of the block, i.e., computes outputs from inputs. For instance, LNT functions defined on lines 1-5 and lines 7-14 of Figure 4 correspond to blocks *Block_Timer_BW* and *Out_Controller* of Figure 2, respectively. A block instance can be allocated inside another block or inside a system. In the first case, the block instance is governed by the pace of the enclosing block and interacts synchronously with other subblocks via input output data flows. In the second case, the block instance is a stand-alone system governed by its own pace and communicating asynchronously with other blocks through mediums. The block instance is translated into a process called *wrapper process*, described hereafter, which invokes the function corresponding to the block definition inside an infinite loop. For instance, the LNT process defined on lines 16-30 of Figure 4 corresponds to block instance *Out_Controller* inside system *Main* (lines 32 and 36) in Figure 2.

The imperative style of LNT makes the translation from GRL blocks to LNT functions quite straightforward for most GRL constructs. The zero-delay assumption is granted for free, since LNT functions execute atomically. However, a main difficulty concerns the memory representation, since LNT has only local variables which lose their values between subsequent executions. To circumvent this limitation, we implement permanent variables using local variables declared and initialized in the wrapper process (parameter *Out_Controller_B02_Pre_Input* lines 18 and 20, Fig. 4). Those variables are propagated through inout parameters to functions corresponding to subblocks, transitively. Therefore, for each function corresponding to a block we have to synthesize the set of variables implementing the memory of the block in a bottom-up manner. For instance, parameters *Pre_Input* and *B02_Pre_Input* (lines 1 and 7, Fig. 4) correspond to the permanent variable *Pre_Input* (line 3, Fig. 2).

Synchronous composition of block instances is translated by sequential composition of calls to the corresponding LNT functions. A difficulty is the translation of unconnected parameters of the form "_" and "?_". For each input actual parameter "_" in the block invocation, the default value of the corresponding formal parameter is fetched and passed to the LNT function call. For each output actual parameter "?_" in the block invocation, an unused variable with the same type as the corresponding formal parameter is declared and passed to the LNT function call.

The translation of asynchronous composition is more subtle. Unlike GRL, which supports communication through variables, LNT processes are parameterized with typed gates through which data can be exchanged. To each GRL channel (set of parameters) of the block invocation is associated a set of local variables and a gate in the LNT definition process. Those variables are passed to the LNT function encapsulated inside the process. GRL channels of the form "_, . . . ,_" and "?_, . . . ,?_" are ignored, since they do not require gate communication. For instance, to the GRL channel *Out_Open_Cmd* (lines 31 and 36, Fig. 2) is associated the LNT variable *Open_Cmd* (line 19, Fig. 4), which is received

```
1: function Block_Timer_BW (in Falling_Edge : Bool, in Rising_Edge : Bool, in Input : Bool, out
   Output : Bool, inout Pre_Input : Bool) is
2:    Output := (((Input and not (Pre_Input)) and Rising_Edge)
3:                 or ((not (Input) and Pre_Input) and Falling_Edge));
4:    Pre_Input := Input
5: end function
6:
7: function Out_Controller (in B02_Falling_Edge : Bool, in B02_Rising_Edge : Bool, in Open_Cmd
   : Bool, ..., inout B02_Pre_Input : Bool) is
8:    var c1 : Bool in
9:       ...
10:      eval Block_Or (Open_Cmd, Open_Distant_Cmd, ?c1);
11:      eval Block_Timer_BW (B02_Falling_Edge,B02_Rising_Edge,c1,?Decrease_Counter,
   !?B02_Pre_Input);
12:      Door_Open := Decrease_Counter
13:   end var
14: end function
15:
16: process Main_Out_Controller [Lock, Release : None, GATE_Out_Open_Cmd : CHANNEL_bool,
   GATE_Decrease_Counter_1 : CHANNEL_bool, ...] is
17:      ...
18:      var Out_Controller_B02_Pre_Input : Bool in
19:         var Open_Cmd : Bool, Decrease_Counter : Bool, ... in ...
20:         Out_Controller_B02_Pre_Input := false;
21:         loop
22:            Lock;
23:            GATE_Out_Open_Cmd (?Open_Cmd);
24:            GATE_Out_Open_Distant_Cmd (?Open_Distant_Cmd);
25:            eval Out_Controller (Out_Controller_B02_Falling_Edge,Out_Controller_B02_Rising_Edge,
   Open_Cmd,?Door_Open,Open_Distant_Cmd,?Decrease_Counter,!?Out_Controller_B02_Pre_Input);
26:            GATE_Out_Door_Open (Door_Open);
27:            GATE_Decrease_Counter_1 (Decrease_Counter);
28:            Release
29:         end loop
30: end var end var end var end process
31:
32: process Medium_Buffer_Bit [GATE_Input : CHANNEL_bool, GATE_Output : CHANNEL_bool]
   (inout Buffer : Bool) is
33:    var Input : Bool, Output : Bool in select
34:         GATE_Input (?Input);
35:         if (Buffer == Cst_Bool_Empty_Buffer) then Buffer := Input else null end if
36:         []
37:         if (Buffer == Cst_Bool_Empty_Buffer) then Output := Cst_Bool_Default_Value
38:         else Output := Buffer; Buffer := Cst_Bool_Empty_Buffer end if;
39:         GATE_Output (!Output)
40:      end select end var
41: end process
42:
43: process Main_Med [GATE_Decrease_Counter_1 : CHANNEL_bool, GATE_Decrease_Counter_2
   : CHANNEL_bool] is
44:    var Med_Buffer : Bool in
45:      Med_Buffer := Cst_Bool_Empty_Buffer;
46:      loop
47:         Medium_Buffer_Bit[GATE_Decrease_Counter_1,GATE_Decrease_Counter_2](!?Med_Buffer)
48:      end loop
49:   end var
50: end process
51: ...
52: channel CHANNEL_bool is (Bool) end channel
53: process Mutex [Lock, Release : None] is  loop Lock; Release end loop  end process
```

Fig. 4: LNT code of the AMS (part 1)

through gate *GATE_Out_Open_Cmd* (line 23, Fig. 4), and passed as input parameter to the encapsulated function call (line 25, Fig. 4). Since LNT gates are

typed by LNT channels, to each LNT gate corresponds an LNT channel defining its communication profile (i.e., number and types of the exchanged values). For instance, the type of the LNT gate *GATE_Out_Open_Cmd* is *CHANNEL_bool* (line 52, Fig. 4). Our translation ensures that the generated LNT channels are pairwise distinct.

The execution of the wrapper process can be summarized as follows. First, it initializes once and for all local variables representing the memory of the block under translation (lines 18 and 20, Fig. 4). Then, it performs cyclically (inside the infinite loop) the following steps (lines 21-29, Fig. 4): (1) data reception on gates corresponding to receive and input channels of the block (lines 23-24, Fig. 4), (2) call to the function corresponding to the block invocation (line 25, Fig. 4), (3) data emission on gates corresponding to output and send channels of the block (lines 26-27, Fig. 4).

Each execution of the synchronous cycle of a block produces a sequence of synchronizations between the process corresponding to the block and those corresponding to its connected mediums and environments (if any). According to the synchronous paradigm, this sequence is atomic. To prevent sequences of synchronizations corresponding to different blocks to interleave, we ensure mutual exclusion of the synchronization sequences using additional gates *Lock* (line 22, Fig. 4) and *Release* (line 28, Fig. 4) that start and finish each sequence, respectively. The process of the block is then synchronized on those gates with the process *Mutex* defined on line 53 of Figure 4.

Note that our translation does not preserve exactly the GRL semantics, since to one transition in the GRL semantics corresponds a sequence of transitions in the LNT semantics. However, there is a clear correspondence between the LTS obtained from the formal semantics of GRL and the one obtained by applying the translation into LNT described above. The label of the GRL transition can be reconstructed from the labels of the atomic sequence of LNT transitions. As an illustration, consider an invocation of the block *Out_Controller* with parameters *Out_Open_Cmd* and *Out_Open_Distant_Cmd* both set to false. The LTS corresponding to the formal semantics of GRL associates the following transition to one execution of the block:

$$0 \xrightarrow{Out_Controller(false;?false)\{false;?false\}} 1$$

The LTS corresponding to the formal semantics of LNT, obtained after translation, associates the following sequence of transitions to one execution of the block:

$$0 \xrightarrow{Lock} 1 \xrightarrow{GATE_Out_Open_Cmd\ !\ false} 2 \xrightarrow{GATE_Out_Open_Distant_Cmd\ !\ false} 3$$

$$\xrightarrow{GATE_Out_Door_Open\ !\ false} 4 \xrightarrow{GATE_Decrease_Counter_1\ !\ false} 5 \xrightarrow{Release} 6$$

Translation of Mediums. Unlike a block definition, a medium definition cannot be translated into an LNT function due to the occurrence of nondeterministic statements and signal statements enabling asynchronous communication. Thus, it is directly translated into an LNT process. For instance, the LNT process

```
54: process Main [GATE_Decrease_Counter_1, GATE_Decrease_Counter_2 : CHANNEL_bool,...] is
55:   hide Lock, Release : None, ... in
56:     par GATE_Decrease_Counter_2, GATE_Decrease_Counter_1, ... in
57:       par Lock, Release in
58:         Mutex [Lock, Release]
59:         ||
60:         par
61:           Main_In_Controller [..., GATE_Decrease_Counter_2, Lock, Release]
62:           || Main_Out_Controller [..., GATE_Decrease_Counter_1, Lock, Release]
63:         end par
64:       end par
65:       ||
66:       par
67:           Main_Med [GATE_Decrease_Counter_1, GATE_Decrease_Counter_2]
68:           || ...
69:       end par
70:     end par
71:   end hide
72: end process
```

Fig. 5: LNT code of the AMS (part 2)

defined on lines 32-41 of Figure 4 corresponds to medium *Medium_Buffer_Bit* of Figure 2. GRL channels are translated similarly to those in block instances, except that even unconnected channels are represented. Permanent variables are translated similarly to those in block definitions. A signal statement of the form "**on** X_0 **,** ... **,** X_n **->** I" is translated into a sequential composition of: (1) reception of messages on the gate corresponding to the channel "X_0 **,** ... **,** X_n" and (2) a translation of the statement "I" (lines 34-35, Fig. 4). A signal statement of the form "**on** ?X_0 **,** ... **,** ?X_n **->** I" is translated into a sequential composition of: (1) a translation of the statement "I" and (2) emission of messages on the gate corresponding to the channel "X_0 **,** ... **,** X_n" (lines 37-39, Fig. 4). Similarly to block instances at system level, to each medium instance (necessarily occurring inside a system) is associated an LNT process. For instance, the LNT process defined on lines 43-50 of Figure 4 corresponds to medium instance *Med* inside system *Main* (lines 33 and 39, Fig. 2). This process encapsulates the LNT process corresponding to the medium definition inside an infinite loop.

Translation of Systems. A system is translated into a process inside which are composed in parallel the processes corresponding to the various block and medium instances of the system. For instance, the LNT process defined on lines 54-72 Figure 5 corresponds to the system *Main* (lines 31 and 40, Fig. 2). The process is parameterized with gates declared in two ways. Gates declared in the process profile (line 54, Fig. 5) are visible in the corresponding LTS and represent parameters that are declared in the GRL system profile. Gates declared using the **hide** construct (line 55, Fig. 5) are not be visible in the corresponding LTS and represent the GRL parameters declared as temporary variables in the system.

Instance processes corresponding to actors are composed in parallel and encapsulated inside the process corresponding to the system (lines 56-70, Fig. 5). Blocks are composed in parallel in pure interleaving (lines 60-63, Fig. 5). Each instance process of a block synchronizes with process *Mutex* on gates *Lock* and

Release (lines 57-64, Fig. 5). Similarly to instance processes of blocks, instance processes of mediums are composed in pure interleaving (lines 66-69, Fig. 5).

5.1 Tool support

We developed a tool named GRL2LNT using the Syntax/Traian Lotos NT compiler construction technology [8]. It consists of about 30,000 lines of code and translates GRL specifications into LNT specifications. Additionally, we developed a tool named GRL.OPEN 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 (including examples of controller applications provided by Crouzet) totalizing about 7,000 lines of code. Some specifications also include external C and LNT code, as supported by GRL. The generated files consist of about 18,000 lines of LNT code and each LNT file is on average 2.5 times larger (in lines of code) than the GRL file. This linear expansion is due mainly to channel declarations and to the fact that each GRL actor is translated into more than one LNT process or function.

6 Functional Verification

We used GRL.OPEN together with the GENERATOR tool of CADP to build the LTS corresponding to each controller and to the whole AMS. The translation from GRL to LNT is instantaneous. The LTS of the entrance controller, the exit controller, and the global system have respectively 4,260,680 states and 6,554,895 transitions, 129 states and 183 transitions, and 13,797,737 states and 17,238,978 transitions and are generated in 30 seconds, 3 seconds, and 190 seconds on a Intel Xeon W3550 (3.07GHz, 8GB RAM) running Linux.

Some behavioural abstractions on the global model were necessary to make it suitable for model checking without loss of accuracy: the body of some blocks has been simplified, the yellow light is not kept on for several cycles, and some timers have been removed.

We have written a set of safety and liveness properties which we have extracted from the example description in natural language. In a first step, we specified a collection of properties that each controller should satisfy (13 safety and liveness properties). As an illustration, the following MCL safety property ensures that the gate controlled by the exit controller should not open less than five cycles. It has the form "*[R] false*" where R is a regular expression, and it evaluates to true if no execution sequence matches R. Here, it states that between two states where the gate is not open, there cannot be one to four cycles where the gate is open. The regular expression inside the brackets describes the entire execution cycle of the controller.

```
[true*. {GATE_OUT_DOOR_OPEN!false}.{GATE_DECREASE_COUNTER?any}.
        (tau*.{GATE_OUT_OPEN_CMD?any}.{GATE_OUT_OPEN_DISTANT_CMD?any}.
        {GATE_OUT_DOOR_OPEN!true}.{GATE_DECREASE_COUNTER?any}){1...4}.
        tau*.{GATE_OUT_OPEN_CMD?any}.{GATE_OUT_OPEN_DISTANT_CMD?any}.
        {GATE_OUT_DOOR_OPEN!false}
]false
```

This property evaluates to true in 4 seconds, using 4MB of memory.

In a second step, we specified a collection of properties that the network of controllers should satisfy (8 safety and liveness properties). As an illustration, the following MCL safety property ensures that the switch from red light to green light should not be possible if no open order on the exit gate has been received. It states that there is no execution path from red light to green light without an opening of the exit gate.

```
[true*.
{GATE_RED_LIGHT!true}.(not({GATE_OUT_DOOR_OPEN!true}))*.{GATE_GREEN_LIGHT!true}
]false
```

This property evaluates to true in 520 seconds using 4KB of memory.

Note that the medium implemented in this example is very simple and does not ensure that a message will eventually reach its destination. For example, we could have expected the following to be correct: when a car leaves the parking, the green light eventually turns on. This can be checked with the following MCL liveness property, which has the form "*[R] inev(A)*" where R is a regular expression and A is an action, and evaluates to true if every execution sequence matching R ends in a state from which A is inevitably reachable.

```
[true*.{GATE_OUT_DOOR_OPEN!true}] inev({GATE_GREEN_LIGHT!true})
```

This property evaluates to false in 2 seconds using 4MB of memory. It means that a requirement is missing in the initial specification.

7 Conclusion

We have presented an approach for the formal modelling and verification of GALS systems. These systems are complex and have to be analysed in powerful verification frameworks specifically designed to model asynchrony and nondeterminism. Contrary to other approaches that combine two different languages for the synchronous part and the asynchronous part [6, 10], we use a unified language named GRL, specifically designed to model the heterogeneous behaviour of GALS systems.

We proposed a translation of GRL specifications into LNT, one of the input languages of the CADP toolbox. This allows the designers of GALS systems to apply all the state-of-the-art verification tools available in CADP to analyse GRL specifications. We automated the translation in the tool GRL2LNT, which has been tested on a large number of examples.

GRL and GRL2LNT represent a step forward to enhance industrial design flows with asynchronous verification tools. We try to address a lack of approaches dealing with asynchronous concurrency in the industry of GALS systems. This lack is at least twofold: (1) asynchronous concurrency is intrinsically more complex than synchronous concurrency and (2) asynchronous verification tools are expensive to integrate in industrial frameworks [7]. Our approach aims at keeping industrial frameworks as they are and enhance them with both automatic connections to asynchronous frameworks and user-friendly analysis interfaces. We believe that it is cost effective compared to approaches based on complex refinement algorithms [17] or automatic generation of distributed implementations [11], since our approach does not require drastic shift in the design flow.

We are currently enriching the GRL framework with reusable libraries modelling basic function blocks as well as communication protocols used in the GALS community such as LTTA [2] and Modbus. Also, in collaboration with Crouzet engineers, we seek to automate the generation of GRL specifications from the *em4soft* design software. We also study the design of a user-friendly property language to make model checking easily accessible by industrial designers.

References

1. A. Benveniste, B. Caillaud, and P. Le Guernic. *From synchrony to asynchrony.* Springer, 1999.
2. A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Proc. of Embedded Software.* Springer, 2002.
3. L.P. Carloni and A.L. Sangiovanni-Vincentelli. A framework for modeling the distributed deployment of synchronous designs. *FMSD*, 28(2):93–110, 2006.
4. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference manual of the LOTOS NT to LOTOS translator, 2011.
5. D.M. Chapiro. Globally-asynchronous locally-synchronous systems, 1984.
6. 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.
7. H. Garavel. Reflections on the future of concurrency theory in general and process calculi in particular. *ENTCS*, 209, 2008.
8. H. Garavel, F. Lang, and R. Mateescu. Compiler construction using LOTOS NT. In *Proc. of CC.* Springer, 2002.
9. 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.
10. H. Garavel and D. Thivolle. Verification of GALS systems by combining synchronous languages and process calculi. In *Proc. of SPIN.* Springer, 2009.
11. A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In *Embedded Software.* Springer, 2002.
12. N. Halbwachs and S. Baghdadi. Synchronous modelling of asynchronous systems. In *Proc. of Embedded Software.* Springer, 2002.
13. F. Jebali, F. Lang, and R. Mateescu. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems. *Proc. of ICFEM*, 2014.

14. 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.
15. Avinash Malik, Alain Girault, and Zoran Salcic. Formal semantics, compilation and execution of the GALS programming language DSystemJ. *Parallel and Distributed Systems*, 2012.
16. R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *Proc. of FM*. Springer, 2008.
17. 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. 2005.
18. 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, 2004.
19. J. Proença, D. Clarke, E. De Vink, and F. Arbab. Dreams: a framework for distributed synchronous coordination. In *Proc. of the ACM Symp. on Applied Computing*. ACM, 2012.
20. S. Ramesh, S. Sonalkar, V. Dsilva, N. Chandra, and B. Vijayalakshmi. A toolset for modelling and verification of gals systems. In *Proc. of CAV*. Springer, 2004.
21. M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Proc. of DATE*. IEEE, 2004.
22. H. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P. Le Guernic, A. Toom, and O. Laurent. System-level co-simulation of integrated avionics using Polychrony. In *Proc. of the ACM Symp. on Applied Computing*. ACM, 2011.