The SIGNAL Approach to the Design of System Architectures

Abdoulaye Gamatié, Thierry Gautier

▶ To cite this version:

Abdoulaye Gamatié, Thierry Gautier. The SIGNAL Approach to the Design of System Architectures. 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03), Apr 2003, Huntsville, Alabama, United States. pp.80-88, 10.1109/ECBS.2003.1194786 . hal-00541913

HAL Id: hal-00541913
https://hal.science/hal-00541913

Submitted on 1 Dec 2010

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.

The SIGNAL Approach to the Design of System Architectures*

Abdoulaye GAMATIÉ, Thierry GAUTIER
IRISA / INRIA
F-35042 RENNES, France
{agamatie, gautier}@irisa.fr

Abstract

Modeling plays a central role in system engineering. It significantly reduces costs and efforts in the design by providing developers with means for cheaper and more relevant experimentations. So, design choices can be assessed earlier. The use of a formalism, such as the synchronous language SIGNAL *which relies on solid mathematical foundations for the modeling, allows validation. This is the aim of the methodology defined for the design of embedded systems where emphasis is put on formal techniques for verification, analysis, and code generation. This paper mainly focuses on the modeling of architecture components using* SIGNAL. *For illustration, we consider the modeling of a bounded FIFO queue, which is intended to be used for communication protocols. We bring out the capabilities of* SIGNAL *to allow specifications in an elegant way, and we check few elementary properties on the resulting model for correctness.*

1. Introduction

Nowadays, systems in general are more and more large and complex. Obviously, the engineering becomes very delicate since the complexity of data structures and computation algorithms is challenging. On the other hand, the design cycle usually involves multiple formalisms and various tools. A major drawback in such a context is that the design and checking tasks are inherently long and complex. In the case of distributed embedded systems, there are additional difficulties: on the one hand, such systems have to be separated efficiently into components, and suitable communication mechanisms between these components must be provided; on the other hand, the validation of the whole is required.

Among solutions that have been proposed to overcome the above obstacles, there are Architecture Description Languages (ADLs) [7], the Unified Modeling Language (UML) [16], or the synchronous technology [11]. They all provide formalisms and tool-sets that help for the description of systems. Solutions which adopt formal methods are widely accepted as a confident way for guaranteeing the quality of designs. As a matter of fact, verification and validation are facilitated. So, it appears desirable for a design formalism to have a well-defined formal semantics. Unfortunately, this is not the case for all the solutions (for instance, UML only has a semi-formal semantics).
The synchronous technology emerges as one of the most promising ways for guaranteeing a safe design of embedded systems. It offers practical design assistance tools with a formal basis. These include possibilities of high level specifications, modular verification of properties on these specifications, automatic code generation through formal transformations, and validation of the generated code against specifications. As a result, earlier architectural choices and behavioral simulation are enabled, and design ambiguities and errors can be significantly reduced. POLYCHRONY, the programming environment of the synchronous language SIGNAL [5], implemented by INRIA[1] (`http://www.irisa.fr/espresso`) incorporates all these functionalities.

A major objective of our work is the definition and implementation of an enhanced methodology for the design of embedded systems within POLYCHRONY. This methodology must significantly reduce the risk of design errors and shorten overall design times. Earlier results have been established during the SACRES project [9]. The main add-on of the methodology is to allow automatic generation of efficient and validated distributed code from the specifications, entirely replacing the manual coding phase still employed in current industrial design flows.
In this paper, we give an overview of the methodology, but we rather concentrate on the approach of modeling compo-

*This work has been supported by the european project IST SAFEAIR (Advanced Design Tools for Aircraft Systems and Airborne Software) [10] (`http://www.safeair.org/`).

[1] There is also an industrial version, SILDEX, implemented and commercialized by TNI-Valiosys (`http://www.tni-valiosys.com`).

nents used in architecture descriptions. The remainder of the paper is organized as follows: section 2 presents the SIGNAL language. Then, in section 3, we introduce the methodology. We highlight its main steps from specification to implementation. In section 4, we illustrate the design of architecture components through the modeling of a *First In First Out* queue. We verify some properties (e.g. safety) on the resulting model for correctness. Finally, a conclusion is given in section 5.

2. The SIGNAL language

The underlying theory of the synchronous approach [2] is that of discrete event systems and automata theory. Time is logical: it is handled according to partial order and simultaneity of events. Durations of execution are viewed as constraints to be verified at the implementation level. Typical examples of synchronous languages [11] are: ESTEREL, LUSTRE, SIGNAL. They differ mainly from each other in their programming style. The first one adopts an imperative style whereas the two others are dataflow oriented (LUSTRE is functional and SIGNAL is relational). However, there have been joint efforts to provide a common format DC+ [1], which allows the interoperability of tools.

The SIGNAL language [5] handles unbounded series of typed values $(x_t)_{t \in \mathbb{N}}$, called *signals*, denoted as `x` in the language, and implicitly indexed by discrete time (denoted by t in the semantic notation). At a given instant, a signal may be present, then it holds a value; or absent, then it is denoted by the special symbol $\perp$ in the semantic notation. There is a particular type of signals called `event`. A signal of this type is always *true* when it is present (otherwise, it is $\perp$). The set of instants where a signal `x` is present is called its *clock*. It is noted as `^x` (which is of type `event`) in the language. Signals that have the same clock are said to be *synchronous*. A SIGNAL program, also called *process*, is a system of equations over signals.

The kernel language. SIGNAL relies on a handful of primitive constructs which are combined using a composition operator. These are:

- **Functions.** `y:= f(x1,...,xn)`, where $y_t \neq \perp \Leftrightarrow x1_t \neq \perp \Leftrightarrow ... \Leftrightarrow xn_t \neq \perp$, and $\forall t$: $y_t = f(x1_t, ..., xn_t)$.
- **Delay.** `y:= x $ 1 init y0`, where $x_t \neq \perp \Leftrightarrow y_t \neq \perp$, $\forall t > 0$: $y_t = x_{t-1}, y_0 = y0$.
- **2-args down-sampling.** `y:= x when b`, where $y_t = x_t$ if $b_t = true$, else $y_t = \perp$.
- **Deterministic merging.** `z:= x default y`, where $z_t = x_t$ if $x_t \neq \perp$, else $z_t = y_t$.
- **Hiding.** `P where x` denotes that the signal `x` is local to the process `P`.
- **Synchronous parallel composition** of `P` and `Q`, encoded by `(| P | Q |)`. It corresponds to the union of systems of equations represented by `P` and `Q`.

These core constructs are of sufficient expressive power to derive other constructs for comfort and structuring. The following operators are also used in the next sections:

- **1-arg down-sampling.** `y:= when b`, where $y_t = true$ if $b_t = true$, else $y_t = \perp$.
- **Clock union.**[2] `y:= x1 ^+ ... ^+ xn`, where `y` is of `event` type, and denotes the set of instants where at least one signal `xi` occurs.
- **Synchronizer.** `x1 ^= ... ^= xn`, where $x1_t \neq \perp \Leftrightarrow ... \Leftrightarrow xn_t \neq \perp$ (i.e. $x1, ..., xn$ are synchronous).
- **Sliding window.** `y:= x window n init y0`, where $\forall t \geq 0$, and $i \in 0 .. n-1$:
$((t + i \geq n) \Rightarrow (Y_t[i] = X_{t-n+i+1})) \wedge$
$((1 \leq t + i < n) \Rightarrow (Y_t[i] = y0[t - n + i + 2]))$
- **Memory.** `y:= x cell b init y0`, defined as:
```
(| y:= x default (y$1 init y0)
 | y ^= x ^+ (when b) |)
```

The next example illustrates the meaning of the *sliding window* and the *memory* operators. Let us consider a process defined as follows:

```
(| y := x window 3 init [-1,0]
 | z := x cell b init 0
 |).
```

Signals `x` and `z` are of integer type, `b` is a boolean, and `y` is a 3-array of integers. A possible run is:

x :	$\perp$	1	2	$\perp$	$\perp$	3	...
b :	*t*	$\perp$	*f*	*t*	*f*	*t*	...
y :	$\perp$	$[-1, 0, 1]$	$[0, 1, 2]$	$\perp$	$\perp$	$[1, 2, 3]$	...
z :	0	1	2	2	$\perp$	3	...

3. A methodology for the design of embedded systems

This methodology relies on the theory of desynchronization [3], which defines the formal basis for an effective implementation of synchronous programs on asynchronous architectures, without changing their original semantics.
Basically, the design of distributed embedded systems consists in the distribution of a SIGNAL program representing a functional graph of flows, operators and dependencies.

[2] Similarly, *intersection* and *difference* of clocks are defined.

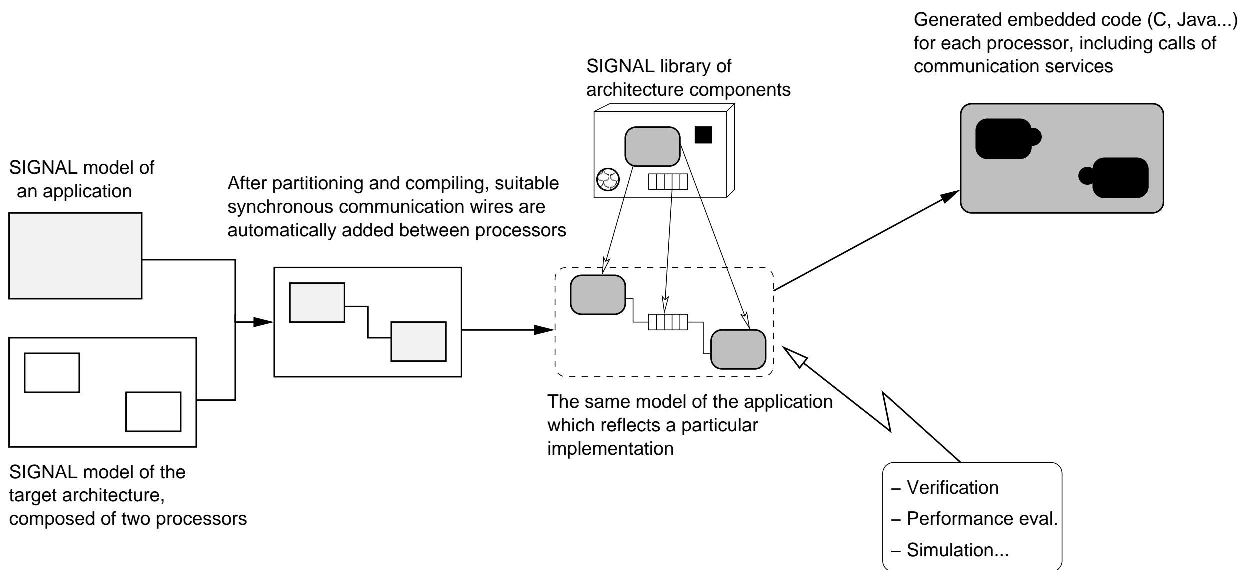

Figure 1. An overview of the SIGNAL methodology for the design of embedded systems.

The target architecture is composed of a set of possibly heterogeneous execution components (processors, micro-controllers...).
A general comment is that the level of detail at which the architecture needs to be known depends quite a lot on the refinement of the mapping to the chosen architecture. This means that in the simplest cases, the amount of data required is fairly small, and simple to assess:

- the set of processors or tasks, and the mapping from operations or sub-processes in the application specification to those processors or tasks. This information enables the partitioning of the graph into sub-graphs grouped according to the mapping.
- the topology of the network of processors, the set of connections between processors, and a mapping from inter-process communications to these communication links. This is useful in the case of signals exchanged between processes located on different processors or tasks, if several of them have to be routed through the same communication medium.
- a definition of the set of system-level primitives used e.g. for communications (readings and writings to the media). Roughly, this amounts to the profiles of the function library to which the code generated for the application will have to be linked.

Further degrees of refinement of the description may be required for a better architecture-adaptation: for example, concerning communications, the type and nature of the links (that could be implemented using shared variables, synchronous or asynchronous communications...). If the target architecture features an OS, the required model consists basically in the profile of the corresponding functions. For instance, according to the degree of use of the OS, we need models of synchronization gates, communications (possibly including routing between processors) or tasking functions (in the case of un-interruptible tasks: start and stop; in the case of interruptible tasks: suspend and resume, assignment and management of priority levels), etc. A specification of such functions has been addressed in [8], where a component library (process, communication and synchronization mechanisms...) has been defined for the design of modular avionics architectures.

In **Figure 1**, we have illustrated the whole design chain. First, the respective SIGNAL descriptions of an application software and the hardware architecture (mainly processors) are given. Then, the application is manually partitioned onto the target architecture. The compilation[3] of the whole determines which information has to be exchanged by processors, and communication wires are automatically added between processors. These communications have a *synchronous*[4] semantics. Of course, if the application has to be deployed on an asynchronous architecture, the instantaneousness of the added communications will be lost. However, (bounded) communication mechanisms can be easily modeled with SIGNAL. In that case, the models can be used in the description of the architecture so as to obtain a model of GALS (Globally Asynchronous Locally Synchronous)

[3]The main part of the compilation is called *clock calculus* in POLYCHRONY.

[4]In other words, zero-delay communications: a sent message is instantaneously received.

```
process basic_FIFO =
     { type message_type; integer fifo_size, message_type default_mess; }
     ( ? message_type mess_in; event access_clock;
       ! message_type mess_out; integer nbmess; boolean OK_write, OK_read;
     )
  (| nbmess := ((prev_nbmess+1) when (^mess_in) when OK_write) default
               ((prev_nbmess-1) when (^mess_out) when OK_read) default
               prev_nbmess                                                    (1.a)
   | prev_nbmess := nbmess$1 init 0                                           (1.b)
   | OK_write := prev_nbmess<fifo_size                                        (1.c)
   | OK_read := prev_nbmess>0                                                 (1.d)
   | queue := (mess_in window fifo_size) cell (^access_clock)                 (1.e)
   | mess_out := prev_mess_out when (not OK_read) when (^mess_out) default
                 queue[fifo_size - prev_nbmess] when (^mess_out)              (1.f)
   | prev_mess_out := mess_out $ 1 init default_mess                          (1.g)
   | nbmess ^= access_clock                                                   (1.h)
   |)
 where
   integer prev_nbmess; [fifo_size]message_type queue;
   message_type prev_mess_out;
 end;%basic_FIFO%
```

mess_in	:	⊥	4	6	⊥	⊥	⊥	5	7	8	⊥	⊥	...
access_clock	:	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	...
mess_out	:	-1	⊥	⊥	4	6	6	⊥	⊥	⊥	7	8	...
nbmess	:	0	1	2	1	0	0	1	2	2	1	0	...
OK_write	:	*t*	*t*	*t*	*f*	*t*	*t*	*t*	*t*	*f*	*f*	*t*	...
OK_read	:	*f*	*f*	*t*	*t*	*t*	*f*	*f*	*t*	*t*	*t*	*t*	...

Figure 2. A model of basic_FIFO with an associated trace.

type. The SIGNAL compiler is used to establish and verify the conditions under which the asynchronous behavior of the application model is equivalent to the synchronous one (this is addressed by the so-called *endo/isochrony properties* [3]). So, one solution is to define a library of components which can be used to model various communication mechanisms in architectures. For instance, the components presented in [8] can be used to describe avionics applications. They have been modeled with SIGNAL in order to take advantage of its formal basis for architecture analysis. So, for a particular implementation, we only have to pick up the required component model from a pre-defined library and insert it into the current system description. Afterwards, we can use the techniques and tools available in POLYCHRONY to assess the final implementation; or generate separate embedded code for each processor, along with the suitable communication protocol. The protocol preserves the semantics of synchronous communication even though an asynchronous communication medium is used. We already mentioned that verification and validation are essential to our approach. The SIGNAL compiler and tools like SIGALI (a model checker, see in the next sections) help for property checking (e.g. safety). Performance evaluation is also possible using implemented techniques such as the profiling of SIGNAL programs [12].

All these features favor a helpful and confident context for the design activity. In the next, we concentrate on the design of component models. We model a bounded FIFO queue, usable for message exchanges between several entities. Besides this particular example which can be used as a communication component, we illustrate more generally a component-based approach. We also show how to verify properties on a component, and how to abstract it for future use. This brings out the main features of SIGNAL programming, and their benefits for a component-based design within a homogeneous formal framework.

4. Design of a safe FIFO queue

A FIFO queue, called `basic_FIFO` is first considered. This component will be enhanced later so as to derive a really "safe" FIFO queue on which properties will be checked.

Model of a basic FIFO queue. Informally, `basic_FIFO` works as follows:

- On a write request, the incoming message is inserted in the queue regardless of its size limit. When the queue was previously full, the oldest message is lost. The other messages are shifted forward, and the incoming is put in the queue.
- On a read request, there is an outgoing message whatever the queue status is. When it was previously empty, two situations are distinguished: if there is not yet any written message, an arbitrary message called *default message* is returned; else the outgoing message is the message that has been read last.

```
process safe_FIFO =
     { type message_type; integer fifo_size; message_type default_mess; }
     ( ? message_type mess_in; event get_mess;
       ! message_type mess_out; boolean OK_write, OK_read;
     )
  (| access_clock := mess_in ^+ get_mess                                  (2.a)
   | new_mess_in := mess_in when OK_write                                 (2.b)
   | mess_out ^= get_mess when OK_read                                    (2.c)
   | (mess_out, nbmess, OK_write, OK_read) :=
      basic_FIFO{ message_type, fifo_size, default_mess }
                    (new_mess_in, access_clock)                           (2.d)
   |)
 where
   use basic_FIFO;
   integer nbmess; message_type new_mess_in; event access_clock;
 end;%safe_FIFO%
```

mess_in	:	$\perp$	4	6	$\perp$	$\perp$	$\perp$	5	7	8	$\perp$	$\perp$	...
get_mess	:	*t*	$\perp$	$\perp$	*t*	*t*	*t*	$\perp$	$\perp$	$\perp$	*t*	*t*	...
mess_out	:	$\perp$	$\perp$	$\perp$	4	6	$\perp$	$\perp$	$\perp$	$\perp$	5	7	...
OK_write	:	*t*	*t*	*t*	*f*	*t*	*t*	*t*	*t*	*f*	*f*	*t*	...
OK_read	:	*f*	*f*	*t*	*t*	*t*	*f*	*f*	*t*	*t*	*t*	*t*	...

Figure 3. A model of safe_FIFO with an associated trace.

Furthermore, for simplicity we suppose that write/read requests never occur simultaneously.

4.1. SIGNAL specifications

Here, we concentrate on the SIGNAL description of the FIFO queue. We give a model for basic_FIFO, then we show how to specify another FIFO queue from the previous one.

The corresponding SIGNAL description (also termed *process model*) is given in **Figure 2**. Variables message_type, fifo_size and default_mess are parameters, which respectively denote the type of messages, the size limit of the queue, and the default message value. The input[5] signals mess_in and access_clock are respectively the incoming message (its presence denotes a write request), and the queue access clock (i.e. instants of read/write requests). The output[6] signals are mess_out, nbmess, OK_write and OK_read. They respectively represent the outgoing message, the current number of messages in the queue, and conditions for writing and reading.

Now, we can take a look at the meaning of the statements in the process body. Let us begin with the equation (1.b); it defines the local signal[7] prev_nbmess, which denotes the previous number of messages in the queue. This signal is used in (1.c) and (1.d), to define respectively when the queue can be "safely" written (the size limit is not reached), and read (there is at least one message received). This is the meaning of the signals OK_write and OK_read.

[5]Introduced by the symbol " ? ".

[6]Introduced by the symbol " ! ".

[7]Declared in the "**where**" section in the process model.

The statement (1.a) expresses how the current number of messages is calculated. That is, its previous value is incremented by one when there is a write request, and if the queue was not full; it is decremented by one when there is a read request, and if the queue was not empty; otherwise it remains unchanged. The equation (1.h) states that the value of nbmess changes whenever there is a request on the queue.

The equation (1.e) defines the message queue. The signal queue is an array of dimension fifo_size that contains the fifo_size latest values of mess_in (expressed by the window operator). The cell operator makes the signal queue available when access_clock is present (i.e. whenever there is a request).

Finally, (1.f) means that on a read request (i.e. at the clock ^mess_out), the outgoing message is either the previous if the FIFO is empty (defined in (1.g)), or the oldest message in the queue. In the resulting trace (in **Figure 2**), the type of the messages is integer, the size limit is 2, and the default message value is -1.

Henceforth, the basic_FIFO model can be used to describe other FIFO queues. This is the topic of the next paragraph.

Model of a safe FIFO queue. In the model depicted in **Figure 3**, the interface[8] is slightly different from that of basic_FIFO. Parameters are the same. A new input signal get_mess has been added. It denotes a read request. The signal nbmess which was previously an output of basic_FIFO, is now a local signal.

The statement (2.a) defines the access clock as the union of instants where read/write requests occur. Equations

[8]Parameters, inputs, and outputs.

(2.b) and (2.c) are in charge of ensuring a safe access to the queue in basic_FIFO. The process call in (2.d) has the local signal new_mess_in as input. This signal is defined only when basic_FIFO was not full, it is stated by (2.b). Similarly, (2.c) expresses that on a read request, a message is received only when basic_FIFO was not empty. In the trace in **Figure 3**, the same parameters as for basic_FIFO are considered.
The safe_FIFO component will serve later in the description of some communication protocol such as the LTTA protocol (Loosely Time-Triggered Architectutes) [4].

We observe that modularity and reusability are key features of the SIGNAL programming. They favor component-based designs. By constraining a given component, one can derive others. The most difficult task is the identification of suitable basic components. Moreover, the adaptability of components is very flexible. As a matter of fact, the SIGNAL process model enables "generic" components by parameterizing the interface (e.g. in the above models, the type of messages is a parameter). Finally, combined with the other characteristics of the language (e.g. possibility of non-deterministic specifications), richer descriptions are enabled.

4.2. Property verification

As mentioned earlier in the paper, a benefit of using SIGNAL for designs is the availability of formal verification tools. Two kinds of properties can be distinguished about SIGNAL programs [13]: *invariant* properties (e.g. a program exhibits no contradiction between clocks of involved signals) on the one hand, and *dynamical* properties (e.g. reachability, liveness) on the other hand.
The SIGNAL compiler itself addresses only the first one. For a given SIGNAL program, it checks the consistency of constraints between clocks of signals, and statically proves properties.
Dynamical properties are addressed by the model checker SIGALI [14], available within POLYCHRONY. SIGALI relies on the theory of polynomial dynamical systems. Roughly speaking, a SIGNAL program is abstracted into a system of polynomial equations[9] over $\mathbb{Z}/_{3\mathbb{Z}}$. This allows to encode all the possible status of a boolean signal: 1 for *true*, -1 for *false*, and 0 for $\perp$. For a non-boolean signal, only the fact that this signal is *present* (whatever its value is) or *absent* is encoded. So, the presence is denoted by 1, and the absence by 0. It must be noted that this "translation" *fully takes into account information about boolean variables (values and clocks), whereas for non-boolean signals, information on values are lost*. Therefore, it is important that a SIGNAL program that will be analyzed by SIGALI is specified as much as possible using boolean variables, since reasoning capabilities capture only synchronization and logic properties. In fact, people most often have to consider a boolean abstraction of programs with numerical properties. This is the main limitation[10] of SIGALI.
In the sequel, properties of interest concern first *safety*:

- (S_1): Write to the full queue never happens.
- (S_2): Read to the empty queue never happens.

Other desirable properties are for example the following *invariants*:

- (I_1): A message can always be written in the queue, when it is not full.
- (I_2): A message can always be read from the queue, when it is not empty.

To check these properties, we consider an abstraction of the process safe_FIFO. It can be obtained using the *state variables* (signals that are defined by *delay* or *memory* operators) that appear in the program. They feature the dynamics of the system defined by the process. Here, the state variables are nbmess, queue and prev_mess_out (defined in basic_FIFO).
On the other hand, we notice that properties of interest can be addressed by considering only the state variable nbmess. Indeed, the queue access conditions rely on this single signal. Therefore, in the abstraction, a state will be simply characterized by nbmess. Furthermore, since SIGALI does not address numerical properties, nbmess must be encoded with a boolean variable.

A n-FIFO queue can be represented by an automaton with $(n+1)$ states, where a state denotes the current number of messages in the queue. For the sake of simplicity, we consider a 2-FIFO queue since all possible relevant configurations can be addressed. So, the results remain valid for any bounded n-FIFO queue where $n > 2$. Moreover, it is assumed that messages in the queue are read in the same order they have been written (i.e. the FIFO queue satisfies the *sampling theorem* in the protocol for Loosely Time-Triggered Architectures [4]).

The automaton in **Figure 4** abstracts a 2-FIFO queue behavior. A state sk (represented by a circle) denotes the fact that the queue currently contains k messages. In other words, for any $k \in \{0, 1, 2\}$:

$$(nbmess = k \Rightarrow sk = true) \quad \wedge \\ (nbmess \neq k \Rightarrow sk = false)$$

The state $s0$ represents the initial state. Labels *in* and *out* are respectively write and read requests. Two special states (represented by rectangles) have also been

[9] In fact, a symbolic automaton.

[10] Some solutions [6] have been proposed to cope with the problem of numerical properties verification.

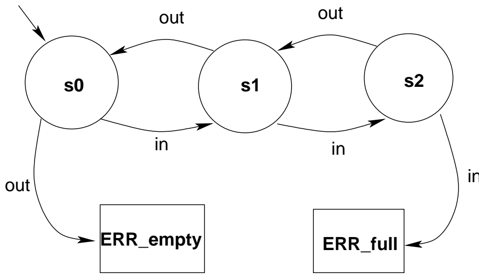

Figure 4. 2**-FIFO queue abstraction.**

```
Sigali;                                      (1)
read("safe_FIFO.z3z");                       (2)
read("Creat_SDP.lib");                       (3)
read("Verif_Determ.lib");                    (4)
Always(B_False(Err_full));     →   True      (5)
POSSIBLE(B_True(Err_full));    →   False     (6)
Reachable(Err_empty);          →   False     (7)
```

Figure 5. Script for safety properties checking.

added. They characterize "illegal" accesses to the queue: *ERR_empty* is reached on an attempt to read an empty queue, and *ERR_full* is reached when overwriting a full queue. They are also encoded by boolean variables.
Automata are very easy to specify in SIGNAL. To show how states can be defined, let us consider the following specification of $s0$:

```
(|s0 := (true when prev_s1 when (^mess_out))
         default (false when prev_s0 when
         (mess_in ^+ mess_out)) default
         prev_s0
 |prev_s0 := s0$1 init true
 |)
```

In the above statements, `prev_s1` represents the previous value of the state `s1`. All the other states are specified in a similar way. It follows the definitions of signals `OK_write` and `OK_read` below:

```
(| OK_write := false when (prev_err_full or
                prev_s2) default true
 | OK_read := false when (prev_err_empty or
               prev_s0) default true
 |)
```

The first equation means that a write request is not authorized when there are already two messages in the queue (`prev_s2` is *true*), or the queue has been overloaded previously (`prev_err_full` is *true*); otherwise it can be accepted. In a same way, the other statement specifies when a read request is legal.
The signals `s0`, `s1`, `s2`, `ERR_empty`, `ERR_full`, `OK_write` and `OK_read` are synchronized with `access_clock`.
To use SIGALI, a file[11] must be produced with the required input format.

Let us consider the script in **Figure 5**: SIGALI is first invoked (line `(1)`), then all the necessary files are loaded (`Creat_SDP.lib` and `Verif_Determ.lib` contain specific functions of SIGALI). The predicates on the lines `(5)` and `(6)` show that the state *Err_full* always remains *false*, and it is not possible that it becomes *true* (i.e. property S_1). In another way, the last statement shows that the state *Err_empty* is not reachable (i.e. property S_2).

Now for invariant properties[12] I_1 and I_2, we consider *observers*, represented by boolean state variables. We have to show that these variables always carry the value *true*. Let `inv1` and `inv2` denote respectively the observers for (I_1) and (I_2).

1. (I_1) is described as follows:

   - On a write request (denoted by the presence of `mess_in`), when the queue is either in $s0$ or $s1$; the signal `inv1` carries the value *true* if the message is actually written into the queue (i.e. `new_mess_in` is present), else `inv1` is *false*.
   - Otherwise, `inv1` keeps its previous value.

   The corresponding SIGNAL code is:

```
(| actual_write := true when(^new_mess_in)
                   default false
 | inv1 := actual_write when(z_s0 or z_s1)
           when(^mess_in) default z_inv1
 | z_inv1 := inv1 $ 1 init true
 |)
```

   The boolean `actual_write` denotes the fact that a message is actually put into the queue.

2. In a similar way, (I_2) is encoded by the following SIGNAL code:

```
(| actual_read := true when(^mess_out)
                  default false
 | inv2 := actual_read when(z_s1 or z_s2)
           when pull_mess default z_inv2
 | z_inv2 := inv2 $ 1 init true
 |)
```

Here also, all the new variables have the same clock as the signal `access_clock`. Then, the properties can be checked as shown in **Figure 6**. The component `safe_FIFO` can be embodied further in a communication

[11] This file has the extention **.z3z**, and is obtained by compiling the SIGNAL source file with the option **- z3z**.

[12] Expressing these invariant properties requires to refer to the dynamics.

```
Sigali;
read("safe_FIFO.z3z");
read("Creat_SDP.lib");
read("Verif_Determ.lib");
Always(B_True(inv1));          →   True
POSSIBLE(B_False(inv1));       →   False
Always(B_True(inv2));          →   True
POSSIBLE(B_False(inv2));       →   False
```

Figure 6. Script for invariant properties checking.

protocol, where new properties are verified (e.g. absence of deadlock during accesses by message writers and readers). The protocol itself will be further used within an application whose behavior can be analyzed, and so on. In that way, properties are incrementally checked and specifications are guaranteed to be correct.

Consistency checking and analysis of component models are essential to their dependability. Here, both models and properties are described using a unique formalism, the SIGNAL model, and adequate tools for verification and analysis are provided by the programming environment. This ensures a certain coherence in the design, contrary to approaches such as [15], where an implementation language (Java) and a formal specification language (labeled transition systems) are combined to implement systems.

Discussion. Essentially, two issues can be observed about the scalability of our approach to large systems.

The first concerns the correct distribution of the system functionalities on a given architecture. This is achieved by providing a synchronous model of the functionalities, on which one can perform verifications and analysis to make sure that requirements are met. In particular, one can check whether or not endo/isochrony properties [3] hold, for a safe deployment of the model on a distributed architecture.
The second issue proceeds in an incremental way. Instead of modeling the whole system through its functionalities, its sub-systems are specified. They can be analyzed separately, and "composed" using communication media (e.g. the safe FIFO described here), or protocols (e.g. the LTTA protocol [4]), defined also in the SIGNAL model. This composition must obviously guarantee some critical properties in the resulting system. For instance, there must be no loss of messages during information exchanges between sub-systems. This is addressed by the so-called *sampling theorem* in the LTTA protocol [4]. The SIGNAL model enables such analysis.

However, the main restriction of the approach lies in the fact that the synchronous modeling does not allow the description of unbounded resources. Typically, an unbounded FIFO queue cannot be completely modeled in the SIGNAL model. One may only define an associated abstraction, which does not provide all the necessary implementation details for an in-depth analysis purpose.
In embedded systems, resources are always limited, so the approach remains valid.

5. Conclusions

We have argued in this paper that the SIGNAL language favors an efficient approach to the design of embedded systems. Basically, a system is first specified in the SIGNAL model. Then, through formal transformations, another SIGNAL model is derived, which reflects the target architecture. These transformations proceed by a desynchronization of synchronous programs, based on the endo/isochrony properties [3]. The level of detail in which the architecture needs to be described may require specific mechanisms to achieve, for instance, communications, synchronizations, etc. Such mechanisms can be also specified and analyzed in the SIGNAL model, as we illustrated here for the modeling of a safe FIFO queue. We also have shown how properties are verified to guarantee the dependability of this FIFO queue for a further use in communication protocols (e.g. LTTA [4]). In the same way, the protocol itself can be analyzed, then may be used in a system which can be part of a larger system, where on every level of complexity we can perform our analysis.

We advocate a design methodology including high level specifications using the modularity and reusability features of the SIGNAL programming; formal verification and performance evaluation; automatic code generation. In such a context, the formal basis of SIGNAL is a key aspect for validation, contrarily to other approaches based on a formalism like UML whose formal foundations are not well-established. This is essential to a reliable design of safety critical systems.

A design of a real world avionics application using this approach is currently under study. The used components [8] have been defined from the specifications of the avionics standard ARINC 653. They include mechanisms for communication (e.g. *buffer*, *blackboard*), synchronization (e.g. *semaphore*), and execution (e.g. *processes* and associated *management services*), etc. This work is to be extented to applications from other safety critical domains like automotive or nuclear industries. In that case, an adaptation of the existing component models may be required to conform to the considered standards (e.g. OSEK for automotive). In this connection, a modeling of the real-time Java API using SIGNAL is currently studied. This should allow to access the available formal techniques and tools of POLYCHRONY for the analysis of real-time Java applications.

6. Acknowledgments

We thank Hervé Marchand for his worthful advices on the use of SIGALI.

References

[1] A. Benveniste. Safety critical embedded systems: the sacres approach. In *Formal techniques in Real-Time and Fault Tolerant Systems, FTRTFT'98 school, Lyngby, Denmark*, September 1998.

[2] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proceeding of the IEEE, vol. 79, No. 9*, pages 1270–1282, April 1991.

[3] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. In *Information and Computation, vol. 163*, pages 125–171, 2000.

[4] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Proc. of 2002 Conference on Embedded Software, EMSOFT'02, J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS vol 2491, Springer Verlag*, 2002.

[5] A. Benveniste, P. Le Guernic, and C. Jacquemot. Programming with events and relations: the SIGNAL language and its semantics. In *Science of Computer Programming, 16:103-149*, 1991.

[6] F. Besson, T. Jensen, and J. Talpin. Timed polyhedra analysis for synchronous languages. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99), LNCS volume 1664, Springer Verlag*, August 1999.

[7] P. Clements. A survey of architecture description languages. In *8th Int'l Workshop on Software Specifications and Design, Paderborn, Germany*, March, 1996.

[8] A. Gamatié and T. Gautier. Modeling of modular avionics architectures using the synchronous language SIGNAL. In *Proceedings of the Work In Progress session, 14th Euromicro Conference on Real Time Systems, ECRTS'02*, pages 25–28. Vienna, Austria, June 2002.

[9] T. Gautier and P. Le Guernic. Code generation in the sacres project. In *Proceedings of the Safety-critical Systems Symposium, SSS'99, Springer.* Huntingdon, UK, February 1999.

[10] D. Goshen-Meskin, V. Gafni, and M. Winokur. SAFEAIR: An integrated development environment and methodology. In *INCOSE 2001, Melbourne*, July 2001.

[11] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publications, 1993.

[12] A. Kountouris and P. Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems, IEE*, pages 6/1–6/9. HP Labs, Bristol, UK, February 1996.

[13] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of SIGNAL programs: Application to a power transformer station controller. In *Science of Computer Programming, 41(1)*, pages 85–104, August 2001.

[14] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. In *Discrete Event Dynamic System: Theory and Applications, 10(4)*, pages 325–346, October 2000.

[15] T. Nelson, D. Cowan, and P. Alencar. Supporting formal verification of crosscutting concerns. In *Third International Conference, REFLECTION 2001, Kyoto, Japan Reflection*, pages 271–285, 2001.

[16] Object Management Group. Omg unified modeling language specification version 1.4. In *(http://www.omg.org/technology/documents/formal/uml.htm)*, September 2001.