HAL
open science

Verification of GALS Systems by Combining Synchronous Languages and Process Calculi

Hubert Garavel, Damien Thivolle

To cite this version:

Hubert Garavel, Damien Thivolle. Verification of GALS Systems by Combining Synchronous Languages and Process Calculi. Model Checking Software, Proceedings of the 16th International SPIN Workshop on Model Checking of Software SPIN'2009, Jun 2009, Grenoble, France. 10.1007/978-3-642-02652-2_20 . inria-00388819

HAL Id: inria-00388819

https://inria.hal.science/inria-00388819v1

Submitted on 27 May 2009

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

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

Verification of GALS Systems by Combining Synchronous Languages and Process Calculi

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

[1] INRIA Grenoble - Rhône-Alpes
655 avenue de l'Europe
38 330 Montbonnot Saint Martin – France
{Hubert.Garavel,Damien.Thivolle}@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* [17] 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 is 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 [24], ESTEREL [3], LUSTRE/SCADE [16], 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 long known in the world of hardware, where the term GALS (*Globally Asynchronous, Locally Synchronous*) was coined to characterise 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 conclusion [28] that asynchronism can be encoded in a synchronous process calculus, there have been approaches [18, 23, 29, 19] 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 recognised as first-class concepts, so verification tools often lack optimisations specific to asynchronous concurrency (e.g. partial orders, compositional minimisation, etc.). Other approaches extend synchronous languages to allow a certain degree of asynchrony, as in CRP [2], CRSM [31], or multiclock ESTEREL [4], but, to our knowledge, such extensions are not (yet) used in industry. Finally, we can mention approaches [15, 30] 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 ignore 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 [21], or PROMELA [20]. 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 behaviour 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.

Regarding related work, we can mention [32], which translates CRSM [31] 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 is the way locally synchronous components are integrated into a globally asynchronous system. The approach of [9] 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 [24]) 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 [22]. A translator exists that transforms LOTOS NT specifications into LOTOS specifications, thus enabling the use of the CADP toolbox [13] for verification and performance evaluation of the generated LOTOS specifications.

This paper is organised 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.

[3] `www.topcased.org`

2 Proposed methodology

This section explains how to make the connection between synchronous languages and process calculi. It takes the SAM and LOTOS NT languages as particular examples, but the principles of our approach are more general.

2.1 Synchronous programs seen as Mealy functions

A *synchronous program* is the synchronous parallel composition of one or several *synchronous components*. A synchronous component performs a sequence of discrete steps and maintains an internal state s. At each step, it receives a set of m input values $i_1, \ldots, i_m$ from its environment, computes (in zero time) a reaction, sends a set of n output values $o_1, \ldots, o_n$ to its environment, and moves to its new state s'. That is to say, it can be represented by a (usually deterministic) *Mealy machine* [27] i.e., a 5-tuple ($\mathcal{S}$, s_0, $\mathcal{I}$, $\mathcal{O}$, f) 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,
- $f \in \mathcal{S} \times \mathcal{I} \rightarrow \mathcal{S} \times \mathcal{O}$ is a *transition function* (also called a *Mealy function*) mapping the current state and the input alphabet to the next state and the output alphabet: $f(s, i_1...i_m) = (s', o_1...o_n)$.

When a synchronous program has several components, these components can communicate with each other by connecting the outputs of some components to the inputs of some other components. By definition of synchronous parallelism, at each step, all the components react simultaneously. Consequently, the composition of several components can also be modelled by a Mealy machine. For the synchronous ESTEREL and LUSTRE, a common format named OC (Object Code) has been proposed to represent those Mealy machines.

2.2 The SAM language

To illustrate our approach, we consider the case of the synchronous language SAM designed by Airbus, a formal description of which is given in [8]. A synchronous component in SAM is an automaton that 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 5-tuple (s_1, s_2, F, G, P), where:

- s_1 is the source state of the transition,
- s_2 is the destination state of the transition,
- F is a boolean condition on the input variables (the transition can be fired only when F evaluates to true),
- G is a set of output variables (when the transition is fired, the variables of G are set to true and the other output variables are set to false), and

– P is a priority integer value.

The priority values from transitions going out of the same state must be pairwise distinct. If a set of input values enables more than one outgoing transition from the current state, the transition with the lowest priority value is chosen, thus ensuring a deterministic execution. Priority values are notational conveniences that can be eliminated as follows: each transition (s_1, s_2, F, G, P) can be replaced by (s_1, s_2, F', G) 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 priority values strictly lower than P.

Each state has an implicit loop transition on itself that sets all the output ports to false and is fired if no other transition is enabled (its priority value is $+\infty$).

Fig. 1 gives an example of a SAM automaton. An interrogation mark precedes the condition F of each transition while an exclamation mark precedes its output variables list G. Priority values are attached to the source of the transitions.

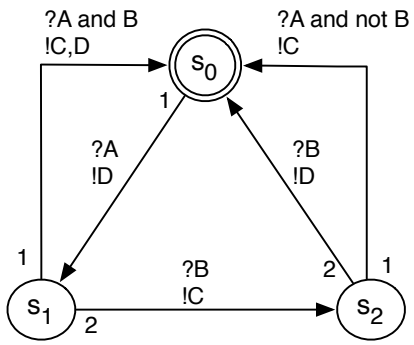

Fig. 1. Example automaton in SAM

SAM supports the synchronous composition of components. A global system in SAM has input and output ports. It is composed of one or several SAM components. Communication between these components 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.

2.3 Translating SAM into LOTOS NT

In this section, we illustrate how a SAM automaton can be represented by its Mealy function encoded in LOTOS NT. For instance, the SAM automaton of Fig. 1 can be encoded in LOTOS NT as follows:

```
type State is
  S0, S1, S2 -- this is an enumerated type
end type

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

We chose LOTOS NT rather than LOTOS because LOTOS NT functions are easier to use than LOTOS equations for describing Mealy functions and manipulating data in general. The imperative style of LOTOS NT makes this straightforward. Using LOTOS algebraic data types would have been more difficult given that LOTOS functions do not have "**out**" parameters. In this respect, LOTOS NT is clearly superior to LOTOS and other "traditional" value-passing process algebras; this contributes to the originality and elegance of our translation. Also, the fact that LOTOS NT functions execute atomically (i.e., they do not create "small step" transitions) perfectly matches the assumption that a synchronous program reacts in zero time.

A SAM system consisting of several SAM automata can also be translated to LOTOS NT easily. Because cyclic dependencies are forbidden, 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.

An alternative approach to translating a synchronous language L into LOTOS NT, if there exists a code generator from L to the C language, would be to invoke the Mealy function (presumably generated in C code) directly from a LOTOS NT program as an external function (a feature that is supported by LOTOS NT). This way, our approach could even allow mixing of components written in different synchronous languages.

2.4 Wrapping Mealy functions into LOTOS NT processes

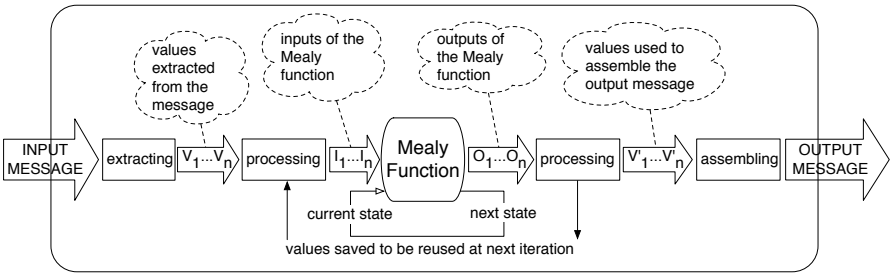

Fig. 2. A wrapper process in detail

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

Our approach to modelling GALS systems in asynchronous languages consists in encoding a synchronous program as a set of native types and functions in a given process calculus. But the Mealy function of a synchronous program alone cannot interact directly with an asynchronous environment. It needs to be *wrapped* (or *encapsulated*) in a process that handles the communications with the environment. This wrapper transforms the Mealy function of a synchronous component into an LTS (*Labelled Transition System*). In our case, the Mealy function is a LOTOS NT function and the wrapper is a LOTOS NT process.

The amount of processing a wrapper can do depends on the GALS system being modelled. Fig. 2 shows the basic processing usually done within a wrapper: extraction of the inputs, aggregation of the outputs, and storage of values for the next iteration. In certain cases, the wrapper can also implement extra behaviours not actually described by the Mealy function itself.

Once encapsulated in a wrapper process, the Mealy function corresponding to a synchronous program can be made to synchronise and communicate with other asynchronous processes using the parallel composition operator of LOTOS NT.

3 The TFTP case study

This case study was provided by Airbus to the participants of the TOPCASED project as a typical example of avionics embedded software. We first present a summary of the principles of the standard TFTP protocol, then we describe the adaptation of TFTP made by Airbus for plane/ground communications.

3.1 The standard TFTP protocol

TFTP [33] stands for *Trivial File Transfer Protocol*. It is a client/server protocol in which several clients can send (resp. receive) a file to (resp. from) one server. As it is designed to run over the UDP (*User Datagram Protocol*) protocol, the

TFTP protocol implements its own flow control mechanism. In order for the server to differentiate between clients, each file transfer is served on a different UDP port.

In a typical session, 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 divided into data fragments of equal size (except the last fragment whose size may be smaller), which are transferred sequentially. The server replies to an `RRQ` by sending in sequence the various data fragments (`DATA`) of the file and to a `WRQ` by sending an acknowledgement (`ACK`). When the client receives this acknowledgement, it starts sending the data fragments of the file. Each data fragment contains an order index which is used to check whether all data fragments are received consecutively. Each acknowledgement also carries the order index of the data fragment it acknowledges, or zero it if acknowledges a `WRQ`. A transfer ends when the acknowledgement of the last data fragment is received.

The protocol is designed to be robust. Any lost message (`RRQ`, `WRQ`, `DATA`, `ACK`) can be retransmitted after a timeout. Duplicate (resent because of a timeout) acknowledgements are discarded upon receipt to avoid the *Sorcerer's Apprentice* bug [5]. The TFTP standard suggests the use of *dallying*, i.e. waiting for a while after sending the final acknowledgement in case this acknowledgement is lost before reaching the other side (that will eventually resend its final data fragment after a timeout).

If an error (memory shortage, fatal error, etc.) occurs, the client or the server sends an error message (`ERROR`) to abort the transfer.

3.2 The Airbus variant of the TFTP protocol

When a plane reaches its final parking position, it is connected to the airport using an Ethernet network. The ground/plane communication protocol currently in use is a very simple and certified to be correct. Airbus asked us to study a more complex protocol, a variant of the TFTP, which might be of interest for future generations of planes. The main differences with the standard TFTP are the following:

- In the protocol stack considered by Airbus, this TFTP variant still runs above the UDP layer but below an avionic communication protocol layer (e.g. ARINC 615a). The files carried by the TFTP variant are frames of the upper layer protocol.
- Each side of the TFTP variant has the ability to be both a client and a server, depending on the upper layer requests.
- Each server communicates with one single client because there is a unique TFTP instance reserved for each plane that lands in the airport. This removes the need for modelling the fact that a server can serve many different clients on as many different UDP ports.

In the rest of this paper, we will use the name TFTP to refer to this protocol variant studied by Airbus.

The behaviour of a TFTP protocol entity was specified by Airbus as a SAM system consisting of one SAM automaton with 7 states, 39 transitions, 15 inputs and 11 outputs.

Airbus was interested in knowing how this TFTP variant would behave in an unreliable environment, in which messages sent over the UDP layer could be lost, duplicated, or reordered.

4 Formal specification of the case study

We have modelled a specification consisting of two TFTP protocol entities connected by two UDP media. As shown in Fig. 3, the TFTP protocol entities are two instances of the same LOTOS NT process, whose behaviour is governed by the Mealy function of the SAM TFTP automaton. We manually translated this function into 215 lines of LOTOS NT code (including the enumerated type encoding the states of the SAM automaton). The media are also two instances of the same LOTOS NT process that models the behaviour of UDP.

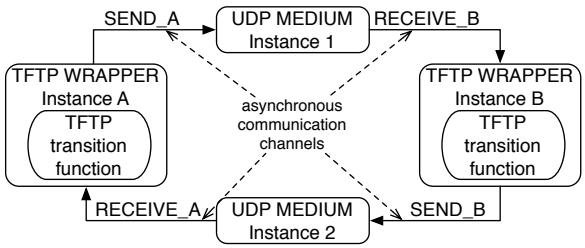

Fig. 3. Asynchronous connection of two TFTP processes via UDP media

We have defined two versions of the LOTOS NT wrapper process encapsulating the TFTP Mealy function. The *basic* TFTP process is the simplest one; it is modelled after Airbus recommendations to connect two TFTP SAM automata head-to-tail in an asynchronous environment. The *accurate* TFTP process is more involved: it is closer to the standard TFTP protocol and copes with limitations that we detected in the basic TFTP process.

4.1 Modelling the basic TFTP entities

The basic TFTP process, as shown by Fig. 4, is a simple wrapper (260 lines of LOTOS NT) around the Mealy function and does no processing on its own. The idea behind this wrapper is to asynchronously connect output ports of one TFTP automaton to corresponding input ports of the other side. Inputs of the Mealy function that can neither be deduced from the input message nor from values stored at the previous iteration are assigned a random boolean value.

A key issue with this design is how to determine if two successive data fragments are different, or if they are the same fragment sent twice. For this

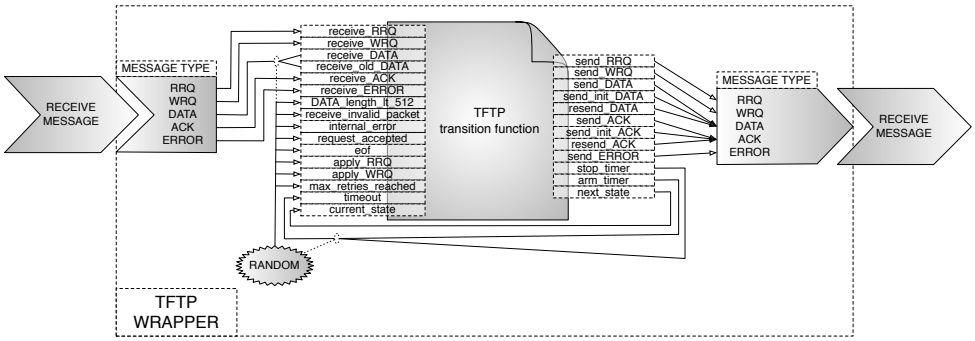

Fig. 4. Basic TFTP process

purpose, the SAM automaton has different input ports (`receive_DATA` and `receive_old_DATA`) and different output ports (`send_DATA` and `resend_DATA`). However, the basic TFTP wrapper is just too simple to interface with these ports in a satisfactory manner. For this reason, we had to refine this wrapper as explained in the next section.

4.2 Modelling the accurate TFTP entities

We developed a more *accurate* TFTP wrapper process (670 lines of LOTOS NT) that receives and sends "real" TFTP frames (as defined in the TFTP standard).

In our model, we assume the existence of a finite set of files (each represented by its file name, which we encode as an integer value) in which each TFTP process can pick up files to write to or read from the other side. Each `RRQ` and `WRQ` frame carries the name of the requested file. The contents of each file are modelled as a sequence of fragments, each fragment being represented as a character. Each `DATA` frame carries three values: a file fragment, an order index for the fragment, and a boolean value indicating whether this is the last fragment of the file. Each `ACK` frame carries the order index of the `DATA` frame it acknowledges, or zero if it acknowledges a `WRQ`.

In order to fight state explosion in the latter phases, we restrict nondeterminism by constraining each TFTP process to select only those files belonging to a "read list" and "write list". Whenever there is no active transfer, a process can randomly choose to send an `RRQ` request for the first file in its read list or a `WRQ` request for the first file 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, for instance the name of the file being transferred, the index value of the last data fragment or acknowledgement received or sent, a boolean indicating whether the last data fragment received is the last one, etc.

4.3 Modelling the UDP media

The two LOTOS NT processes describing the UDP media are not derived from a SAM specification: they have been written by hand.

These processes should reproduce accurately the behaviour of an UDP layer over an Ethernet cable connecting the plane and the ground. As UDP is a connection-less protocol without error recovery mechanism, any error that is not detected and corrected by the lower networking layers will be propagated to the upper layers (i.e., TFTP in our case). These errors are: message losses, message reordering, and message duplications. Message losses are always possible, due to communication failures. Reordering of messages should be limited in practice (as modern routers use load-balancing policies that usually send all related packets through the same route), but we cannot totally exclude this possibility. Message duplications may only occur if the implementation of the lower networking layers is erroneous, so we can discard this possibility.

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 as they are received. The second process models the case where message reordering can happen. It uses a bag as a buffer. 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 line long.

4.4 Interconnecting TFTP entities and UDP media

To compose the TFTP protocol entities and the UDP media asynchronously as illustrated in Fig. 3, 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: basic TFTP specification with bag media, basic TFTP specification with FIFO media, accurate TFTP specification with bag media, and accurate TFTP specification with FIFO media.

5 Functional verification by model checking

In this section, we detail how to generate the state spaces for the specifications and how to define correctness properties characterising the proper behaviour of these specifications. Then, we discuss the model checking results obtained using CADP.

5.1 State space generation

LOTOS NT specifications are automatically translated into LOTOS specifications (using the LPP/LNT2LOTOS [7] compilers) which are, in turn, compiled into LTSs (*Labelled Transition Systems*) using the CÆSAR.ADT [14] and CÆSAR [10] compilers of CADP.

One important issue in model checking is the state space explosion problem. Because of this, we restrict the buffer size n of the UDP media processes to small values (e.g., $n = 1, 2, 3...$). In the case of the accurate TFTP we also limit the size of each file to two fragments (this is enough to exercise all the transitions of the SAM automaton) and we constrain the number of files exchanged between the two TFTP protocol entities by bounding the lengths of the read and write lists. To cover all the possibilities, we consider four scenarios:

- Scenario 1: TFTP entity A writes one file.;
- Scenario 2: TFTP entities A and B both write one file;
- Scenario 3: TFTP entity A writes one file and B reads one;
- Scenario 4: TFTP entities A and B both read one file;

Additionally, we make use of the compositional verification tools available in CADP to fight state explosion. Compositional verification is a *divide and conquer* approach that allows significant reductions in time, memory, and state space size. Applied to the TFTP case study, this approach consists in generating the LTSs for all the four processes (two TFTP processes and two media processes), minimising these LTSs according to strong bisimulation (using the BCG_MIN tool of CADP), and composing them progressively in parallel (using the EXP.OPEN and GENERATOR tools of CADP) by adding one LTS at a time.

For instance, on the example of basic TFTP specification with two *BAG(2)* media, it took 7 minutes and 56 seconds on a 32-bit machine (2.17 Ghz Intel Core 2 Duo processor running Linux with 3 GB of RAM), to directly generate the corresponding LTS, which has 2,731,505 states and 11,495,662 transitions. Using compositional verification instead, it only takes 13.9 seconds to generate, on the same machine, a strongly equivalent, but smaller, LTS with 542,078 states and 2,543,930 transitions only.

Practically, compositional verification is made simple by the SVL [12] script language of CADP. SVL lets the user write compositional verification scenarios at a high level of abstraction and takes care of all low level tasks, such as invoking the CADP tools with appropriate command-line options, managing all temporary files, etc.

Tables 1 and 2 illustrate the influence of the buffer size on the state spaces of the basic and accurate TFTP specifications, respectively. In these tables, the hyphen symbol ("–") indicates the occurrence of state space explosion.

5.2 Temporal logic properties

After a careful analysis of the standard TFTP protocol and discussions with Airbus engineers, we specified a collection of properties that the TFTP specification

Medium	Minimised Medium LTS		Entire Specification Generation		
	States	Transitions	States	Transitions	Time (s)
BAG(1)	13	60	20,166	86,248	10.49
BAG(2)	70	294	542,078	2,543,930	13.90
BAG(3)	252	1,008	6,698,999	32,868,774	54.89
BAG(4)	714	2,772	–	–	–
FIFO(1)	13	60	20,166	86,248	9.95
FIFO(2)	85	384	846,888	3,717,754	15.13
FIFO(3)	517	2,328	31,201,792	137,500,212	200.32
FIFO(4)	3,109	13,992	–	–	–

Table 1. LTS generation for the basic TFTP

Medium	Minimised Medium LTS		Entire Specification Generation		
	States	Transitions	States	Transitions	Time (s)
BAG(1)	31	260	71,974	319,232	20.04
BAG(2)	231	1,695	985,714	4,683,197	27.44
BAG(3)	1,166	7,810	6,334,954	31,272,413	78.28
BAG(4)	4,576	28,655	–	–	–
FIFO(1)	31	260	71,974	319,232	20.29
FIFO(2)	321	2,640	1,195,646	5,373,528	29.26
FIFO(3)	3,221	26,440	18,885,756	85,256,824	174.15
FIFO(4)	32,221	264,440	–	–	–

Table 2. LTS generation for the accurate TFTP (scenario 1)

should verify. These properties were first expressed in natural language and then translated into temporal logic formulas.

For the basic TFTP specification, we wrote a first collection of 12 properties using modal μ-calculus (extended with regular expressions as proposed in [25]). These properties were evaluated using the EVALUATOR 3.5 model checker of CADP. We illustrate two of them here:

- The TFTP automaton has two output ports `arm_timer` and `stop_timer` that respectively start and stop the timer used to decide when an incoming message should be considered as lost. The following property ensures that between two consecutive `stop_timer` actions, there must be an `arm_timer` action. It states that there exists no sequence of transitions containing two `stop_timer` actions with no `arm_timer` action in between. The suffix "`_A`" at the end of transition labels indicates that this formula holds for TFTP protocol entity A. There is a similar formula for entity B.

```
[ true* . "STOP_TIMER_A" . not ("ARM_TIMER_A")* .
  "STOP_TIMER_A" ] false
```

- When a TFTP protocol entity receives an error, it must abort the current transfer. The following property ensures that receiving an error cannot be

followed by sending an error. It states that there exists no sequence of transitions featuring the reception of an error directly followed by sending an error:

```
[ true* . "RECEIVE_A !ERROR" . "SEND_A !ERROR" ] false
```

For the accurate TFTP specification, the collection of 12 properties we wrote for the basic TFTP specification can be reused without any modification, still using EVALUATOR 3.5 to evaluate them. We also wrote a second collection of 17 new properties that manipulate data in order to capture the messages exchanged between the TFTP protocol entities. These properties could have been written using the standard μ-calculus but they would have been too verbose. Instead, we used the MCL language [26], which extends the modal μ-calculus with data manipulation constructs. Properties written in the MCL language can be evaluated using the EVALUATOR 4.0 [26] model checker of CADP. We illustrate two of these new properties below:

- Data fragments must be sent in proper order. We chose to ensure this by showing that any data fragment numbered x cannot be followed by a data fragment numbered y, where $y < x$, unless there has been a re-initialisation (transfer succeeded or aborted) in between. This property is encoded as follows:

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

- Resent write requests must be replied to, in the limits set by the value of the maximum number of retries. The following formula states that for every write request received and accepted, it is possible to send the acknowledgement more than once, each time (within the limit of MAX_RETRIES_A) the write request is received – the r $\{p\}$ notation meaning that the regular formula r must be repeated p times.

```
[ not {RECEIVE_A !"WRQ" ...}* . {RECEIVE_A !"WRQ" ?n:Nat} .
  i . {SEND_A !"ACK" !0 of Nat}
] forall p:Nat among {1 ... MAX_RETRIES_A ()} .
  < ( not (REINIT_A or {RECEIVE_A !"WRQ" !n})* .
      {RECEIVE_A !"WRQ" !n} . {SEND_A !"ACK" !0 of Nat}
    ) {p} > true
```

5.3 Model checking results

Using the EVALUATOR 3.5 model checker, we evaluated all properties of the first collection on all the LTSs generated for the basic and accurate TFTP specifications. Using the EVALUATOR 4.0 model checker, we did the same for all properties in the second collection on all the LTSs generated for the accurate TFTP specifications.

Several of the first collection of 12 properties did not hold on either the basic or the accurate TFTP specifications. This enabled us to find 11 errors in the

TFTP automaton. From the two properties presented in Section 5.2 for the first collection, the first held while the second did not.

The verification of the second collection of 17 properties specially written for the accurate TFTP specifications led to the discovery of an additional 8 errors. From the two properties presented in Section 5.2 for the second collection, the first held while the second did not.

For both the basic and accurate TFTP specifications, we observed that the truth values of all these formulas did not depend on the sizes of bags or FIFOs. Notice that, because EVALUATOR 3.5 and 4.0 can work on the fly, we could have applied them directly to the LOTOS specifications generated for the TFTP instead of generating the LTSs first. Although this might have enabled us to handle larger state spaces, we did not chose this approach, as we felt that further increasing the bag and FIFO sizes would not lead to different results.

Regarding the amount of time needed to evaluate formulas, we observed that it takes in average 35 seconds per formula on an LTS having 3.4 million states and 19.2 million transitions (basic TFTP specification) and 6.5 minutes per formula on an LTS having 18.2 million states and 88 million transitions (accurate TFTP specification).

In total, we found 19 errors, which were reported to Airbus and were acknowledged as being actual errors in the TFTP variant. We also suggested changes in the TFTP automaton to correct them. As stated in Section 3.2, it is worth noticing that these errors only concern a prototype variant of TFTP, but not the communication protocols actually embedded in planes and airports. While some of these errors could have been found by a human after a careful study of the automaton, some others are more subtle and would have been hard to detect just by looking at the TFTP automaton: for instance, the fact that if both TFTP entities send a request (`RRQ` or `WRQ`) at the same time, both requests are just ignored.

6 Performance evaluation by simulation

In spite of the errors we detected, the TFTP automaton can always recover with timeouts, i.e. by waiting long enough that the timer expires. However, these extra timeouts and additional messages cause a performance degradation that needed to be quantified.

There are several approaches to performance evaluation, namely queueing theory, Markov chains (the CADP toolbox provides tools for *Interactive Markov Chains* [11]), and simulation methods. For the TFTP case study, we chose the latter approach.

6.1 Simulation methodology with CADP

To quantify the performance loss caused by the errors, an "optimal" model was needed to serve as a reference. For this purpose, we wrote a TFTP Mealy function in which all the errors have been corrected. We also produced, for each error e,

a TFTP Mealy function in which all the errors but e had been corrected, so as to measure the individual impact of e on the global performance.

State space explosion does not occur with simulation. This allowed us to increase the complexity of our models:

- The number of files exchanged was set to 10,000. Before each simulation, these files are randomly distributed in the read and write lists of the TFTP.
- The file size was increased to be between 4 and 10 fragments. File fragments are assumed to be 32 kB each. File contents are randomly generated before each simulation. A simulation stops when all the files in the read and write lists have been transferred.
- We used bag UDP media with a buffer size of 6.

We considered two simulation scenarios:

1. One TFTP protocol entity acts as a server and initiates no transfer. The other acts as a client that reads files from and writes files to the server. This is a realistic model of actual ground/plane communications.
2. Both TFTP protocol entities can read and write files. This is a worst-case scenario in which the TFTP protocol entities compete to start file transfers. This can happen under heavy load and Airbus engineers recognised it ought to be considered.

To perform the simulations, we adapted the EXECUTOR tool of CADP, which can explore random traces in LOTOS specifications on the fly. By default, in EXECUTOR, all transitions going out of the current state have the same probability of being fired. To obtain more realistic simulation traces, we modified EXECUTOR (whose source code is available in CADP) to assign different probabilities to certain transitions. Namely, we gave to timeouts and message losses (resp. to internal errors) a probability that is 100 (resp. 10,000) times smaller than the probability of all other transitions. In the bag UDP media, older messages waiting in the buffers were given higher chance than newer messages to be chosen for delivery.

To each transition, we also associated an estimated execution time, computed as follows:

- The UDP media are assumed to have a speed of 1 MB/s and a latency of 8 ms.
- Receiving or sending an `RRQ`, a `WRQ`, or an `ACK` takes 2 ms (one fourth of the latency)
- Receiving or sending a `DATA` takes 18 ms: 2 ms from the medium latency plus half the time required to send 32 kB at 1 MB/s.
- For the timeout values, we tried 20 different values in total, ranging from 50 ms to 1 second, varying by steps of 50 ms.
- All other transitions have an estimated execution time of 0 ms.

For each error e, for both simulations scenario, and for each timeout value, we ran ten simulations on the TFTP specification in which all errors but e had been corrected. We then analysed each trace produced by these simulations to compute:

– its execution time, i.e. the sum of the estimated execution times for all the transitions present in the trace,
– the number of bytes transferred during the simulation, which is obtained by multiplying the fragment size (32 kB) by the number of file fragments initially present in the read and write lists.

Dividing the latter by the former gives a transfer speed, the mean value of which can be computed over the set of simulations.

6.2 Simulation results

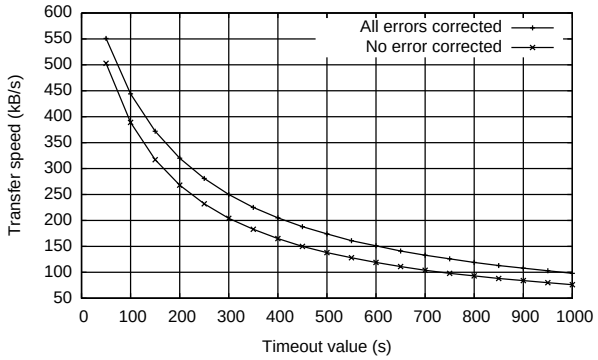

Fig. 5. Simulation results for scenario 1.

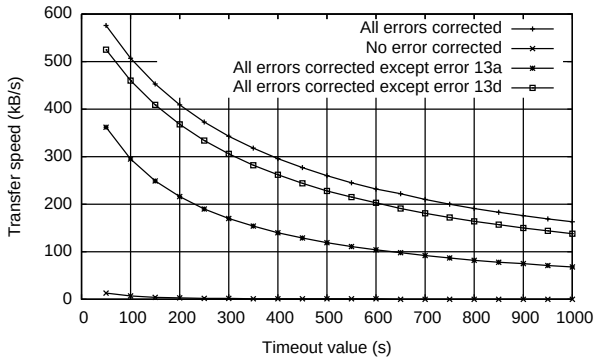

Fig. 6. Simulation results for scenario 2.

For the simulation scenario 1, we observed (see Fig. 5) that the TFTP specification in which all the errors have been corrected performs 10% faster than the original TFTP specification containing all the 19 errors.

For the simulation scenario 2, the original TFTP specification has a transfer speed close to zero, whatever the timeout value chosen. This confirms our

initial intuition that the errors we detected prevent the TFTP prototype from performing correctly under heavy load (this intuition was at the source of our performance evaluation study for the TFTP). After all errors have been corrected, the numerical results obtained for scenario 2 are the same as for the simulation scenario 1. We observed that certain errors play a major role in degrading the transfer speed. For instance (see Fig. 6), this is the case with errors 13a (resp. 13c), which are characterised by the fact that the TFTP automaton, after sending the last acknowledgement and entering the dallying phase, ignores incoming read (resp. write) requests, whereas it should either accept or reject them explicitly.

7 Conclusion

In this paper, we have proposed a simple and elegant 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 process calculi. It allows us to reuse without any modification the existing compilers for synchronous languages, together with the existing compilers and verification tools for process calculi.

We 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 (sets of) synchronous components into Mealy machines — which is almost always the case — and to any process calculus equipped with asynchronous concurrency and user-defined functions.

Regarding future work, we received strong support from Airbus. Work has already been undertaken to automate the translation from SAM to LOTOS NT and to verify another avionics embedded software system. We would also like to compare our simulation results against results from "traditional" simulation tools and to apply our approach to other synchronous languages than SAM.

Acknowledgements

We are grateful to Patrick Farail and Pierre Gaufillet (Airbus) for their continuing support and to Claude Helmstetter (INRIA/Vasy), Pascal Raymond (CNRS/Verimag), and Robert de Simone (INRIA/Aoste), as well as the anonymous referees, 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 Champelovier, Xavier Clerc, and Hubert Garavel. Reference Manual of the LOTOS NT to LOTOS Translator, Version 4G. Internal Report, INRIA/VASY, January 2009.
8. Xavier Clerc, Hubert Garavel, and Damien Thivolle. Présentation du langage SAM d'Airbus. Internal Report, INRIA/VASY, 2008. Available from TOPCASED forge: `http://gforge.enseeiht.fr/docman/view.php/33/2745/SAM.pdf`.
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 *FME'02*, volume 2391 of *LNCS*, pages 410–429, Copenhagen, Denmark, July 2002. Springer-Verlag.
12. Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.
13. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *CAV'07*, volume 4590 of *LNCS*, pages 158–163, Berlin, Germany, July 2007. Springer-Verlag.
14. Hubert Garavel and Philippe Turlier. CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In Rachida Dssouli and Gregor v. Bochmann, editors, *Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'93 (Montréal, Canada)*, 1993.
15. Alain Girault and Clément Ménier. Automatic Production of Globally Asynchronous Locally Synchronous Systems. In *EMSOFT '02*, pages 266–281, London, UK, 2002. Springer-Verlag.
16. 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.

17. Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic, 1993.
18. Nicolas Halbwachs and Siwar Baghdadi. Synchronous Modelling of Asynchronous Systems. In *EMSOFT '02*, pages 240–251, London, UK, 2002. Springer-Verlag.
19. 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.
20. G.J. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.
21. 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.
22. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.
23. 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.
24. F. Maraninchi and Y. Rémond. Argos: an Automaton-Based Synchronous Language. *Computer Languages*, 27(1–3):61–92, October 2001.
25. Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.
26. Radu Mateescu and Damien Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *FM'08*, number 5014 in LNCS, pages 148–164, Turku, Finland, May 2008. Springer-Verlag.
27. George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.
28. R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
29. 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.
30. Dumitru Potop-Butucaru and Benoît Caillaud. Correct-by-Construction Asynchronous Implementation of Modular Synchronous Specifications. *Fundam. Inf.*, 78(1):131–159, 2007.
31. S. Ramesh. Communicating Reactive State Machines: Design, Model and Implementation. *IFAC Workshop on Distributed Computer Control Systems*, September 1998.
32. S. Ramesh, Sampada Sonalkar, Vijay D'Silva, Naveen Chandra, and B. Vijayalakshmi. A Toolset for Modelling and Verification of GALS Systems. In *CAV '04*, volume 3114 of *LNCS*, pages 506–509. Springer-Verlag, 2004.
33. K. Sollins. The TFTP Protocol (Revision 2). RFC 1350, Internet Engineering Task Force, July 1992.