Asynchronous Testing of Synchronous Components in GALS Systems

Lina Marsso, Radu Mateescu, Ioannis Parissis, Wendelin Serwe

Asynchronous Testing of Synchronous Components in GALS Systems

Lina Marsso[1], Radu Mateescu[1], Ioannis Parissis[2], and Wendelin Serwe[1]

[1] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, 38000 Grenoble, France
[2] Univ. Grenoble Alpes, Grenoble INP*, LCIS, 26000 Valence, France

Abstract. GALS (Globally Asynchronous Locally Synchronous) systems, such as the Internet of Things or autonomous cars, integrate reactive synchronous components that interact asynchronously. The complexity induced by combining synchronous and asynchronous aspects makes GALS systems difficult to develop and debug. Ensuring their functional correctness and reliability requires rigorous design methodologies, based on formal methods and assisted by validation tools. In this paper we propose a testing methodology for GALS systems integrating: (1) synchronous and asynchronous concurrent models; (2) functional unit testing and behavioral conformance testing; and (3) various formal methods and their tool equipments. We leverage the conformance test generation for asynchronous systems to automatically derive realistic scenarios (input constraints and oracle), which are necessary ingredients for the unit testing of individual synchronous components, and are difficult and error-prone to design manually. We illustrate our approach on a simple, but relevant example inspired by autonomous cars.

1 Introduction

A reactive system controls its environment by observing it (via input sensors) and modifying it (via output commands) to obtain the desired behavior. A simple example is the control of the room temperature by commanding a heater. The synchronous approach [16] to reactive systems programming supposes that the system operates triggered by a clock, such that at each clock instant the system instantaneously computes the outputs from the inputs and its internal state. The simplicity of this abstraction is the reason for the success of the synchronous approach, which has been widely used for over two decades for the design and analysis of safety critical systems in various application domains (e.g., avionics, railway transportation, nuclear plants, etc.).

However, modern complex systems, such as autonomous cars or the Internet of Things, are increasingly large and distributed, consisting of multiple components that execute independently and interact with each other. As the assumption of a global clock is less realistic, these systems are better described as GALS (Globally Asynchronous, Locally Synchronous) [4,36], i.e., composed of concurrent synchronous reactive systems interacting with each other asynchronously,

* Institute of Engineering Univ. Grenoble Alpes

by means of message passing with non-zero communication delays. For instance, in an autonomous car, the perception devices (radar, lidar, cameras, etc.) and the engine controls are separate components, located in various places of the car, operating independently and connected through communication links. Each of these components might be considered and implemented as a synchronous reactive system, but the complete autonomous car is rather a GALS system. Notice that the GALS approach allows the smooth integration and reuse of existing and time-proven synchronous components when designing new systems that no longer fit into the framework of synchronous programming.

GALS systems are intrinsically complex, and the simultaneous presence of synchronous and asynchronous aspects makes their development and debugging difficult. Hence, to ensure their functional correctness and reliability, it is necessary to follow a rigorous design approach, based on formal methods and assisted by efficient validation and verification tools. Such design approaches and tools exist for the separate modeling and analysis of synchronous or asynchronous parts of a GALS system, but their integration can further improve the effective design of a GALS system.

In this paper we propose a testing methodology for GALS systems integrating: (1) synchronous and asynchronous concurrent models, (2) functional unit testing and behavioral conformance testing, and (3) various formal methods and their tool equipments. The idea is to exploit the information gathered by the analysis of a globally asynchronous system to automate and finely tune the analysis of its individual synchronous components. First, we model the GALS system in GRL (GALS Representation Language) [25,24], a formal language connected to the CADP verification toolbox[3] [10] for asynchronous systems. Using CADP, we validate the asynchronous aspects of the GRL model, e.g., by checking temporal logic formulas expressing desired (global) correctness properties. Next, we use TESTOR [27] to automatically generate conformance tests, which can be used to assess whether an actual implementation of the GALS system conforms to the GRL model. Then, we project such a conformance test on a synchronous component C and translate it automatically into a scenario (i.e., input constraints in Lutin [34] and an oracle in Lustre [18]), required to automate the testing of C using Lurette [22], the test generation tool of the Lustre V6 toolbox[4] [18,34]. Because these scenarios are automatically generated from the GRL model of the GALS system, they correspond by construction to relevant (and often complex) executions of the synchronous component.

We illustrate our approach on a simple, but relevant example, namely an autonomous car, which has to reach a destination, following roads on a map, in the presence of moving obstacles. The car is modeled as a GALS system, comprising synchronous components for perception, decision, and action.

The remainder of the paper is organized as follows. In Section 2, we introduce the formal GRL model of an autonomous car. In Section 3, we describe the application of analysis techniques for asynchronous systems to the overall

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

[4] http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/

GALS system, in particular model checking and conformance test generation. In Section 4, we recall techniques for the functional testing of synchronous components. In Section 5, we present the main contribution of this paper, namely the integration of asynchronous conformance testing with automated synchronous testing to improve the latter. In Section 6, we compare our approach to existing GALS validation approaches. Finally, in Section 7 we give concluding remarks and suggest future research directions.

2 GRL Model of an Autonomous Car

To illustrate our approach, we consider the behavioral model of a (simplified) autonomous car interacting with its physical environment, i.e., a given set of moving obstacles (pedestrians, cyclists, other cars, etc.) evolving with the car on a (geographical) map. To limit the complexity, each obstacle executes a fixed number of random or statically chosen movements. The autonomous car itself consists of four synchronous components:

(i) a *GPS* keeps the current position of the car updated,
(ii) a *radar* detects the presence of the obstacles close to the car and builds a perception grid summarizing information about perceived obstacles,
(iii) a *decision* (or trajectory) controller computes an itinerary from the current position to the destination, avoiding streets containing obstacles, and
(iv) an *action* controller commands the engine and direction to follow the itinerary computed by the decision controller, using the perception grid built by the radar to avoid collisions.

These four components communicate in various ways:

– the GPS sends the current position to the decision controller upon request,
– the radar periodically sends the perception grid to the action controller, and
– the action controller requests a new itinerary from the decision controller.

GRL (GALS Representation Language) [25,24] is a formal language designed to model GALS systems. It integrates the synchronous reactive model underlying dataflow languages and the asynchronous interleaving semantics of concurrency underlying process algebras. Figure 1 shows the architecture of our GRL model of the autonomous car. Each synchronous component (`ACTION`, `RADAR`, `DECISION`, and `GPS`) is represented in GRL as a **block**, depicted as a (light blue) rectangle with solid border in Figure 1. These blocks exchange data via asynchronous communication media (`POSITION`, `PATH`, `CURRENT_GRID`), each of which is represented in GRL as a **medium**, depicted as a (pink) ellipse with dashed border in Figure 1. The interaction between blocks also respects constraints (`MAP_MANAGEMENT`), each one being represented in GRL as an **environment**, depicted as a (light pink) ellipse with thick dashed border in Figure 1. The overall model of the car is

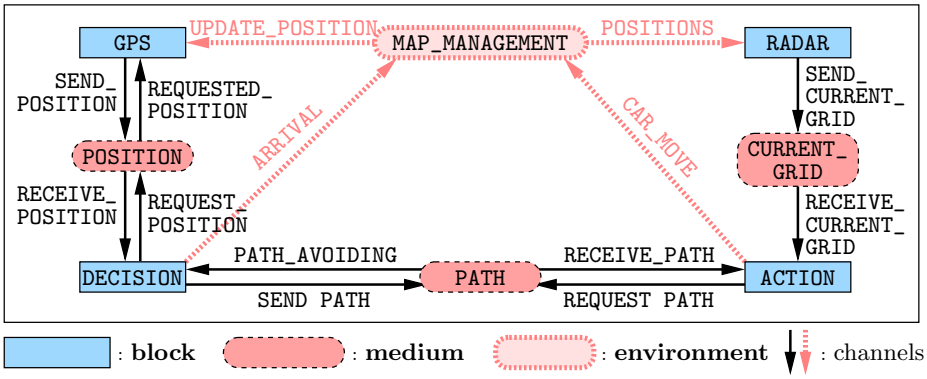

Fig. 1. Architecture of the GRL model of an autonomous car

represented in GRL as a **system**, which describes the composition and interactions of blocks, media, and environments. In the sequel, we present excerpts of a block, a medium, and an environment of our GRL model (1271 lines).[5]

The map is represented as a directed graph, in which edges correspond to streets and nodes correspond to crossroads; for simplicity, we assume that the car or an obstacle occupies a street completely (a longer street can be represented by several edges in the graph). A set of functions is defined to explore this graph, to compute itineraries, etc. The GRL model is instantiated by providing global constants encoding the map, the initial position and destination of the car, and the set of obstacles with their initial positions and lists of movements.

A GRL **block** defines the deterministic code executed by the synchronous component at each *activation* (i.e., clock instant). For instance, the radar of our autonomous car is modeled by the GRL block `RADAR`, which has a **static** variable `previous_grid` to keep track of the perception grid computed during the previous activation. This grid is considered to be initially empty (i.e., it has the value `Grid (NIL)`). At each activation, the radar receives the current positions of the car and all the obstacles as input from the environment (**in** parameter `POSITIONS`).[6] It then computes the current perception `grid` indicating, for each possible direction the car might take, whether at least `radar_visibility` steps are free of any obstacle. If there is a change between `previous_grid` and `grid`, both the variable `previous_grid` and the output `CURRENT` are updated; otherwise the output is set to the particular value `already_sent` indicating that the grid did not change. At the end of the activation, the computed grid is sent to the connected medium (**send** parameter `CURRENT`).

[5] The complete GRL model, test purpose, the MCL properties, SVL and XTL scripts, and other resources related to the example are available as a demo in the GRL distribution at `http://convecs.inria.fr/software/grl`.

[6] Currently, for the interaction between a block and an environment, GRL requires to wrap several inputs or outputs into a single structured input or output.

```
block RADAR (in POSITIONS: Car_Obstacle_Pos) [send CURRENT: Grid] is
  static var previous_grid: Grid := Grid (NIL)
  var grid: Grid
  grid := perception (POSITIONS, radar_visibility);
  if grid != previous_grid then
    previous_grid := grid;
    CURRENT := grid
  else
    CURRENT := Grid (already_sent)
  end if
end block
```

A GRL **medium** enables (asynchronous) interaction between synchronous blocks. Explicitly representing media makes it possible to finely model a large panel of behaviors (i.e., message buffering, message loss, nondeterminism, etc.). A medium is connected to each block by at most two channels, called **receive** and **send** channels. Note that a **receive** channel corresponds to the reception of some value in a variable prefixed by "?". Each channel has an associated Boolean condition (tested with a **when** clause), stating whether a message is available. For instance, the following GRL medium `CURRENT_GRID` enables the block `RADAR` to send the current perception grid (via the **receive** channel `INPUT`) to the block `ACTION` (via the **send** channel `OUTPUT`); the transmission takes place only when the perception grid has not already been sent.

```
medium CURRENT_GRID [receive INPUT: Radar_Grid,
                     send OUTPUT: Radar_Grid] is
  static var buffer: Radar_Grid := Grid (NIL)
  select
    when ?INPUT ->
      if INPUT != Grid (already_sent) then buffer := INPUT end if
  [] when OUTPUT -> OUTPUT := buffer
  end select
end medium
```

A GRL **environment** provides blocks with inputs and receives their outputs. Block activations are particular inputs, enabling an environment to precisely control the activations of synchronous blocks, e.g., to control the relative clock speeds and/or drifts. The following fragment of the GRL environment `MAP_MANAGEMENT` ensures that: (1) the positions of the car (`map.c`) and obstacles (`grid`) are shared with the block `RADAR` (by sending this information to `RADAR` as input `POSITIONS`); (2) this information is updated when the car or the obstacles move (by receiving these moves from blocks `ACTION` and `RADAR` as outputs `CAR_MOVE` and `OBSTACLE_MOVE`, respectively); and (3) the blocks are only activated as long as the car neither arrived at destination, nor crashed. Note that an environment may be nondeterministic, e.g., it may contain nondeterministic choices, modelled in GRL using the **select** statement.

```
environment MAP_MANAGEMENT (block RADAR, ...
                            in OBSTACLE_MOVE: Obstacle,
                            in CAR_MOVE: Control, ...
```

```
                              out POSITIONS: Car_Obstacle_Pos, ...) is
  static var grid: Grid := Grid (NIL),
        map: Localization := Localization (initial_street, initial_map),
        crash: Bool := false, car_arrived: Bool := false, ..
  var collison_detected: Bool, ...
  if not (crash or car_arrived) then
    select
      -- send updated inputs (car and obstacle positions) to the radar
      when POSITIONS -> POSITIONS := pos (map.c, grid)
    [] -- car movement
      when ?CAR_MOVE ->
        -- update car position in the map
        map := move_car (map, CAR_MOVE);
        -- check for collisions (car and an obstacle on the same street)
        collision_detected := intersection (grid, map);
        if collision_detected then crash := true end if
    [] -- potential obstacle movement
      when ?OBSTACLE_MOVE ->
        if OBSTACLE_MOVE != null_obstacle then
          -- update obstacle positions in the grid
          grid := move_obstacle_grid (grid, OBSTACLE_MOVE)
        else
          -- no effective movement
          grid := grid
        end if
    ...
    end select
  end if
end environment
```

GRL defines the semantics of a GALS system as an LTS (Labeled Transition System), whose states represent the memories of all blocks, media, and environments of the system [24, Chapter 4]. The initial state is the initial memory, in which each static variable has its (mandatory) initial value. Each transition going out of a state corresponds to the atomic execution of a block, which consists in reading the values of input and receive channels, executing the code of that block activation, and producing the values for the outputs and send channels of the block. The transition is labeled with all these values, and its target state corresponds to the updated memories of the participating components, i.e., the block and its connected mediums and environments. Thus, the atomic executions of synchronous blocks are interleaved in the LTS, which reflects the GALS nature of the system.

For technical reasons, we rely in this paper on the semantics of GRL as induced by the current translation-based implementation of GRL (see below). In this semantics [24, Chapter 5], the execution of a synchronous block activation is split into an input transition followed by an output transition. These two transitions are executed atomically, i.e., without interleaving of any other transitions corresponding to an activation of another block.

3 Model Checking and Conformance Test Generation

When considering a complete GALS system from the outside, all parts based on the synchronous programming paradigm are hidden. Thus, the overall GALS system is amenable for classic analysis techniques developed for asynchronous systems. GRL is equipped with the GRL2LNT [25,24] translator to LNT [11], the modern formal modeling language recommended as input for the CADP verification toolbox [10]. Using GRL2LNT and CADP, for a map (with 22 streets and 8 crossroads) and two obstacles, each with a first random movement and a second statically chosen movement, we generated (in about 4 minutes on a standard laptop) the LTS corresponding to our GRL autonomous car model (3, 568, 781 states and 5, 619, 802 transitions; 287, 103 states and 406, 780 transitions after strong bisimulation minimization).

3.1 Model Checking

We first validated our GRL model by checking several safety and liveness properties characterizing the correct behavior of the autonomous car. We expressed the properties in MCL [29], which is the data-handling, action-based, branching-time temporal logic of the on-the-fly model checker of CADP. For simplicity, we describe here the properties in natural language, only giving and commenting the MCL code of the first one to illustrate the flavor.[5]

– The position of the car is correctly updated after any movement of the car. This safety property specifies that on all transition sequences, an update of the car position (action "`UPDATE_POSITION ?current_street`", where `current_street` is the street on which the car is) followed by a car movement (action "`CAR_MOVE ?control`", where `control` is a movement command) must be followed by an update of the car position consistent with `current_street`, `control`, and the map. This can be expressed in MCL using the necessity modality below, which forbids the transition sequences containing inconsistent position updates:

```
[ true* .
   { UPDATE_POSITION ?current_street:String } .
   (not ({ CAR_MOVE ... } or { UPDATE_POSITION ... }))* .
   { CAR_MOVE ?control:String } .
   (not ({ CAR_MOVE ... } or { UPDATE_POSITION ... }))* .
   { UPDATE_POSITION ?new_street:String where
     not (Consistent_Move (current_street, control, new_street)) }
] false
```

The values of the current position, movement, and new position of the car present on the actions `UPDATE_POSITION` and `CAR_MOVE` are captured in the variables `current_street`, `control`, and `new_street` of the corresponding action predicates (surrounded by `{ }`) and reused in the **where** clause of the last action predicate. The predicate `Consistent_Move` defines all valid combinations for `current_street`, `control`, and `new_street` allowed by the map.

– A same message from one of the autonomous car components must be considered only once. For instance, the radar should not send twice the same perception grid, i.e., two successive occurrences of action `SEND_CURRENT_GRID` must carry different values of the grid, reflecting the changes in perception due to obstacle or car movements.
– Inevitably (by avoiding the non-progressing iterations of the synchronous blocks), the system should reach a state where either the car arrived, a collision occurred between the car and an obstacle, or all obstacles have finished their list of movements.

3.2 Conformance Test Generation

Conformance testing aims at establishing whether an SUT (System Under Test) conforms to a formal model, i.e., whether the SUT is an implementation of the model. For concurrent asynchronous systems, both the formal model and the SUT are represented as IOLTSs (Input-Output LTSs), i.e., LTSs whose actions are separated into controllable inputs and observable outputs, even if the IOLTS of the SUT is not necessarily known. A popular relation used for conformance testing is **ioco** [37], which specifies that an SUT conforms to a model if after executing each trace of the model, the SUT exhibits only the outputs and quiescences (i.e., deadlocks, outputlocks, and livelocks) allowed by the model. A classical conformance testing approach consists in deriving from the formal model a suite of *test cases*, which are then executed on the SUT to check the conformance using black-box testing techniques. This approach is based on the hypothesis of synchronous communication between tester and SUT; adaptations are required if this hypothesis is not satisfied (e.g., in the case of remote interaction using buffered channels) [32,13]. To focus the testing process and help the test-case extraction, *test purposes* are used to describe the goal states, i.e., those states of the model to be reached during execution of the test case on the SUT. Executing a test case may produce one of three possible verdicts: *pass* when a goal state has been reached, *fail* when the observed behavior of the SUT is not conform to the model, and *inconclusive* if the goal states become unreachable.

The online-testing tool TESTOR [27] is capable of extracting controllable test cases on the fly. Concretely, starting from the GRL model of a GALS system, a description of the input actions, and a test purpose, TESTOR explores the model and generates automatically a set of test cases or a *complete test graph* (CTG) [23] to be executed on a physical implementation of the system. Intuitively, a CTG denotes a set of traces containing visible actions and quiescence that should be executable by the SUT to assess its conformance with the model and a test purpose. The quiescence is represented in the CTG as self-loops labeled by a special output action δ on the quiescent states. TESTOR handles possibly nondeterministic models and ensures by construction that the complete test graph provides only inputs that allow to reach a goal state (if possible).

For the GRL model of the autonomous car (see Section 2), the only controllable inputs are the movements of obstacles; the observable outputs make it possible to study the behavior of the car. An example test purpose ($T2$) is to

specify the situation where a collision occurs between a car and an obstacle. The test purpose is expressed as an LTS (in the AUT format), where a transition representing the collision (action `COLLISION`) should lead to a goal state, i.e,, having an outgoing transition labelled by an `ACCEPT` action. We also defined four other examples of test purposes ($T1$, $T3$, $T4$, and $T5$) constraining the car and obstacles interaction on the map. For these test purposes, TESTOR generates the complete test graphs in less than a minute (see Table 1).

4 Testing Techniques for Synchronous Components

Functional testing of a reactive system requires to provide a *sequence* of inputs and observe the corresponding sequence of outputs. In general, the previously observed outputs must be taken into account when computing the next input, so as to provide the reactive system with a realistic behavior of the physical environment the reactive system is controlling. Furthermore, the decision about success or failure of a test also requires the sequence of (input, output) pairs, because the reactive system might change its function to adapt to its environment. Hence, testing a reactive system requires specific techniques and tool support to automate the testing process [21].

In this section, we briefly present the testing tool Lurette [22] for synchronous programs. Using formal specifications of the input constraints and an oracle, Lurette automates both, the generation of appropriate inputs for the SUT and the decision about the test result. Lurette takes three inputs:

(i) a specification in Lutin [34] to dynamically constrain the inputs;
(ii) an oracle in Lustre [18] implementing the test decision; and
(iii) some parameters controlling the execution and the coverage-computation of the generated and executed input sequences.

Lurette interacts with the SUT, generates the input sequences with their corresponding outputs in a file, and displays the test decisions.

We illustrate the usage of Lurette to test an implementation of a radar in the C language, which might be a part of an implementation of our GALS autonomous car example described in Section 2. Usually, a radar builds an occupancy grid with respect to its position (this grid reflects the visibility of the radar); in an autonomous car, the position of the radar is the position of the car. To simplify, in our example, the radar takes as input the position of the car and the obstacles (provided by the GRL environment on channel `POSITIONS`), and the radar outputs the perception grid (sent to the medium `CURRENT_GRID`).

As in Section 3, we consider a fixed instance with two obstacles (`leo` and `lilly`). Testing the synchronous radar component consists in providing a sequence of inputs and observing the generated sequence of outputs. Each of the radar's inputs (position of the car and the obstacles) can take one out of the 21 streets of the map, yielding 21^3 possible inputs. Fortunately, not all sequences of these inputs are realistic, because the car is not flying and has to respect

the constraints of the map. However, relevant tests of the radar should include situations where the radar detects obstacles.

The input constraints for the radar should enforce that the positions of the car and the obstacles evolve in a realistic manner, i.e., respecting the map. The following Lutin code corresponds to a simple scenario, where the car starts on street 9 and possibly moves to street 12, and `lilly` (respectively, `leo`) appears on street 5 (respectively, street 14) and moves back and forth to street 12 (respectively, street 11). This scenario can be described by an automaton with five states and seven transitions; the corresponding constraints on the inputs of the radar can be encoded in Lutin as a node `input_constraints` with four outputs (the three inputs of the radar plus the state `s` of the automaton). Although simple, the scenario covers the apparition and movement of obstacles, and the case where the perception grid should remain unchanged.

```
node input_constraints () returns (car, leo, lilly, s: int) =
  let not_visible: int = 2000 in
  (* initial state: car on street 9 and no visible obstacles *)
  car = 9 and lilly = not_visible and leo = not_visible and s = 0 fby
  loop {
    | (* s = 0 -> car on street 9, lilly on street 5, leo on street 14, s = 1 *)
      (pre s = 0) and car = 9 and lilly = 5 and leo = 14 and s = 1
    | (* s = 0 -> car on street 12, lilly on street 5, leo on street 14, s = 2 *)
      (pre s = 0) and car = 12 and lilly = 5 and leo = 14 and s = 2
    | (* s = 1 -> car on street 9, leo on street 11, lilly on street 12, s = 3 *)
      (pre s = 1) and car = 9 and lilly = 12 and leo = 11 and s = 3
    | (* s = 1 -> car on street 12, leo on street 11, lilly on street 5, s = 4 *)
      (pre s = 1) and car = 12 and lilly = 5 and leo = 11 and s = 4
    | (* s = 2 -> car on street 12, leo on street 11, lilly on street 5, s = 4 *)
      (pre s = 2) and car = 12 and lilly = 5 and leo = 11 and s = 4
    | (* s = 3 -> car on street 12, leo on street 14, lilly on street 18, s = 1 *)
      (pre s = 3) and car = 9 and lilly = 5 and leo = 14 and s = 1
    | (* s = 4 -> car on street 12, leo on street 14, lilly on street 5, s = 2 *)
      (pre s = 4) and car = 12 and lilly = 5 and leo = 14 and s = 2
  }
```

At each iteration, Lurette generates inputs by executing one transition of the Lutin node. Note that if the input values are not explicitly constrained in the Lutin specification, random numbers will be generated. Although larger and more complex scenarios can be written manually, this task is tedious and error-prone, in particular due to the representation of street names by natural numbers (the input language of Lutin supports only Boolean and numerical types), and may easily introduce redundant definitions or equivalent states.

The role of oracles is to determine whether the generated outputs are correct or not. An oracle should contain all possible pairs of inputs with the corresponding expected outputs, and signal an error for each unexpected output.

In addition to checking correctness, Lurette supports additional Boolean outputs (called coverage variables) to measure coverage. While testing the SUT, Lurette records the coverage variables that were at least once true, and com-

putes the ratio of covered versus uncovered variables. This information is stored in a file, and updated by any subsequent run of Lurette for the same SUT, input constraints, and oracle.

A small example of an oracle for the previous scenario is given by the following Lustre node `oracle`, describing the expected output (perception grid) for each given input vector (the positions of the car and obstacles). As the Lutin node, the oracle takes, besides the inputs and outputs of the radar, as input also the state `s` to keep track of the evolution (according to the same small automaton mentionned previously). The oracle outputs the verdict `res`, i.e., whether the observed outputs are those expected for the state and the inputs. For instance, in state 3, `lilly` (in street 12) and `leo` (in street 11) should both be detected by the car (in street 9), whereas they should not be detected in the initial state. The oracle also computes two coverage variables `pass` and `blocked`: the former measures the coverage of state 2 (representing the situation where the car arrives at destination and all obstacles appeared), and the latter measures the coverage of the situation where the car is blocked by the obstacles.

```
const invisible = 2000, already_sent = 3000;
node oracle (s, car, lilly, leo, perception_leo, perception_lilly: int)
returns (res, pass, blocked: bool);
let res = true ->
  ((* lilly and leo are visible from the street 9 *)
    (s = 0 and car = 9 and lilly = invisible and leo = invisible and
     perception_lilly = invisible and perception_leo = invisible)
   or (* the perception did not change, it is already sent *)
    (s = 1 and car = 9 and lilly = 5 and leo = 14 and
     perception_lilly = already_sent and perception_leo = already_sent)
   or (* leo and lilly are visible from the street 9 *)
    (s = 3 and car = 9 and lilly = 12 and leo = 11 and
     perception_lilly = 12 and perception_leo = 11)
   or ... );
   (* true if the car reached the destination (state 2) *)
   pass = false -> if s = 2 then true else pre pass;
   (* true, if the car is blocked by the obstacles *)
   blocked = false -> if s = 3 then true else pre pass;
tel
```

Even more than for the Lutin constraints, manually deriving the oracle is complicated, mainly due to the dependency on the map and the necessity to enumerate all possible movements. For instance, to detect the `already_sent`, one should manually follow the scenario's evolution. Because one can easily forget some cases, an automated generation of these input constraints and the oracle is more convenient, as we illustrate in the next section.

5 Test Projection and Exploration

Each synchronous component of a GALS system is constrained by the other synchronous components, communication media, and environments present in

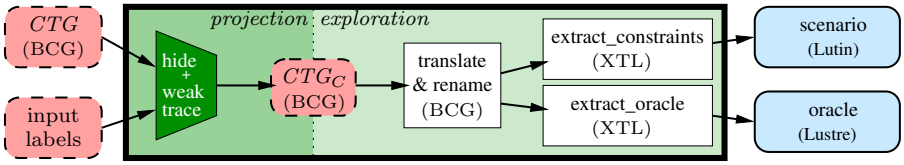

Fig. 2. Overview of the derivation of synchronous test scenarios by projection and exploration of an asynchronous complete test graph CTG

the system. In this section we explicitly exploit these constraints to improve the unit testing of a synchronous component taken separately.

The idea is to automatically derive inputs for synchronous testing tools by projecting the complete test graph generated for the entire GALS system on the inputs and outputs of the synchronous component. In this way, the inputs provided to the synchronous component are realistic and relevant, because they are chosen according to possible execution scenarios of the overall GALS system. Furthermore, the synchronous tests generated in this way contain all possible inputs leading to the goals of test purposes (e.g., a possible input leading to collision in the case of $T2$).

Figure 2 gives an overview of the approach. In a first step, a conformance complete test graph for the overall GALS system is projected on the synchronous component C to be tested, resulting in a test graph for C. In a second step, this test graph is translated and renamed to be compatible with the synchronous testing tool Lurette. In the last step, the test graph is explored (using XTL [28] scripts), generating the input constraints and the oracle for testing C separately. In the remainder of this section, we present the approach in more detail, illustrating how it improves the testing of the radar of our autonomous car example.

5.1 Test Graph Projection

Projecting a complete test graph CTG on a synchronous component C consists in hiding all transitions labeled with an action that is neither an input nor an output of C, and reducing the resulting graph for weak trace equivalence, yielding the projected test graph CTG_C. The reduction removes all internal transitions (created by the hiding), so that all actions of CTG_C are either an input or an output of C. A precondition for a successful projection is that all inputs and outputs of C, as well as the verdict transitions, are present and visible in CTG. Notice that projecting onto the interface of C enables synchronous interaction between the tester and an SUT of C, avoiding all issues related to the asynchronous communication [32,13] present in the GALS model.

In the example of the autonomous car, the radar takes as input the positions of the car and of the obstacles. The positions of the obstacles are also a controllable input of the overall GALS system (see Section 3). But the position of the car is computed by the scenario depending on the output of another synchronous component, namely the action controller. The output of the radar is the

Table 1. Sizes and run-time performance for the tests generated for the test purposes

	TP		CTG		CTG_{RADAR}		constraints	oracle	time	mem.
	states	trans.	states	trans.	states	trans.	(Lutin)	(Lustre)	(s)	(MB)
$T1$	5	4	15,466	29,665	89	281	286	295	29	200
$T2$	4	3	102,985	211,455	584	3617	3622	1916	1237	231
$T3$	5	4	15,444	29,957	83	258	263	282	26	200
$T4$	5	4	2,278	4,959	218	1119	1124	557	86	193
$T5$	5	5	21,930	42,788	105	356	361	344	36	201

perception grid, which is, inside the GALS system, sent to the action controller. Hiding, in the complete test graph CTG, all transitions but those corresponding to these inputs and outputs, and reducing the result with respect to weak trace equivalence yields the LTS $\texttt{CTG}_{\texttt{RADAR}}$. Columns 6 & 7 of Table 1 give the number of states and transitions of $\texttt{CTG}_{\texttt{RADAR}}$ for the five test purposes considered.

5.2 Translating and Renaming

Because some of the data types used in the GALS model might not be exactly the same as those supported by the synchronous testing tools, a preliminary step is the conversion of the values present on the transition labels of CTG_C, for instance by applying appropriate renaming rules. Null values have been added, in order to transform non scalar data on transitions into variable names and values tuples of constant length.

We defined a generic format for the exploration tools to work properly. Each input transition is renamed into "INPUT $!s_1$ $!v_1$ ... $!s_m$ $!v_m$", where s_i is the name of the input and v_i its value; each output transition is similarly renamed into "OUTPUT $!s_1$ $!v_1$ ... $!s_n$ $!v_n$". For instance, the projected complete test graph $\texttt{CTG}_{\texttt{RADAR}}$ contains non-scalar data structures, in particular, the perception grid computed by the radar is represented as a list and streets are identified by their names (i.e., character strings). To be usable with the synchronous test generator Lurette [22], these lists need to be transformed into tuples of constant length, and the street names need to be translated into the corresponding (numeric) constants.

5.3 Test Graph Exploration

By construction, the projected CTG_C is an LTS describing the interaction with the SUT, and as such contains both, the sequence of inputs for the SUT and the verdict concerning the test outcome (i.e., the test oracle). Because the test inputs and oracle should be provided to Lurette using two different languages (Lutin for the input constraints and Lustre for the oracle), two explorations of (the renamed) CTG_C are required.

The input constraints are generated by encoding CTG_C as a possibly non-deterministic node in Lutin with the XTL [28] script `extract_constraints` (87 lines). This node has the same inputs as C and an additional input variable

`s` corresponding to the current state of CTG_C, initialized to the initial state. The main loop of the node contains a nondeterministic choice, with a branch for each transition in CTG_C. A branch corresponding to a transition T is executed if `s` is equal to the source state of T, and as result of execution it sets `s` to the target state of T. A branch for an output transition specifies that the inputs are kept unchanged, which corresponds to the behavior expected for the special output transition δ on quiescent states. A branch for an input transition updates the corresponding inputs. A branch for a verdict transition (i.e., a `pass` or `inconclusive` self-loop on a state of CTG_C), resets the variable `s` to the initial state of the CTG, thus avoiding to generate the same inputs (as a self-loop would) and covering faster the whole CTG. Thus, the Lutin node describes exactly the set of input sequences contained in CTG_C.

An input of the radar is a new position of the car and the obstacles (lilly and leo). Because the exploration takes into account all transitions of $\texttt{CTG}_{\texttt{RADAR}}$, the generated Lutin node incorporates all evolutions of the car and the obstacles that are relevant for the considered test purpose. The generated Lutin nodes are quite long (see Table 1) and complex, because they contain nondeterministic choices, induced by the random movements of the obstacles. Writing similar Lutin nodes by hand would probably have been difficult and error-prone.

Because a synchronous block C is executed atomically, i.e., not interleaved with other synchronous components, a sequence of input transitions for C is immediately followed by the expected sequence of output transitions. Because the target state s of the last transition in such a sequence of inputs is also the source state of the first transition in the sequence of outputs, we call such a state s a *corner state*. For the radar, the sequence of input transitions corresponding to new positions of the car or the obstacles is followed by an output transition corresponding to the expected perception grid. The oracle is generated with the XTL script `extract_oracle` (263 lines), by encoding CTG_C as a deterministic node in Lustre, which, to each corner state and its set of inputs/outputs, associates a Boolean verdict, indicating whether the outputs are the expected ones. This node is also defined for the verdict states of CTG_C. For a *pass* (respectively, *fail*) verdict it always returns true (respectively, false). For an *inconclusive* verdict it returns true iff the outputs are unchanged. In order to observe the coverage of the generated tests, for each verdict state (`pass` and `inconclusive`) and each other state of the CTG, a coverage variable is introduced in the Lustre node generated for the oracle. These coverage variables are true if at least one of the executions passed by the corresponding state of the CTG (see Section 4).

```
node oracle (s, car, lilly, leo, perception_leo, perception_lilly: int)
returns (res, pass, inconclusive, s0, ... s86: bool);
let
  res = true -> (
    if s = 6 then
    (car = 11 and lilly = 13 and leo = 6 and
     perception_lilly = 13 and perception_leo = 2000)
      ... );
  pass = false -> if s = 83 then true else pre pass;
```

```
inconclusive = false -> if s = 7 then true else pre inconclusive;
s0 = false -> if s = 0 then true else pre s0; ...
s86 = false -> if s = 86 then true else pre s86; ...
```

The generated oracles are quite long (see Table 1) and complex, because they contain all oracle verdicts and coverage criteria. For each of the five test purposes, we executed the generated scenarios on the radar SUT using Lurette (the last columns of Table 1 indicate the time and memory consumed by the overall testing process). This enabled us to discover (and fix) a mistake in the SUT related to the incorrect management of the special value `already_sent`.

6 Related Work

Similar to our approach, test purposes for sub-systems can be obtained by projecting symbolic executions of the overall system [7]. A major difference of our approach is that we handle GALS systems (rather than systems homogeneously modeled by IOSTSs), requiring to integrate various formal methods and tools.

The differences between testing synchronous and asynchronous systems in terms of test-suite length, abstraction level, mutation score, non-determinism, and suitability for real-time systems are discussed in [26, Chap. 12]. This analysis enabled to improve the overall testing of a GALS system by taking advantage of the synchronous properties of the components (using a two-level approach combining an overall asynchronous automaton with a dedicated synchronous automaton creating the link to the concrete implementation). In our setting, these properties are taken care of directly by the GRL semantics.

Milner's proposal [30] to encode asynchronism in a synchronous process calculus has been used to specify GALS systems using synchronous programming approaches [17,14,31,19]. Here we follow the opposite way, by specifying the global aspects of a GALS system in an asynchronous language, which enables us to take advantage of the existing tools for asynchronous systems and to leverage them to improve the testing of the synchronous components.

Following a bottom-up approach [15], which manually defines contracts for the synchronous components (to be verified locally, for instance using SCADE [2]), the overall GALS system can be verified by translating the network of component contracts and verification properties into Promela (for verification with SPIN [20]) or timed automata (for timing analysis with UPPAAL [1]). Our approach is top-down and completely automatic, deriving test cases for the synchronous components from a model of the overall GALS system.

A graphical tool set [33], based on the specification of the synchronous components as communicating reactive state machines, translates the system and its properties specified as observers into Promela for verification with SPIN. This tool set focuses on the verification of the overall GALS system, whereas we aim at improving the unit testing of the synchronous components.

Encapsulating synchronous components makes the overall GALS system amenable for analysis with asynchronous verification tools. This approach has

been followed for a combination of the synchronous language SAM, the asynchronous language LNT, and the CADP toolbox [12], and a combination of the synchronous language SIGNAL, the asynchronous language Promela, and the model checker SPIN [6]. Compared to these two approaches, we retain a finer modeling of the synchronous components in the overall GALS system and consider the verification of both, the overall GALS system and its synchronous components. Our approach to derive synchronous test scenarios might be adaptable to these language combination techniques.

Focusing on communication media for GALS hardware circuits, the asynchronous connections between synchronous blocks can be encoded into variants of Petri nets dedicated to the analysis of hardware circuits [3]. On the contrary, our approach targets the test of synchronous components of more generic GALS systems, relying on less precise models of the communication signals.

7 Conclusion

We presented an automatic approach integrating both asynchronous and synchronous testing tools to derive complex, but relevant unit test cases for the synchronous components of a GALS system. From a formal model of the system in GRL [25] and a test purpose, the conformance testing tool TESTOR [27] automatically generates a complete test graph [23] capturing the asynchronous behavior of the system relevant to the test purpose. Such a complete test graph is then projected on a synchronous component C and explored using XTL [28] scripts to provide a synchronous test scenario (input constraints in Lutin [34] and an oracle in Lustre [18]) required to test C with the Lurette tool [22]. All these steps have been automated in an SVL [9] script. The approach substantially relieves the burden of handcrafting these test scenarios, because, by construction, the derived scenarios constrain the inputs provided to C to relevant values, covering a test purpose, which might arise during the execution of the GALS system. We illustrated the approach on an autonomous car example.

As future work, we plan to consider the behavioral coverage of GALS systems, which can be achieved by identifying a test suite (ideally as small as possible) covering the whole state space of a GALS system. Such a test suite could be generated by deriving purposes from the action-based, branching-time temporal properties of the model (similar to [8] in the state-based, linear-time setting), by synthesizing purposes according to behavioral coverage criteria [35], or by constructing a complete test suite for a fault domain of the GALS system [5].

Acknowledgements. This work was supported by the Région Auvergne-Rhône-Alpes within the ARC6 programme.

References

1. G. Behrmann, A. David, K. G. Larsen, O. Möller, P. Pettersson, and W. Yi. Uppaal: Present and Future. In: Decision and Control. IEEE (2001)

2. G. Berry. SCADE: Synchronous design and validation of embedded control software. In: Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems, pp. 19–33. Springer (2007)
3. F. P. Burns, D. Sokolov, and A. Yakovlev. A Structured Visual Approach to GALS Modeling and Verification of Communication Circuits. IEEE Trans. on CAD of Integrated Circuits and Systems 36(6), 938–951 (2017)
4. D. M. Chapiro. Globally-Asynchronous Locally-Synchronous Systems. Doctoral thesis, Stanford University, Department of Computer Science (1984)
5. A. da Silva Simão and A. Petrenko. Generating Complete and Finite Test Suite for ioco: Is It Possible? In: H. Schlingloff and A. K. Petrenko (eds.) MBT 2014, EPTCS, vol. 141, pp. 56–70 (2014)
6. F. Doucet, M. Menarini, I. H. Krüger, R. K. Gupta, and J. Talpin. A Verification Approach for GALS Integration of Synchronous Components. ENTCS 146(2), 105–131 (2006)
7. A. Faivre, C. Gaston, and P. Le Gall. Symbolic Model Based Testing for Component Oriented Systems. In: A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp (eds.) Testing of Software and Communicating Systems, LNCS, vol. 4581, pp. 90–106. Springer (2007)
8. Y. Falcone, J.-C. Fernandez, T. Jéron, H. Marchand, and L. Mounier. More testable properties. STTT 14(4), 407–437 (2012)
9. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In: M. Kim, B. Chin, S. Kang, and D. Lee (eds.) FORTE 2001, pp. 377–392. Kluwer (2001)
10. 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)
11. H. Garavel, F. Lang, and W. Serwe. From LOTOS to LNT. In: J.-P. Katoen, R. Langerak, and A. Rensink (eds.) ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday, LNCS, vol. 10500, pp. 3–26. Springer (2017)
12. H. Garavel and D. Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. In: C. Pasareanu (ed.) SPIN 2009, LNCS, vol. 5578, pp. 241–260. Springer (2009)
13. A. Graf-Brill and H. Hermanns. Model-Based Testing for Asynchronous Systems. In: L. Petrucci, C. Seceleanu, and A. Cavalcanti (eds.) FMICS-AVoCS 2017, LNCS, vol. 10471, pp. 66–82. Springer (2017)
14. P. L. Guernic, J. Talpin, and J. L. Lann. POLYCHRONY for system design. Journal of Circuits, Systems, and Computers 12(3), 261–304 (2003)
15. H. Günther, S. Milius, and O. Möller. On the formal verification of systems of synchronous software components. In: SAFECOMP 2012, LNCS, vol. 7612, pp. 291–304. Springer (2012)
16. N. Halbwachs. Synchronous Programming of Reactive Systems. Kluwer (1993)
17. N. Halbwachs and S. Baghdadi. Synchronous Modelling of Asynchronous Systems. In: EMSOFT 2002, LNCS, vol. 2491, pp. 240–251. Springer (2002)
18. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991)
19. N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In: ACSD 2006, pp. 3–14. IEEE (2006)
20. G. J. Holzmann. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)

21. E. Jahier, N. Halbwachs, and P. Raymond. Engineering functional requirements of reactive systems using synchronous languages. 8th IEEE International Symposium on Industrial Embedded Systems, 8:140–149 (2013)
22. E. Jahier, P. Raymond, and P. Baufreton. Case studies with Lurette V2. STTT 8(6), 517–530 (2006)
23. C. Jard and T. Jéron. Tgv: Theory, principles and algorithms – a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. STTT 7(4), 297–315 (2005)
24. F. Jebali. Formal Framework for Modelling and Verifying Globally Asynchronous Locally Synchronous Systems. PhD thesis, Grenoble Alpes University, France, Sept. 2016.
25. F. Jebali, F. Lang, and R. Mateescu. Formal Modelling and Verification of GALS systems using GRL and CADP. FAoC 28(5), 767–804 (2016)
26. F. Lorber. It's about Time — Model-Based Mutation Testing for Synchronous and Asynchronous Timed Systems. Phd thesis, Institute of Software Technology, Graz University of Technology, Austria (2016)
27. L. Marsso, R. Mateescu, and W. Serwe. TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation. In: D. Beyer and M. Huisman (eds.) TACAS 2018, LNCS, vol. 10806, pp. 211–228. Springer (2018)
28. R. Mateescu and H. Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In: T. Margaria (ed.) STTT 1998, pp. 33–42. BRICS (1998)
29. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In: J. Cuellar, T. Maibaum, and K. Sere (eds.) FM 2008, LNCS, vol. 5014, pp. 148–164. Springer (2008)
30. R. Milner. Calculi for Synchrony and Asynchrony. Theoretical Computer Science 25, 267–310 (1983)
31. M. R. Mousavi, P. L. Guernic, J.-P. Talpin, S. K. Shukla, and T. Basten. Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks. In: DATE 2004, pp. 384–389. IEEE (2004)
32. N. Noroozi, R. Khosravi, M. R. Mousavi, and T. A. C. Willemse. Synchrony and asynchrony in conformance testing. Software & Systems Modeling 14(1), 149–172 (2015)
33. S. Ramesh, S. Sonalkar, V. D'Silva, , N. Chandra, and B. Vijayalakshmi. A Toolset for Modelling and Verification of GALS Systems. In: R. Alur and D. A. Peled (eds.) CAV 2004, LNCS, vol. 3114, pp. 506–509. Springer (2004)
34. P. Raymond, Y. Roux, and E. Jahier. Lutin: a language for specifying and executing reactive scenarios. EURASIP Journal on Embedded Systems (2008)
35. R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. IEEE Trans. Software Eng. 18(3), 206–215 (1992)
36. P. Teehan, M. Greenstreet, and G. Lemieux. A Survey and Taxonomy of GALS Design Styles. IEEE Design Test of Computers 24(5), 418–428 (2007)
37. J. Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. Computer networks and ISDN systems 29(1), 49–79 (1996)