A Formal Framework for Modelling and Verifying Globally Asynchronous Locally Synchronous Systems

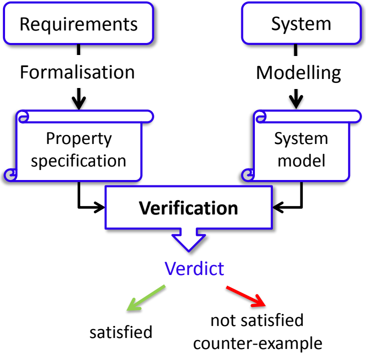
Fatma Jebali

Université Grenoble Alpes
Inria Grenoble – Rhône-Alpes / Convecs
LIG

September 12, 2016

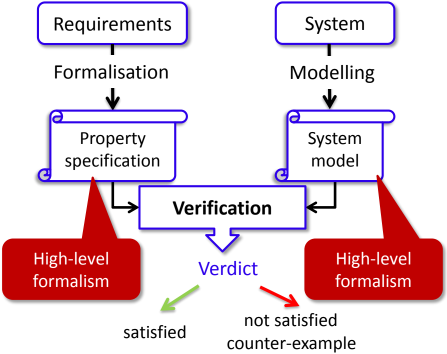
Introduction

- **Goal:** building correct systems
- **Means:** formal methods
- **Technique:** model checking



Introduction

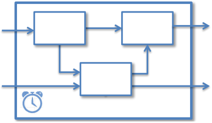
- **Goal:** building correct systems
- **Means:** formal methods
- **Technique:** model checking



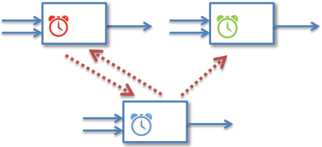
- **Need:** adequate formalisms to write models and properties

Globally Asynchronous Locally Synchronous systems [Chapiro]

- A set of **synchronous components** **composed asynchronously**
- Synchronous components
  - Infinite sequence of zero-delay steps
  - Composition: one shared clock
  - Communication: zero-delay
  - Determinism



- Asynchronous composition of synchronous components
  - Composition: no shared clock
  - Communication: non-zero delay
  - Nondeterminism



- GALS examples: networks-on-chip, flight control systems, networks of Programmable Logic Controllers (PLCs)

Modelling and verifying GALS systems (1)

Problem: which formalisms to model and verify GALS systems?
Solution 1: synchronous languages and corresponding verification tools

Existing approaches	☼	Use a single-clock model (e.g., Lustre)
	☼	Use a mutli-clock model (e.g., Signal, Multiclock Esterel)
Advantages	☺	Built-in constructs for synchrony (synchronous parallel and delay operators, synchrony assumptions)
	☺	Simplicity of usage
Limitations	☹	Only deterministic GALS applications are addressed
	☹	Mainly safety properties are specified

Modelling and verifying GALS systems (2)

Problem: which formalisms to model and verify GALS systems?
Solution 2: asynchronous languages and corresponding verification tools

Existing approaches	☼	Translate GALS languages into input languages of model checkers (e.g., CRSM → Promela, Signal → nuSMV)
	☼	Combine a synchronous language with an asynchronous language (e.g., Signal + Promela, SAM + LNT)
Advantages	☺	Built-in constructs for asynchrony: asynchronous parallel operator, abstraction means (e.g., hiding, nondeterminism)
	☺	General properties (e.g., unbounded liveness, fairness)
		$\Longrightarrow$ **Expressiveness for general GALS systems**
Limitations	☹	Lack of built-in constructs for synchrony
	☹	Complexity of usage $\Rightarrow$ steep learning curve

Our motivation

Goal: Circumvent the limitations of existing approaches to model and verify GALS systems

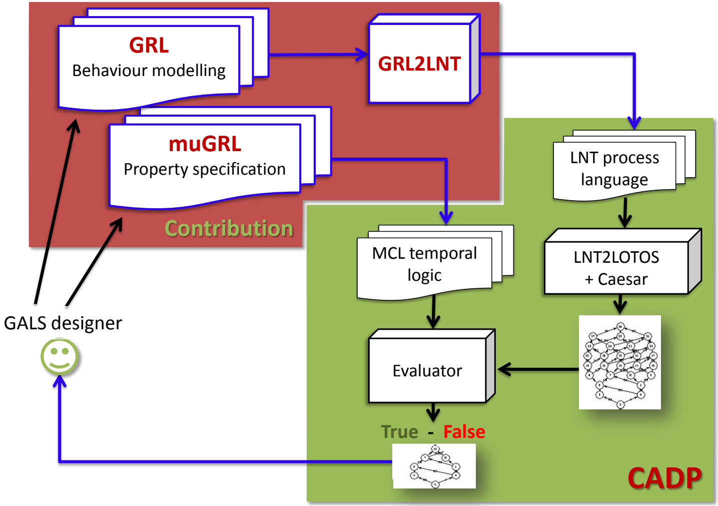
Context: Bluesky project (Minalogic competitiveness cluster, Crouzet Automatismes)

- Existing: a Crouzet synchronous language to design single PLCs (graphical syntax, no formal semantics)
- Contributions:
  - ⇒ Scale up distributed applications based on networks of PLCs (GALS systems)
  - ⇒ Enhance Crouzet design process with formal verification

Approach: Propose *Domain-Specific Languages* as pivot forms to:

- ☺ Connect seamlessly GALS design tools to verification tools
- ☺ Enhance the modularity of the connection
- ☺ Reduce the complexity of usage

Proposed approach: GALS-specific languages



GRL: a language for modelling the behaviour of GALS

GRL in a nutshell

GRL is a new modelling language intended for GALS systems

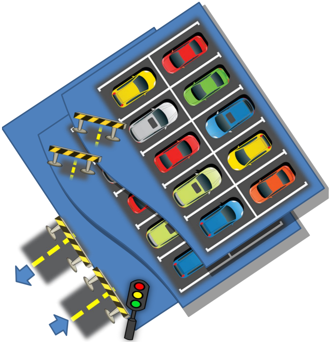
- Rich data structures (e.g., integers, sets, lists)
- **Blocks** denote synchronous components
  → deterministic
  → Locally Synchronous
- **Mediums** are user-defined communication channels
  → asynchronous, nondeterministic
- **Environments** are optional user-defined constraints on blocks to close the system
  → asynchronous, nondeterministic
- **Systems** are a composition of blocks, environments, and mediums
  → Globally Asynchronous

GRL systems

- A process algebraic style is adopted
- Components are composed in **asynchronous parallelism**
  → Interleaving semantics
  → Implicit asynchronous parallel operator
- Communication is done by **message-passing synchronization**
- Hiding mechanism is supported for abstraction
- System behaviour:
  - Blocks evolve **infinitely and independently**
    → pure interleaving
    → **active** components
  - Environments and mediums are **triggered by blocks**
    → pure interleaving
    → **passive** components
- Modelling is compositional
  → several environments and mediums can be plugged

Example: GRL systems

```
-- Car park management system
system Car_Park (Dmd_Out, Dmd_In, (* observable *)
                 Car_Out, Car_In: bool, ...)
is
  var From_Exit, To_Entrance: bool (* non-observable *)
  block list (* PLCs *)
    Exit     (Dmd_Out, ?Car_Out) [?To_Entrance],
    Entrance (Dmd_In, ?Car_In, ...) [From_Exit],
    ...
  environment list (* constraints *)
    Same_Pace_3 (Entrance, Exit, ...),
    Counter     (Car_In, Car_Out, ?Dmd_Out)
  medium list (* asynchronous communication *)
    Unreliable [?From_Exit, To_Entrance]
end system
```



GRL blocks

- The synchronous loop is built-in
- The code of a block describes an execution step
  1. Read inputs
  2. Deterministically compute outputs and next internal state
- The internal state is explicit and represented by static variables
- Computation is instantaneous
- Example:

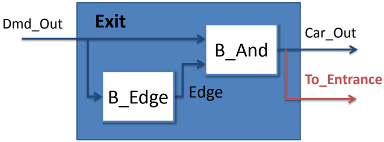
```
-- GRL code
block B_Edge (in Signal: bool,
              out Edge: bool) is
  static var pre_Signal: bool := false
  Edge := Signal and not (pre_Signal);
  pre_Signal := Signal
end block
```

```
-- Corresponding Lustre node
node B_Edge (Signal: bool)
  returns (Edge: bool);
  let
    Edge := Signal and
            not (pre (Signal))
  tel
```

Synchronous composition of blocks

- Blocks can be composed inside other blocks hierarchically
- There is no synchronous parallel operator
- The order of block invocations should be:
  - → either specified by the user
  - → or inferred by a front-end compiler
- Dataflow communication is adopted (zero-delay)
- Example:

```
block Exit
  (in    Dmd_Out      : bool, (* request leaving *)
   out   Car_Out      : bool) (* grant a request *)
  [send To_Entrance: bool] (* inform Entrance *)
is
  var Edge: bool  (* links *)
  B_Edge (Dmd_Out, ?Edge);
  B_And  (Edge, Dmd_Out, ?Car_Out);
  To_Entrance := Car_Out
end block
```

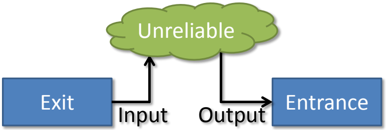


GRL mediums

- Mediums describe the asynchronous communication between blocks
- A medium interacts with one or several blocks
  → Each interaction is either a reception or an emission of tuples of values (messages)
  → Received messages are buffered in the medium state before being emitted
- Explicit nondeterministic statements are supported
- Mediums are user-defined
  → Various buffering mechanisms and communication protocols can be described

Example: unreliable communication medium

```
-------------------------------------------------------------------------
-- Communication between the Exit and the Entrance PLCs
medium Unreliable [receive Input: bool, send Output: bool] is
  static var buffer: bool := false -- one-place buffer
  select
     when ?Input -> select
                      buffer := Input -- buffering
                    [] null           -- loss
                    end select
  [] when Output -> Output := buffer
  end select
end medium
-------------------------------------------------------------------------
```



GRL environments

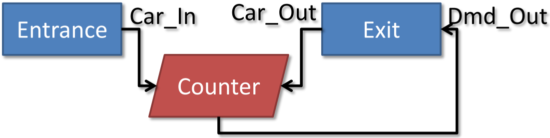
- Environments describe constraints on the behaviour of blocks
- Two types of constraints are possible:
  - **Data constraints** concern the values carried by block inputs
  - **Activation constraints** concern the execution (also called activation) of bocks
- Explicit nondeterministic statements are supported
- Combining activation and data constraints is possible
  → Complex constraints can be described, e.g., test case scenarios

GRL environments: data constraints

- Environments produce block inputs and react to block outputs
- Constraints on the inputs of one or several blocks can be described
  → The value of a block input may depend on the past values carried by inputs and outputs of other blocks
  → Past values are stored in the environment state
- Data constraints are similar to, but more general than, assertions in synchronous languages

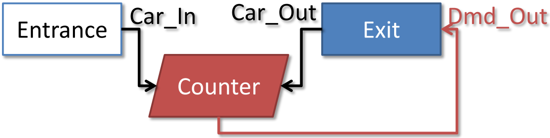
Example: data constraints

```
-- If the car park is empty, no leaving request is possible
environment Counter (in  Car_In: bool,    (* car entering *)
                     in  Car_Out: bool,   (* car leaving  *)
                     out Dmd_Out: bool)   (* leaving request *)
is
  static var cars: nat := 0 (* actual number of cars *)
  select
     when ?Car_In  -> if Car_In  then cars := cars + 1 end if
  [] when ?Car_Out -> if Car_Out then cars := cars - 1 end if
  [] when  Dmd_Out -> if (cars == 0) then Dmd_Out := false else Dmd_Out := any bool end if
  end select
end environment
```



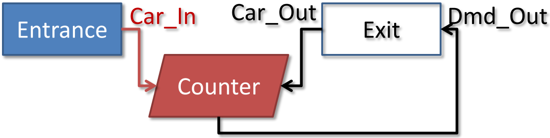
Example: data constraints

```
-- If the car park is empty, no leaving request is possible
environment Counter (in  Car_In: bool,    (* car entering *)
                     in  Car_Out: bool,   (* car leaving  *)
                     out Dmd_Out: bool)   (* leaving request *)
is
  static var cars: nat := 0 (* actual number of cars *)
  select
     when ?Car_In  -> if Car_In  then cars := cars + 1 end if
  [] when ?Car_Out -> if Car_Out then cars := cars - 1 end if
  [] when  Dmd_Out -> if (cars == 0) then Dmd_Out := false else Dmd_Out := any bool end if
  end select
end environment
```



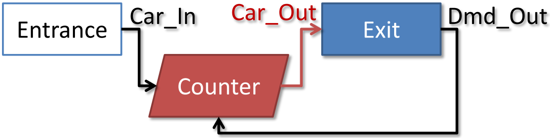
Example: data constraints

```
-- If the car park is empty, no leaving request is possible
environment Counter (in  Car_In: bool,    (* car entering *)
                     in  Car_Out: bool,   (* car leaving  *)
                     out Dmd_Out: bool)   (* leaving request *)
is
  static var cars: nat := 0 (* actual number of cars *)
  select
     when ?Car_In  -> if Car_In  then cars := cars + 1 end if
  [] when ?Car_Out -> if Car_Out then cars := cars - 1 end if
  [] when  Dmd_Out -> if (cars == 0) then Dmd_Out := false else Dmd_Out := any bool end if
  end select
end environment
```



Example: data constraints

```
-- If the car park is empty, no leaving request is possible
environment Counter (in  Car_In: bool,    (* car entering *)
                     in  Car_Out: bool,   (* car leaving  *)
                     out Dmd_Out: bool)   (* leaving request *)
is
  static var cars: nat := 0 (* actual number of cars *)
  select
     when ?Car_In  -> if Car_In  then cars := cars + 1 end if
  [] when ?Car_Out -> if Car_Out then cars := cars - 1 end if
  [] when  Dmd_Out -> if (cars == 0) then Dmd_Out := false else Dmd_Out := any bool end if
  end select
end environment
```



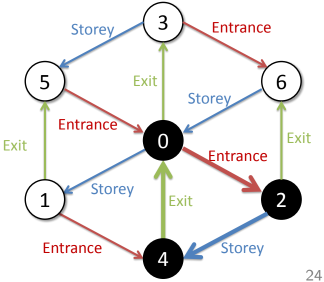
GRL environments: activation constraints

- Environments control the degree of asynchrony in block composition
- Constraints on the activations of one or several blocks can be described
  → They permit or deny block activations at specific moments of the system execution
  → The history of block activations is stored in the environment state
- Various activation strategies can be implemented
  → Abstract real-time properties in an asynchronous model
  → e.g., relations between block paces, priorities, failure

Example: activation constraints

```
-- Blocks Entrance, Exit, and Storey evolve at almost the pace
environment Same_Pace_3 (block Entrance, Exit, Storey) is
  static var ok_N, ok_X, ok_S: bool := true
  select
       if ok_N then enable Entrance; ok_N := false end if (* 1 *)
  []   if ok_X then enable Exit;     ok_X := false end if (* 3 *)
  []   if ok_S then enable Storey;   ok_S := false end if (* 2 *)
  end select;
  if not (ok_N or ok_X or ok_S) then ok_N := true; ok_X := true; ok_S := true end if
end environment
```

The activation strategy of blocks Entrance, Exit, and Storey, induced by the environment:



Semantics of GRL

- 140 static semantic rules (typing, binding, initialisation)
  → Reject syntactically correct but semantically incorrect programs
- 24 dynamic semantic rules
  → Formally defined (Structural Operational Semantics)
- Systems are represented by Labelled Transition Systems (LTSs)
  - States correspond to static variables
  - Transitions correspond to blocks steps (Esterel-like)
  - A transition label indicates:
    → The block under execution
    → Its observable interactions with the outside world (process algebra)
  - A system LTS is the interleaving of the possible transitions corresponding to the system blocks

Example: Semantics of a GRL block

```
-- GRL code
block Exit (in Dmd_Out: bool, out Car_Out: bool) [send To_Entrance: bool] is
   ...
end block
```

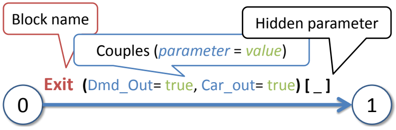


Figure a: One step of block Exit

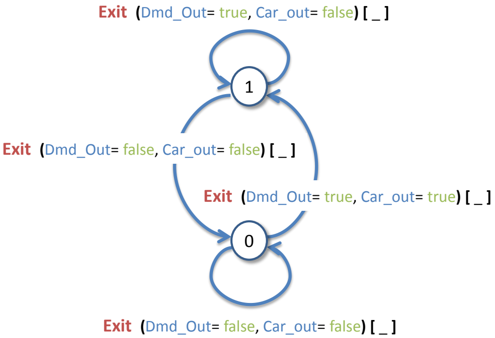


Figure b: The behaviour of block Exit

Example: Semantics of a GRL system

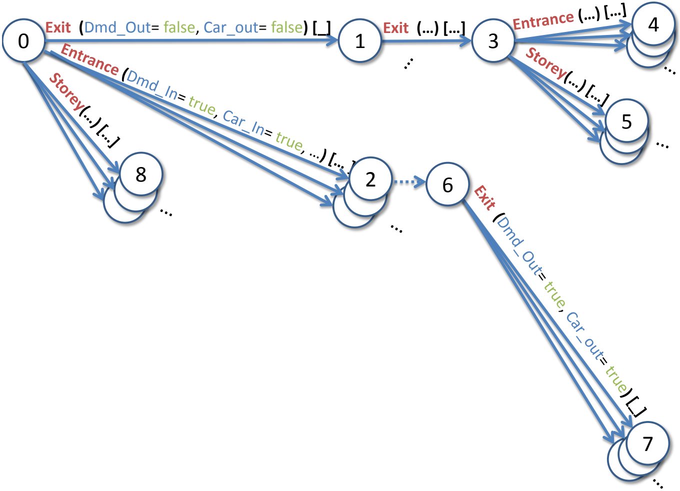


Figure c: The behaviour of system Car_Park (excerpt)

GRL vs. Existing GALS approaches

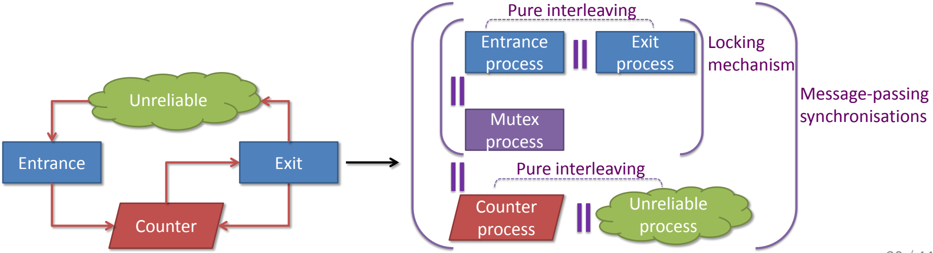
	Rich data types	Data constraints	Activation constraints	User-defined mediums (with nondeterminism)
CRSM	✗	✗	✓	✗
SystemJ	✓	✗	✓	✗
Signal+Promela	✓	✗	✓	✗
SAM+LNT	✓	✗	✗	✓
GRL	✓	✓	✓	✓

Translation from GRL into LNT

Translation of systems

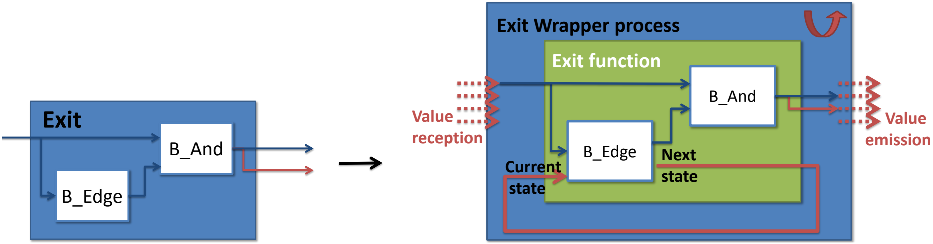
- GRL systems ⟶ LNT processes

  GRL top-level blocks, mediums, environments ⟶ LNT processes
- Block processes interleave
  → A locking mechanism ensures their atomicity
- Medium and environment processes interleave
- Message-passing synchronisations are done between different processes
- Example:



Translation of blocks

- GRL blocks $\longrightarrow$ LNT functions implementing one step
- GRL top-level blocks $\longrightarrow$ LNT wrapper processes [Garavel]
  - Implement the block synchronous loop:
    (1) Value reception, (2) LNT function call, (3) Value emission
- GRL internal state $\longrightarrow$ LNT local variables
  - Declared before the synchronous loop
  - Propagated to functions as **in out** parameters
- Example:

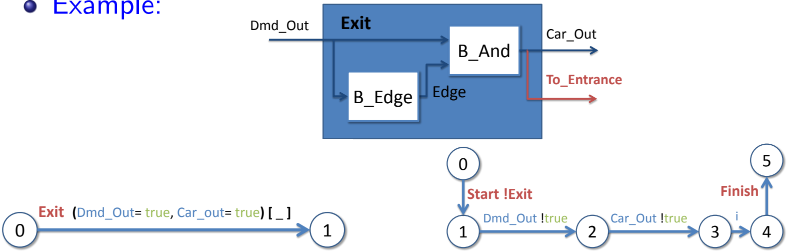


Translation of environments and mediums

- GRL environments & mediums $\longrightarrow$ LNT processes
- GRL data signals $\longrightarrow$ LNT communication actions with message-passing
- GRL activation signals $\longrightarrow$ LNT communication actions synchronizing with block processes and the lock
- GRL internal state is translated in the same way as in block translation

Tool support

- GRL2LNT is a tool implementing:
  - GRL static semantics rules
  - The translation function from GRL to LNT
  - → 30,000 lines of Syntax & Lotos NT code
  - → Tested on 555 GRL programs
- Each GRL transition ⟶ an LNT transition sequence
  → Linear expansion in the number of transitions (locking mechanism)
- Example:



GRL transition ⟶ Atomic sequence of LNT transitions

The muGRL property patterns

- muGRL is a set of property patterns for GALS systems
  $\rightarrow$ 42 patterns
- It aims at reducing the complexity of using temporal logics
- Property patterns include:
  - General property patterns (safety, liveness, fairness)
  - GALS property patterns
  - Discrete real-time property patterns
- They are are translated into MCL
- The interpretation model is the LTSs of GRL2LNT

GALS property patterns (1)

- GALS property patterns include deadlocks (activation, data), livelocks (activation, data), and instability [Caspi]
- Example 1: data deadlock

"For block Exit, inputs and outputs carry infinitely the same values"

muGRL pattern	Translation into MCL
Idle (Dmd_Out)	[**true**∗.{Dmd_Out ?x:**bool**}. **true**∗.{Dmd_Out ?y:**bool** **where** x <> y}] **false**
All_Idle (Dmd_Out, Car_Out)	**Idle** (Dmd_Out) **and** **Idle** (Car_Out)

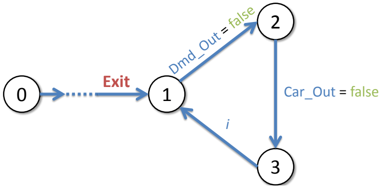


Figure a: Data deadlock situation in block Exit behaviour

GALS property patterns (2)

- Example 2: stability

  "For block Entrance, if input Dmd_In stabilise, i.e., carry infinitely the same value, output Car_In should stabilise in the future"

muGRL pattern	Translation into MCL
Stability (Dmd_In, Car_In)	[**true**∗.{Dmd_In ?x:bool}. **true**∗.{Dmd_In ?y:bool where x <> y}] **false implies not** < **true**∗.{Car_In ?v:bool}. **true**∗.{Car_In ?w:bool where v <> w} > @

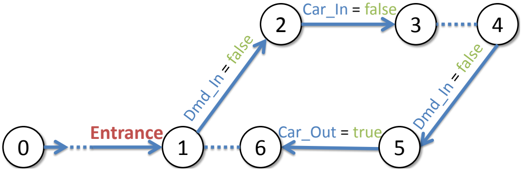


Figure b: Instability situation in block Entrance behaviour

Discrete real-time property patterns

- Discrete real-time property patterns include deadline, event sustain, and boundedness
- Example: boundedness

  "Between two successive activations of Exit, Entrance is activated at most twice"

muGRL pattern	Translation into MCL
Not_To_Unless_More (**true**∗. {Exit}, {Exit}, {Entrance}, 2)	[**true**∗ . {Exit} . (**not** ({Entrance}))∗ . ({Entrance} . (**not** ({entrance} **or** {Exit}))∗){3}] **false**

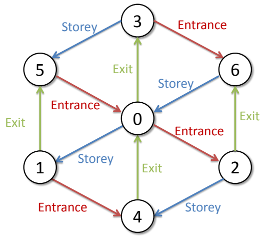


Figure c: Bounded activation ensured

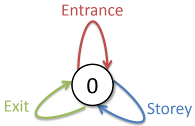


Figure d: Bounded activation not ensured

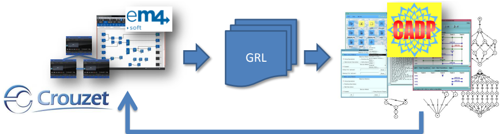
Real-life applications

AutoFlight Control Systems

- Work with IRT Saint-Exupéry, Thalès Avionics
- Apply our work on systems with **strict real-time constraints**
- Approximate real-time constraints in GRL
  - On synchronous blocks, by counting block steps
  - On asynchronous systems, by implementing activation strategies
- Altitude target acquisition function
  - 571 lines of GRL, 1988 lines of LNT, 438 line verification script
  - Tractable state spaces (20 million states, 30 million transitions)
- Results
  - GRL, muGRL, and CADP are appropriate for theses systems
  - Promising results on the verification of real-time properties (comparison with the Tina toolbox)
  - GRL will be evaluated at IRT Saint-Exupéry

Networks of PLCs

- Work with Crouzet Automatismes, Bluesky project
- Apply our approach on systems with **no real-time guaranties**
  - High degree of asynchronous parallelism
  - Unreliable communication
- Car park, among several distributed applications
  - 463 lines of GRL, 1187 lines of LNT, 391 line verification script
  - Large state spaces (800 million states, 1 billion transitions)
- Results
  - A compiler from Crouzet design language into GRL is developed by Crouzet
  - Libraries of reusable GRL components are built
  - Crouzet investigates to use GRL as end-user language



Conclusion

Conclusion: contributions

- Combine principles of synchronous languages and (asynchronous) process algebra into a single, coherent language: GRL
- Define property patterns to reduce the complexity of using temporal logics: muGRL
- Equip GRL and muGRL with verification tools by mapping to the LNT and MCL languages supported by the CADP toolbox
- Apply GRL and muGRL to realistic GALS problems and connect GRL to industrial tools for PLCs
- Positive feedback from industrial GALS users

Conclusion: future work

- Prove the translation function from GRL into LNT
- Explore more CADP techniques, e.g., probabilistic and compositional verification
- Use GRL to connect synchronous languages, e.g., Lustre, to CADP
- Experiment other real-life applications

References

1. **F. Jebali**, F. Lang, R. Mateescu. Formal Modelling and Verification of GALS Systems Using GRL and CADP. *Formal Aspects of Computing*, Springer Verlag, 2016, 28 (5), pp.767-804.
2. **F. Jebali**, F. Lang, and R. Mateescu. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems. *Proceedings of the 16th International Conference on Formal Engineering Methods (ICFEM'14)*, Luxembourg. Springer, 8829, pp.219-234, 2014, LNCS.
3. **F. Jebali** et al. Modélisation et validation formelle de systèmes globalement asynchrones et localement synchrones. *Approches Formelles dans l'Assistance au Développement de Logiciels*, 2014, Paris, France. pp.97–102, 2014.
4. **F. Jebali**, F. Lang, and R. Mateescu. GRL: A Specification Language for Globally Asynchronous Locally Synchronous Systems (Syntax and Formal Semantics). *Research Report* 8527, Inria, 84 pages, 2014.