Verification of GALS Systems by Combining Synchronous Languages and Process Calculi

Damien Thivolle[1,2] and Hubert Garavel[1]

[1] INRIA Grenoble - Rhône-Alpes
655 avenue de l'Europe
38 330 Montbonnot Saint Martin – France
{Damien.Thivolle,Hubert.Garavel}@inria.fr
[2] Polytechnic University of Bucharest
Splaiul Independentei 313
060042 Bucharest – Romania

Abstract. A Gals (Globally Asynchronous Locally Synchronous) system typically consists of a collection of sequential, deterministic components that execute concurrently and communicate using slow or unreliable channels. This paper proposes a general approach for modelling and verifying Gals systems using a combination of synchronous languages (for the sequential components) and process calculi (for communication channels and asynchronous concurrency). This approach is illustrated with an industrial case-study provided by Airbus: a Tftp/Udp communication protocol between a plane and the ground, which is modelled using the Eclipse/TopCased workbench for model-driven engineering and then analysed formally using the Cadp verification and performance evaluation toolbox.

1 Introduction

In computer hardware, the design of synchronous circuits (i.e., circuits the logic of which is governed by a central clock) has long been the prevalent approach. In the world of software, *synchronous languages* [15] are based on similar concepts. Whatever their concrete syntaxes (textual or graphical) and their programming styles (data flow or automata-based), these languages share a common paradigm: a synchronous program consists of components that evolve by discrete steps, and there is a central clock ensuring that all components evolve simultaneously. Each component is usually deterministic, as well as the composition of all components; this assumption greatly simplifies the simulation, testing and verification of synchronous systems.

During the two last decades, synchronous languages have gained industrial acceptance and are being used for programming critical embedded real-time systems, such as avionics, nuclear, and transportation systems. They have also found applications in circuit design. Examples of synchronous languages are Argos [22], Esterel [3], Lustre/Scade [14], and Signal/Sildex [1].

However, embedded systems do not always satisfy the assumptions underlying the semantics of synchronous languages. Recent approaches in embedded

systems (modular avionics, X-by-wire, etc.) introduce a growing amount of asynchronism and nondeterminism. This situation has been known for long in the world of hardware, where the term GALS (*Globally Asynchronous, Locally Synchronous*) was coined to characterize circuits consisting of a set of components, each governed by its own local clock, that evolve asynchronously. Clearly, these evolutions challenge the established positions of synchronous languages in industry.

There have been several attempts at pushing the limits of synchronous languages to model GALS systems. Following Milner's result [25] that asynchronism can be encoded in a synchronous process calculus, there have been approaches [16, 21, 26, 17] suggesting ways to describe GALS systems using synchronous languages; for instance, nondeterminism is expressed by adding auxiliary input variables (*oracles*), the value of which is undefined; a main limitation of these approaches is that asynchronism and nondeterminism are not recognized as first-class concepts, so verification tools often lack optimizations specific to asynchronous concurrency (e.g. partial orders, compositional minimization, etc.). Other approaches extend synchronous languages to allow a certain degree of asynchrony, as in CRP [2], CRSM [28], or *multiclock* ESTEREL [4], but, to our knowledge, such extensions are not (yet) used in industry. Finally, we can mention approaches [13, 27] in which synchronous programs are compiled and distributed automatically over a set of processors running asynchronously; although these approaches allow the generation of GALS implementations, they do not address the issue of modelling and verifying GALS systems.

A totally different approach would be to forget about synchronous languages and adopt languages specifically designed to model asynchrony and nondeterminism, and equipped with powerful verification tools, namely process calculi such as CSP [6], LOTOS [19], or PROMELA [18]. Such a radical migration, however, would not be so easy for companies that invested massively in synchronous languages and whose products have very long life-cycles calling for stability in programming languages and development environments.

In this paper, we propose an intermediate approach that combines synchronous languages and process calculi for modelling, verifying, and evaluating the performance of GALS systems. Our approach tries to retain the best of both worlds:

- We continue using synchronous languages to specify the components of GALS systems, and possibly sets of components running together in synchronous parallelism.
- We introduce process calculi to: (1) encapsulate those synchronous components or sets of components; (2) model additional components whose behavior is nondeterministic, a typical example being unreliable communication channels that can lose, duplicate and/or reorder messages; (3) interconnect all parts of a GALS systems that execute together according to asynchronous concurrency. The resulting specification is asynchronous and can be analysed using the tools available for the process calculus being considered.

As regards related work, we can mention [29], which translates CRSM [28] into PROMELA and then uses the SPIN model checker to verify properties expressed

as a set of distributed observers; our approach is different in the sense that it can use synchronous languages just as they are, instead of introducing a new synchronous/asynchronous language such as Crsm.

Closer to our approach is [9], which uses the Signal compiler to generate C code from synchronous components written in Signal, embeds this C code into Promela processes, abstracts hardware communication buses as Promela finite Fifo channels, and finally uses Spin to verify temporal logic formulas. A key difference between their approach and ours relies in the way locally synchronous components are integrated into a globally asynchronous system. Their approach is *stateful* in the sense that the C code generated for a synchronous Signal component is a transition system with an internal state that does not appear at the Promela level; thus, they must rely upon the "**atomic**" statement of Promela to enforce the synchronous paradigm by merging each pair of input and output events into one single event. To the contrary, our approach is *stateless* in the sense that each synchronous component is translated into a Mealy function without internal state; this allows a smoother integration within any asynchronous process calculi that has types and functions, even if it does not possess an "**atomic**" statement — which is the case of most process calculi.

We illustrate our approach with an industrial case study provided by Airbus in the context of the TopCased[3] project: a ground-plane communication protocol consisting of two Tftp (*Trivial File Transfer Protocol*) entities that execute asynchronously and communicate using unreliable Udp (*User Datagram Protocol*) channels. For the synchronous language, we will consider Sam [8], a simple synchronous language (similar to Argos [22]) that was designed by Airbus and that is being used within this company. Software tools for Sam are available within the TopCased open-source platform based on Eclipse. For the process calculus, we will consider Lotos NT [7], a simplified version of the international standard E-Lotos [20]. A translator exists that transforms Lotos NT specifications into Lotos specifications, thus enabling the use of the Cadp toolbox [12] to perform verification and performance evaluation on the generated Lotos specifications.

This paper is organized as follows. Section 2 presents the main ideas of our approach for analysing systems combining synchrony and asynchrony. Section 3 introduces the Tftp industrial case study. Section 4 gives insights into the formal modelling of Tftp using our approach. Section 5 reports on state space exploration and model checking verification of Tftp models. Section 6 addresses performance evaluation of Tftp models by means of simulation. Finally, Section 7 gives concluding remarks and discusses future work.

2 Proposed methodology

A synchronous program performs a sequence of steps. At each step, it receives *inputs* from the environment, computes a reaction, and sends outputs to the environment. It maintains its own internal state.

[3] `www.topcased.org`

A synchronous program may be a synchronous composition of several synchronous components. All these components react exactly in one step and may communicate with each other (the output of one serving as input of another one).

2.1 Modelling synchronous components as Mealy functions

A synchronous component has an internal state s. At each step, it receives a set of m input values $i_1, \ldots, i_m$ and computes (in zero time) a set of n output values $o_1, \ldots, o_n$ as well as its new state s'. That is to say, it is a function of the form:

$$(s', o_1 \ldots o_n) = f(s, i_1 \ldots i_m)$$

This function corresponds to a (usually deterministic) Mealy machine [24] i.e., a 5-tuple ($\mathcal{S}$, s_0, $\mathcal{I}$, $\mathcal{O}$, $\mathcal{T}$) where:

- $\mathcal{S}$ is a finite set of states,
- s_0 is the initial state,
- $\mathcal{I}$ is a finite input alphabet,
- $\mathcal{O}$ is a finite output alphabet,
- $\mathcal{T}$ is a transition function $\mathcal{S} \times \mathcal{I} \to \mathcal{S} \times \mathcal{O}$ mapping the current state and the input alphabet to the next state and the output alphabet.

The function f, corresponding to a synchronous component, can either be generated directly in C using the compiler of the synchronous language (for this purpose, there exists a common object code format for synchronous languages named OC) or be programmed directly in a process calculus as a user-defined function (we will follow this second approach).

The same applies to a synchronous composition of synchronous components because this composition can also be modelled by a Mealy machine. This is a property of synchronous parallelism.

2.2 The SAM language

To illustrate our approach, we consider the case of the synchronous language SAM designed by Airbus. A synchronous component in SAM is an automaton has a set of input and output ports, each port corresponding to a boolean variable.

A SAM component is very similar to a Mealy machine. The main difference lies in the fact that a transition in SAM is a quintuple (s_1, s_2, F, G, P), where s_1 and s_2 is the source state of the transition, s_2, the destination state of the transition, where F is a boolean condition on the input variables, where G is a set of output variables, and where P is a priority integer value. where P is a priority index. If a set of input values enables more than one outgoing transition from the current state, the transition with the smallest priority index is chosen. For this reason, the priority indexes from transitions going out of the same state must be pairwise distinct. The other difference is that every state has an implicit outgoing transition leading to itself that is performed if no other transition can be perfomed; it sets all the output ports to `false`.

Fig. 1 gives an example of a SAM automaton. An interrogation mark precedes the condition F of each transition. An exclamation mark precedes the list G of the output variables that are set to `true` when the transition is performed. If an output variable is absent from that list, it is set to `false` when the transition is performed. Priority indexes are attached to the base of the transitions.

Priority indexes are notational conveniences that can be eliminated as follows: each transition (s_1, s_2, F, G, P) must be replaced by (s_1, s_2, F', G, P) where $F' = F \wedge \neg(F_1 \vee \ldots \vee F_n)$ such that $F_1, \ldots, F_n$ are the conditions attached to the outgoing transitions of state s_1 with a priority index strictly lower than P.

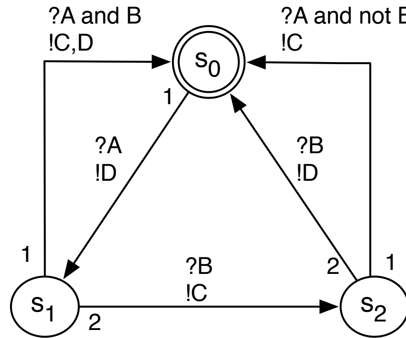

Fig. 1. Example automaton in SAM

This SAM automaton can be encoded in LOTOS NT as follows:

```
type State is
  S0, S1, S2
end type
function transition (in CurrentState:State, in A:Bool, in B:Bool
               out NextState:State, out C:Bool, out D:Bool) is
  case CurrentState in
  S0 ->
    if A then
      NextState := S1; C := false; D := true
    else
      NextState := CurrentState; C := false ; D := false
    end if
  | S1 ->
    if A and B then
      NextState := S0; C := true; D := true
    elsif B then
      NextState := S2; C := true ; D := false
    else
      NextState := CurrentState; C := false; D := false
    endif
  | S2 ->
    if A and not (B) then
      NextState := S2; C := true ; D := false
    elsif B then
      NextState := S0; C := false ; D := true
```

```
      else
        NextState := CurrentState; C := false ; D := false
      end if
    end case
  end function
```

Sam supports the synchronous composition of automata. A global system in Sam has input and output ports. It is composed of one or several Sam automata. Communication between automata is expressed by drawing connections between input and output ports, with the following rules:

- inputs of the system can connect to outputs of the system or inputs of automata;
- outputs of automata can connect to inputs of other automata or outputs of the system;
- cyclic dependencies are forbidden.

Because of the last rule, one can find a topological order for the dependencies between automata. Thus, a Sam system can be encoded in Lotos NT as a sequential composition of the Mealy functions of its individual Sam automata.

2.3 Wrapping Mealy functions into input/output processes

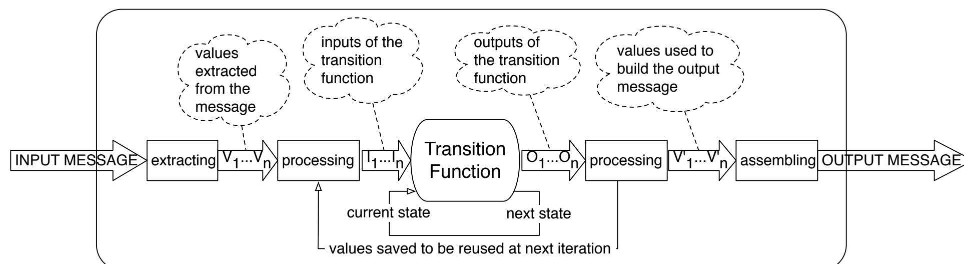

Fig. 2. The wrapper process in detail

In contrast with synchronous programs, components of asynchronous programs run concurrently, at their own pace, and synchronize with each other through communications on gates or channels.

Our approach to modelling Gals systems in asynchronous languages builds on encoding synchronous programs as a set of native types and functions in a given process calculus.

This transition (or Mealy) function alone cannot interact with an asynchronous environment. It needs to be wrapped into a process that handles the communication with the environment. This wrapper transforms the Mealy function of a synchronous component into an Lts (*Labelled Transition System*). The function of this process is illustrated by Fig. 2. Every time it receives a message, it extracts values from it. These values can be directly mapped to the inputs of the transition function or may be processed to produce these inputs. Then it constructs a message from the outputs returned by the transition function (or

values processed from them) and makes the message available for sending. The message is not actually sent right away; it is sent when the recipient process is ready for synchronization. The process and the function, together act as a reactive process. They react to a stimulation, the reception of a message, to produce an answer, the output message.

The amount of processing a wrapper can do is arbitrary. It depends on the GALS system being modeled. It can be very little, in which case the message received could be an aggregation of the inputs of the transition function. It can be significant: extraction of the inputs (of the transition function) from a complex message, construction of a complex message from the ouputs, storing outputs or values derived from them to be reused for the next stimulation.

2.4 Composing processes with asynchronous parallelism

Parallel composition of the wrapper and the environment with which we make it interact can be achieved using the parallel operator of LOTOS NT. This operator defines on the channels (or gates) on which the wrapper and the environment will communicate. The environment could be indifferently written in LOTOS NT or made of other synchronous programs wrapped into LOTOS NT processes and communicating asynchronously.

3 The TFTP case study

This case study was provided to us by Airbus. We first recall the principle of the standard TFTP protocol then we present the custom adaptation made at Airbus for plane/ground communications.

3.1 The standard TFTP protocol

TFTP [30] stands for *Trivial File Transfer Protocol*. It is a client/server protocol in which the client can request to send (resp. receive) a file to (resp. from) server. As it is designed to run over the UDP (*User Datagram Protocol*) protocol, the TFTP protocol implements its own flow control mechanism.

In a typical case, a client initiates a transfer by sending a request to the server, `RRQ` (*Read ReQuest*) for reading a file or `WRQ` (*Write ReQuest*) for writing (i.e. sending) a file. The files are truncated into fragments of equal size, which are transferred sequentially. The server replies to a read request by sending the first data fragment of the requested file. A read request is answered by the first data fragment (`DATA`) of the requested file and a write request is answered by an acknowledgement (`ACK`). In addition to the data it carries, a data fragment also contains an index value which is used to make sure that all the data fragments are received consecutively. The last data fragment is characterized by a size smaller than that of the other fragments. An acknowledgement carries the index value of the data fragment it acknowledges. An acknowledgement numbered 0 answers a write request.

The protocol is designed to be robust. Any lost message can be retransmitted after a timeout. Duplicate (resent because of a timeout) acknowledgements are not replied to in order to avoid the Sorcerer's Apprentice bug [5].

If an error (memory shortage, fatal error, etc.) occurs in one of the hosts, it sends an error message (`ERROR`) to the other in order to abort the transfer.

A transfer ends when the acknowledgement of the last data fragment is received. The host that sent this acknowledgement is encouraged to wait for a while in order to resend the acknowledgement should the final data fragment be resent by the other host. This is called *dallying*.

In order for the server to differentiate between clients, each incoming request is served on a different UDP port.

3.2 The Airbus custom TFTP implementation

For the purpose of ground/plane communication, Airbus is experimenting with a simplified version of the TFTP protocol. In the future protocol stack used by Airbus, this simplified version of the TFTP protocol runs above the UDP layer and below a layer of protocols dedicated to communications in avionics such as ARINC 615. The TFTP protocol will carry the frames of these dedicated protocols. For that reason, every host will have the ability to be both a client and a server, depending on what that upper layer dictates. A static number of TFTP hosts is instantiated. Whenever a plane docks, a TFTP host is assigned to serve it. It means that at any given time a TFTP host (either from the plane or in the airport) will only communicate with a single other TFTP host. This removes the need for modelling the fact that a server can serve many different clients on as many different UDP ports.

4 Modelling the TFTP architecture

Airbus was interested in verifying that these TFTP hosts would behave correctly in a realistic environment in which messages sent from one host to the other could be lost and reordered. For this purpose, we model a specification involving two TFTP protocol entities connected by two media. As shown in Fig. 3, the TFTP protocol entities are two instances of the same LOTOS NT process whose behaviour is dictated by the Mealy function of the SAM TFTP automaton while the media are two instances of the same LOTOS NT process that models the behaviour of the UDP protocol running over a wired network. We chose LOTOS NT rather than LOTOS because LOTOS NT functions are easier to use than LOTOS equations for representing the Mealy function of the SAM TFTP automaton and for manipulating data in general.

4.1 Modelling the TFTP protocol entities

The behaviour of Airbus TFTP hosts is encoded as a SAM system consisting of one SAM automaton. The automaton has 7 states, 39 transitions, 15 inputs and 11 outputs. We translated this SAM automaton manually into 215 lines of

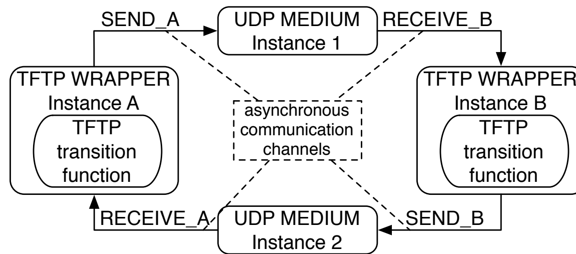

Fig. 3. Asynchronous connection of two TFTP processes via unreliable media

LOTOS NT(precisely, an enumerated type to encode the states and the Mealy function of the automaton).

The Mealy function representing the behaviour of the TFTP protocol entity must be encapsulated into a wrapper to communicate with the UDP media. We defined two different LOTOS NT wrapper processes: the "simplified TFTP process" which was modelled according to Airbus recommendations, and the "accurate TFTP process" which is closer to the standard TFTP protocol.

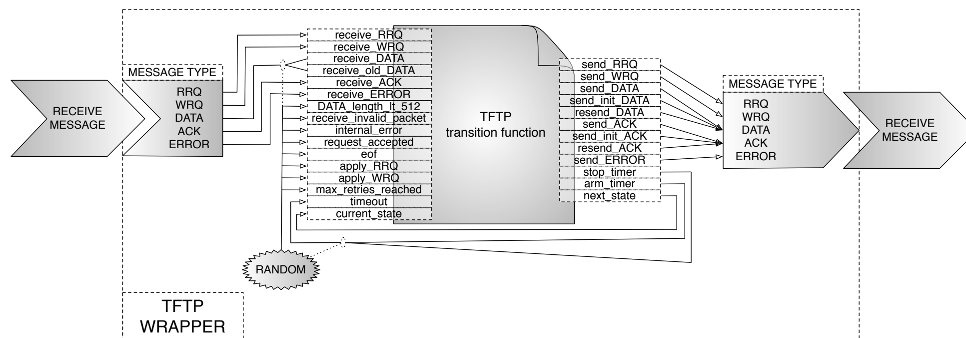

Fig. 4. Simplified TFTP Process

The simplified TFTP process consists of a simple wrapper around the Mealy function and does no processing on its own. The messages it receives are an aggregation of the inputs of the Mealy function; it unmarshals them and passes them on to the Mealy function, then it retrieves the outputs, marshals them and sends the message thus obtained. It is as if the outputs of one Mealy function in one TFTP protocol entity connect directly to the corresponding inputs of the Mealy function in the other TFTP protocol entity. The files under transfer are not modelled. A host can randomly send a read or write request (provided that there is no ongoing transfer) and the transfer will continue until the sender arbitrarily decides to send the last data fragment. The simplified TFTP process and the types and functions that we had to define for the messages were written in 260 lines of LOTOS NT.

The simplicity of that design helped us writing properties for the verification phase but limited us at the same time. For example, the TFTP automaton has two outputs named `send_DATA` and `resend_DATA` and two inputs `receive_DATA` and `receive_old_DATA`. Connecting respectively `send_DATA` and `resend_DATA` to `receive_DATA` and `receive_old_DATA` might seem reasonable when considering a perfect medium but prevents us from testing error cases in our design with a lossy medium. Indeed, if a data fragment is sent from one host (`send_DATA`) and lost by the medium, it will be resent (`resend_DATA`) after a timeout. If that second message is received, its recipient should see it as a `receive_DATA` whereas in our simplified design the `receive_old_DATA` input of the Mealy function will be erroneously set to true.

In order to suppress such limitations we improved our modelling and created an accurate TFTP process that receives and sends real TFTP messages (as defined in the standard TFTP protocol) and processes those messages in order to compute the proper values for the inputs of the Mealy function. In this accurate TFTP process, read and write requests carry the name of the requested file. This file name is modelled as an integer. Both processes have a repository in which an integer corresponds to a file. Files are modelled as sequences of characters, each character being one fragment of the file. Data fragments carry three values, an integer for the index value of the fragment, a character for the file fragment and a boolean value indicating whether this is the last fragment of the file. Acknowledgements carry the index value of the data fragment they acknowledge. In order to fight state explosion in the latter phases we instantiate the processes with a list of names of files to read ("read list") and a list of names of files to write ("write list") instead of just letting the compiler generate every possible file. Whenever there is no active transfer a process can randomly choose to send a read request for the first file name in its read list or a write request for the first file name in its write list. Besides the state of the automaton, additional values must be kept in memory between two subsequent calls to the Mealy function: the name of the file being transferred and the index value of the last data fragment or acknowledgement received or sent, whether the last data fragment received is the last one, etc. The accurate TFTP process and the types and functions that we defined for the files, repositories and messages were written in 671 lines of LOTOS NT.

4.2 Modelling the UDP media

The medium LOTOS NT process must reproduce the behaviour of the link between the plane and the ground: UDP protocol over an Ethernet network. UDP is a connection-less protocol, so as regards our modelling it means that a process writing on the medium will never block. UDP does not have any error recovery mechanism so a problem that occured and was not corrected by the lower networking layers will be propagated to any application using UDP. These errors are message losses, message reordering, and message duplications. Message losses can always happen. Modern routers use load-balancing software to send all packets of the same stream through the same route. This reduces message reordering to a certain extent but does not guarantee it will not happen. Message

duplications would have to come from bogus implementations of the lower level networking layers so we discard the possibility that they can happen.

We chose to model the medium in two different ways, using two different LOTOS NT processes. Both processes allow messages to be lost and have a buffer of fixed size in which the messages are stored upon reception, waiting for delivery. The first process models the case where message reordering does not happen. It uses a FIFO as a buffer: messages are delivered in the same order they were received. The second process models the case where message reordering can happen. It uses a bag as a buffer: it does not guarantee that messages are delivered in the same order they were received. We denote FIFO (n) (resp. BAG (n)) a medium with a FIFO (resp. bag) buffer of size n. The LOTOS NT processes for the FIFO medium and the bag medium are respectively 24 and 27 lines long.

4.3 Asynchonous composition of TFTP entities and UDP media

To compose the TFTP protocol entities and the UDP media asynchronously, we use the parallel operator of LOTOS NT:

```
par
  RECEIVE_A, SEND_A -> TFTP_WRAPPER [RECEIVE_A, SEND_A]
||
  RECEIVE_B, SEND_B -> TFTP_WRAPPER [RECEIVE_B, SEND_B]
||
  SEND_A, RECEIVE_B -> UDP_MEDIUM [SEND_A, RECEIVE_B]
||
  SEND_B, RECEIVE_A -> UDP_MEDIUM [SEND_B, RECEIVE_A]
end par
```

As we have two different TFTP processes and two different medium processes we obtain four specifications: simplified TFTP specification with bag media, simplified TFTP specification with FIFO media, accurate TFTP specification with bag media, and accurate TFTP specification with FIFO media.

5 Verifying functional correctness

In this section, we detail how we generate the state space for the specifications and how we define correctness properties to ensure the proper behaviour of these specifications. Then, we explain how we perform model checking using CADP.

5.1 State space generation

An LTS (*Labelled Transition System*) representation of the specifications is produced by CADP compilers. LOTOS NT specifications are automatically translated into LOTOS specifications, which are, in turn, compiled into an LTS by the CÆSAR [10] compiler.

One important issue in model checking is the problem of the state explosion. For example if we try to compile the simplified TFTP specification with BAG (2) media, it takes on a machine equipped with a dual-core processor running

at 3.17Ghz and 3 Gb of RAM, 8 hours and 25 minutes to obtain an LTS with 27,721,727 states and 216,183,185 transitions.

To fight this, we use compositional generation. It is a *divide and conquer* approach, available in CADP which consists in producing the LTS for each of the processes involved in the specification, minimizing them using the BCG_MIN tool of CADP and composing them together.

With this method, the aforementioned specification can be generated in 10 minutes and 34 seconds yielding an LTS of 6,552,898 states and 35,762,508 transitions.

Tab. 1 gives an insight into the influence of the buffer size on the whole specification size. For every additional element in the buffer, the specification increases around tenfold. Writing all those compositions and calling the tools

Medium	Medium Generation			Specification Generation		
	States	Transitions	Time (s)	States	Transitions	Time (s)
BAG (1)	31	150	2.45	66,227	271,710	19.98
BAG (2)	321	1,630	2.67	1,656,577	7,283,171	37.34
BAG (3)	2,366	11,930	3.77	18,923,839	93,350,943	435.07
BAG (4)	11,926	62,370	120.44	– (did not finish)	–	–
FIFO (1)	31	150	2.62	66,227	271,710	19.67
FIFO (2)	321	1,540	2.62	1,137,246	4,776,989	31.19
FIFO (3)	3,221	15,440	3.55	18,337,328	77,600,123	375.10
FIFO (4)	32,221	154,440	59.35	–	–	–

Table 1. Generation times for a client/server scenario with one file exchanged

manually to perform them is a demanding task. For this reason we have a scripting language named SVL in CADP. It wraps the functionnalities of CADP tools in order to let the user write its processings (composition, reduction, verification, bisimulations, etc.) in a readable and orderly fashion.

5.2 Temporal logic properties

Correctness properties must first be expressed in natural language and then translated into temporal logic formulas. For example, we have a property which states that the data fragments must be sent properly ordered. We chose to ensure this by showing that any data fragment numbered x can not be followed by a data fragment numbered y, where $y < x$, unless there has been a reinitialization (transfer succeeded or aborted) in between. EVALUATOR 4.0 uses an extension of the μ-calculus temporal logic that can manipulate data. In this logic, the property can be expressed as follows:

```
[
  true* .
  {SEND_A !"DATA" ?x:Nat ...} .
  not (REINIT_A)* .
  {SEND_A !"DATA" ?y:Nat ...
     where y < x}
] false
```

The formula states that there exists no sequence of transitions in the LTS that leads to TFTP protocol entity 1 sending a data fragment numbered x then a data fragment numbered y, where $y < x$, without the transfer succeeding or aborting (`REINIT_A1`).

We wrote 12 properties to be verified on both the simplified TFTP specifications and the accurate TFTP specifications. Another 17 were written exclusively for the accurate TFTP specifications.

5.3 Model checking the specifications

To verify whether a property holds for a specification, we use the EVALUATOR 4.0 model checker [23]. We feed it one specification represented as an LTS and one formula of temporal logic and it answers whether the property holds by exhaustively exploring the transition system.

It is worth noting that CADP allows one to perform on-the-fly verification, letting the model checker drive the generation of the LTS as it verifies the property. This is particularly useful when the LTS would not fit on the machine hard drive disk. In our case, it is better to first generate the LTS, then verify all the properties, reusing the same LTS for every property. We can do so because the LTSs for our specifications are of reasonable size.

Several of the first batch of 12 properties did not hold on the simplified TFTP specifications. This enabled us to find 11 errors in the TFTP automaton.

Verification of the accurate TFTP specifications requires constraining the files that can be exchanged between the two TFTP protocol entities so as to reduce the size of the LTS that is generated. For that purpose, we instantiate the TFTP protocol entities with lists of names of files to read and write. To cover all the possibilities, four scenarios were required:

1. TFTP protocol entity 1 and 2 both write one file;
2. TFTP protocol entity 1 writes one file and TFTP protocol entity 2 reads one file;
3. TFTP protocol entity 1 and 2 both reads one file;
4. TFTP protocol entity 1 reads one file and TFTP protocol entity 2 writes one file.

We limit the size of the files to 2 characters as it is sufficient to cover all the transitions of the automaton. The verification of the first batch of 12 properties on the accurate TFTP specifications yielded the same results that we had already obtained by verifying them on the simplified TFTP specifications. The verification of the second batch of 17 properties especially written for the accurate TFTP specifications led to the discovery of an additional 8 errors in the TFTP automaton.

We tested the simplified and accurate TFTP specifications using BAG (1), BAG (2), BAG (3), FIFO (1), FIFO (2), and FIFO (3) as media and always obtained the same results.

In total, we found 19 errors on the TFTP case study. 11 errors were found on the simplified TFTP specification and another 8 on the accurate TFTP specification. They were reported to Airbus and were acknowledged as being actual errors

in the TFTP automaton. We also suggested changes in the TFTP automaton to correct them. These errors do not occur in reality as the TFTP implementation embedded in planes is even simpler than the one given to us for study. While some of these errors could be found by a human after a careful study of the automaton, some others are more subtle and would probably be missed by an engineer. For example, if both TFTP entities send a request (`RRQ` or `WRQ`) at the same time, they would ignore each other. This would be hard to detect when looking at a TFTP automaton alone.

6 Performance evaluation by simulation

The model checking verification described in section 6 found errors but without quantitative measurements. The presence of the errors we detected does not prevent the existence of satisfying sequences of transitions leading to successful transfers. It is always possible to finish the transfer and the errors in the TFTP automaton apparently only cause extra timeouts and additional messages to be sent.

To obtain quantitative information about the impact of the errors detected, we turned to simulation as it allowed us to measure the seriousness of the errors. There are several approaches to simulation, network of waiting lines (queuing theory), models based on Markov chains (Interactive Markov Chains) [11] and random simulation by generation of random traces for a given specification. In our case, the last approach was the most adapted because CADP allowed us to reuse the accurate TFTP specification.

6.1 Simulation methodology with CADP

The specification we used for simulation is a slightly modified version of the accurate TFTP specification with bag media. We also built several TFTP automata in addition to the original one that Airbus gave us. First, we have an automaton on which we corrected all the errors we found. Second, for each error, we built an automaton that only exhibits that particular error. The idea is to be able to quantify the loss in performances induced by each error individually and then to quantify the loss caused by the errors altogether.

We made the following modifications to the specification: the file repository is now generated randomly for every simulation and with an arbitrary number of files, the TFTP protocol entities are instantiated with an arbitrary number of files to read and write, the size of the media queue was raised to 6, and the queue type was modified in order to give a higher priority to messages that arrived first.

The EXECUTOR tool, distributed with CADP as a C source code file generates a random execution trace of a specification (instead of the entire state space). As its source code is available, we modified it in order to implement a system of weights for the transitions. For each different automaton, we ran the same number x of simulations in order to obtain x execution traces. For each of these traces, we computed a *time* of execution based on the TFTP messages sent that the trace exhibited. At the end we obtained a mean *time* of execution for each one of the automata and we could compare them.

We defined two scenarios of simulation. The first one consisted of one TFTP protocol entity acting like a server (with empty lists of files to read or write) and the other one acting like a client, with files to read and files to write. This scenario is a realistic model of Airbus deployment of their TFTP hosts. In the second scenario, both protocol entities had lists of files to read and to write. They competed for obtaining the right to start transfering their files. This is a worst-case scenario that is unlikely to happen in Airbus deployment but Airbus engineers recognized it ought to be tested as it can happen under heavy load but is hard to reproduce in reality.

By default, the EXECUTOR tool will randomly choose the next transition if there are several possibilities in the state it arrived to. We modified this behaviour by assigning weights to the transitions, a transition being more likely to be chosen than another if its weight is higher. The idea is to give very low weight values to error transitions such as the internal error or the reception of an invalid packet. We used the following values: 1 for error cases (internal error, request rejected, invalid packet), 100 for a timeout or a loss in the medium and 10000 for any other transition. The values for the error cases are totally arbitrary; we tried to make them realistic but the truth is we cannot predict how reliable the link or the hardware will be.

In order to compute the execution time, we gave our media some characteristics. Their latency time would be 8 ms and their bandwidth would be 1 MB/s. We also considered that the size of the TFTP data packets would be 32 kilobytes. Receiving a read request (`RRQ`), a write request (`WRQ`), an acknowledgement (`ACK`) or an error (`ERROR`) would take 2 ms (a quarter of the latency). Sending a read request, a write request, an acknowledgement or an error would take also 2 ms. Receiving or sending a data fragment would take 18ms (2ms plus half the time required to send 32 kilobytes at 1MB/s).

For each automaton we used, we tried different values of timeout. The idea was to ensure that for any timeout value, the results were consistent.

6.2 Simulation results

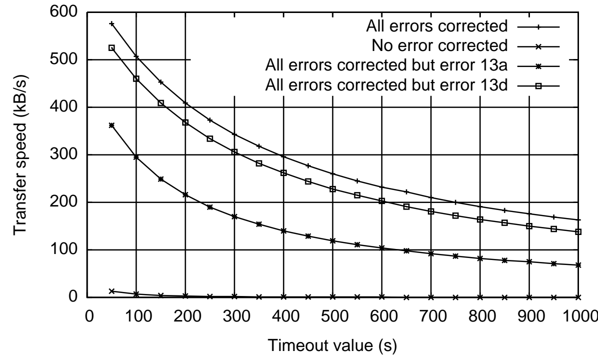

Fig. 5. Simulation results for scenario 2.

Most of the errors that we found would have an effect only if both processes were competing for transferring files. In the scenario 1, only one of the processes was reading or writing files while the other acted as a server. Yet, the difference in performances is around 10%, regardless of the timeout value we used. In a typical usage the corrections we made induced a significant gain in performances.

Fig. 5 displays the results for the scenario 2. It clearly shows that the original TFTP automaton from Airbus can not handle concurrency while the corrected version can handle it without any noticeable loss in speed compared to scenario 1. Out of the 19 errors, only 6 seriously impact the performances and we chose to display two of them on Fig. 5.

7 Conclusion

In this paper, we have proposed a novel approach for modelling and analysing systems consisting of synchronous components interacting asynchronously, commonly referred to as GALS (*Globally Asynchronous Locally Synchronous*) in the hardware design community.

Contrary to other approaches that stretch or extend the synchronous paradigm to model asynchrony, our approach preserves the genuine semantics of synchronous languages as well as the well-known semantics of asynchronous concurrency. It allows to reuse without any modification the existing compilers for synchronous languages together with the existing compilers and verification tools for process calculi.

We have demonstrated the feasibility of our approach on an industrial case study, the TFTP/UDP protocol for which we successfully performed model checking verification and performance evaluation using the TOPCASED and CADP software tools. Although this case study was based on the SAM synchronous language and the LOTOS/LOTOS NT process calculi, we believe that our approach is general enough to be applicable to any synchronous language whose compiler can translate synchronous components (or sets such components) into Mealy machines — which is almost always the case — and to any process calculus equiped with asynchronous concurrency and user-defined functions.

As regards future work, we received strong support from Airbus to apply the proposed approach to more case studies and to generalize it to other languages than SAM.

Acknowledgements

We are grateful to Patrick Farail and Pierre Gaufillet (Airbus) for their continuous support and to Claude Helmstetter (INRIA/Vasy), Pascal Raymond (CNRS/Verimag), and Robert de Simone (INRIA/Aoste) for their insightful comments about this work.

References

1. Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

2. G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *POPL '93*, pages 85–98, New York, NY, USA, 1993. ACM.
3. Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
4. Gérard Berry and Ellen Sentovich. Multiclock Esterel. In *CHARME '01*, pages 110–125, London, UK, 2001. Springer-Verlag.
5. R. Braden. Requirements for Internet Hosts - Application and Support. RFC 1123, Internet Engineering Task Force, October 1989.
6. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.
7. David Campelovier, Xavier Clerc, and Hubert Garavel. Reference Manual of the LOTOS NT to LOTOS Translator, Version 4G. Technical report, INRIA, January 2009.
8. Xavier Clerc, Hubert Garavel, and Damien Thivolle. Préesentation du langage SAM d'Airbus. Technical report, INRIA, 2008. In French.
9. Frederic Doucet, Massimiliano Menarini, Ingolf H. Krüger, Rajesh K. Gupta, and Jean-Pierre Talpin. A Verification Approach for GALS Integration of Synchronous Components. *Electr. Notes Theor. Comput. Sci.*, 146(2):105–131, 2006.
10. Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.
11. Hubert Garavel and Holger Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proceedings of the 11th International Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark)*, volume 2391 of *Lecture Notes in Computer Science*, pages 410–429. Springer Verlag, July 2002. Full version available as INRIA Research Report 4492.
12. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer Verlag, July 2007.
13. Alain Girault and Clément Ménier. Automatic Production of Globally Asynchronous Locally Synchronous Systems. In *EMSOFT '02*, pages 266–281, London, UK, 2002. Springer-Verlag.
14. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
15. Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic, 1993.
16. Nicolas Halbwachs and Siwar Baghdadi. Synchronous Modelling of Asynchronous Systems. In *EMSOFT '02*, pages 240–251, London, UK, 2002. Springer-Verlag.
17. Nicolas Halbwachs and Louis Mandel. Simulation and Verification of Asynchronous Systems by means of a Synchronous Model. In *ACSD '06*, pages 3–14, Washington, DC, USA, 2006. IEEE Computer Society.
18. G.J. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley Professional, 2004.
19. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

20. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.
21. Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers. World Scientific*, 12, 2003.
22. F. Maraninchi and Y. Rémond. Argos: an Automaton-Based Synchronous Language. *Computer Languages*, 27(1–3):61–92, October 2001.
23. Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In Jorge Cuellar and Tom Maibaum, editors, *Proceedings of the 15th International Symposium on Formal Methods FM'08 (Turku, Finland)*, number 5014 in Lecture Notes in Computer Science, pages 148–164. Springer Verlag, May 2008.
24. George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.
25. R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
26. Mohammad Reza Mousavi, Paul Le Guernic, Jean-Pierre Talpin, Sandeep Kumar Shukla, and Twan Basten. Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks. In *DATE '04*, page 10384, Washington, DC, USA, 2004. IEEE Computer Society.
27. Dumitru Potop-Butucaru and Benoît Caillaud. Correct-by-Construction Asynchronous Implementation of Modular Synchronous Specifications. *Fundam. Inf.*, 78(1):131–159, 2007.
28. S. Ramesh. Communicating Reactive State Machines: Design, Model and Implementation. *IFAC Workshop on Distributed Computer Control Systems*, September 1998.
29. S. Ramesh, Sampada Sonalkar, Vijay D'Silva, Naveen Chandra, and B. Vijayalakshmi. A Toolset for Modelling and Verification of GALS Systems. In Rajeev Alur and Doron Peled, editors, *CAV '04*, volume 3114 of *Lecture Notes in Computer Science*, pages 506–509. Springer, 2004.
30. K. Sollins. The TFTP Protocol (Revision 2). RFC 1350, Internet Engineering Task Force, July 1992.