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.archives-ouvertes.fr/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 GAMA TIÉ, 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 INRIA1 (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-

1There is also an industrial version, SILDEX, implemented and commercialized by TNI-Valiosys (http://www.tni-valiosys.com).
ments 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 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 \(^c x\) (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.** \(y := f(x_1, \ldots, x_n)\), where 
  - \(y_t \neq \perp \iff x_1_t \neq \perp \iff \ldots \iff x_n_t \neq \perp\), and 
  - \(\forall t: y_t = f(x_1_t, \ldots, x_n_t)\).

- **Delay.** \(y := x \:$: 1 \ init \ y_0\), where 
  - \(x_t \neq \perp \iff y_t \neq \perp\), \(\forall t > 0: y_t = x_{t-1}, y_0 = y_0\).

- **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.** \(y := x_1 \ldots \ldots \ldots \ldots \perp + x_n\), where \(x_i\) is of event type, and denotes the set of instants where at least one signal \(x_i\) occurs.

- **Synchronizer.** \(x_1 \ldots \ldots \ldots \ldots \perp + \ldots + x_n\), where \(x_i\) is of event type, and denotes the set of instants where at least one signal \(x_i\) occurs.

- **Sliding window.** \(y := x \ window \ n \ init \ y_0\), where 
  - \(\forall t \geq 0, i \in 0..n-1: (t + i \geq n) \Rightarrow (Y_t[i] = X_{t-n+i+1})\) \(\land\) 
  - \(i \leq t < n \Rightarrow (Y_t[i] = y_0[t-n+i+2])\)

- **Memory.** \(y := x \ cell \ b \ init \ y_0\), defined as: 
  - \(\text{def} \ y := x \ default \ (y$1 \ init \ y_0)\) 
  - \(y \ := \ x \ ^+ \ (\text{when} \ b)\)

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

\[
\begin{align*}
| \ y := & x \ window \ 3 \ init \ [-1,0] \\
| \ z := & x \ cell \ b \ init \ 0 \ \\
\end{align*}
\]

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

\[
\begin{align*}
| x : & \perp 1 2 \ perp 3 \ldots \\
| b : & t \perp f t f \ldots \\
| y : & \perp [-1,0,1] 0 1 2 \perp \perp [1,2,3] \ldots \\
| z : & 0 1 2 2 \perp 3 \ldots \\
\end{align*}
\]

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.

\[\text{Similary, intersection and difference of clocks are defined.}\]
The target architecture is composed of a set of possibly heterogeneous execution components (processors, microcontrollers...).

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

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.
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.a); it defines the local signal 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.

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

---

\(^5\)Introduced by the symbol " ? ".

\(^6\)Introduced by the symbol " ! ".

\(^7\)Declared in the “where” section in the process model.

\(^8\)Parameters, inputs, and outputs.
(2.b) and (2.c) are in charge of ensuring a safe access to the queue in \texttt{basic_FIFO}. The process call in (2.d) has the local signal \texttt{new_mess_in} as input. This signal is defined only when \texttt{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 \texttt{basic_FIFO} was not empty. In the trace in Figure 3, the same parameters as for \texttt{basic_FIFO} are considered.

The \texttt{safe_FIFO} component will serve later in the description of some communication protocol such as the LTTA protocol (Loosely Time-Triggered Architectures) [4].

We observe that modularity and reusability are key features of the \texttt{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 \texttt{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 \texttt{SIGNAL} for designs is the availability of formal verification tools. Two kinds of properties can be distinguished about \texttt{SIGNAL} programs [13]: \textit{invariant} properties (e.g. a program exhibits no contradiction between clocks of involved signals) on the one hand, and \textit{dynamical} properties (e.g. reachability, liveness) on the other hand.

The \texttt{SIGNAL} compiler itself addresses only the first one. For a given \texttt{SIGNAL} program, it checks the consistency of constraints between clocks of signals, and statically proves properties.

Dynamical properties are addressed by the model checker \texttt{SIGALI} [14], available within \texttt{POLYCHRONY}. \texttt{SIGALI} relies on the theory of polynomial dynamical systems. Roughly speaking, a \texttt{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 \texttt{true}, \(-1\) for \texttt{false}, and \(0\) for \texttt{absent}. For a non-boolean signal, only the fact that this signal is \texttt{present} (whatever its value is) or \texttt{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 \texttt{SIGNAL} program that will be analyzed by \texttt{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 \texttt{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 \texttt{safe_FIFO}. It can be obtained using the \textit{state variables} (signals that are defined by \texttt{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 \texttt{nbmess}, \texttt{queue} and \texttt{prev_mess_out} (defined in \texttt{basic_FIFO}).

On the other hand, we notice that properties of interest can be addressed by considering only the state variable \texttt{nbmess}. Indeed, the queue access conditions rely on this single signal. Therefore, in the abstraction, a state will be simply characterized by \texttt{nbmess}. Furthermore, since \texttt{SIGALI} does not address numerical properties, \texttt{nbmess} must be encoded with a boolean variable.

A \(n\)-\texttt{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-\texttt{FIFO} queue since all possible relevant configurations can be addressed. So, the results remain valid for any bounded \(n\)-\texttt{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 \texttt{FIFO} queue satisfies the \textit{sampling theorem} in the protocol for Loosely Time-Triggered Architectures [4]).

The automaton in Figure 4 abstracts a 2-\texttt{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 \(k \in \{0,1,2\}\):

\[
\text{\tt (nbmess} = k \Rightarrow \text{sk} = \text{true}) \land \\
\text{\tt (nbmess} \neq k \Rightarrow \text{sk} = \text{false})
\]

The state \texttt{s0} represents the initial state. Labels \texttt{in} and \texttt{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.
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:

\[
\begin{align*}
|s0 := (true \text{ when prev_s1 when } (\lnot \text{ mess_out})) \\
\text{default } (false \text{ when prev_s0 when } (\text{ mess_in } \lor \text{ mess_out})) \text{ default } prev_s0 \\
|prev_s0 := s0 \lor 1 \text{ init true }
\end{align*}
\]

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:

\[
\begin{align*}
|OK_write := false \text{ when } (prev_err_full or prev_s2) \text{ default true} \\
|OK_read := false \text{ when } (prev_err_empty or prev_s0) \text{ default true}
\end{align*}
\]

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 SIGNAL, a file\(^{11}\) must be produced with the required input format.

Let us consider the script in Figure 5: SIGNAL is first invoked (line (1)), then all the necessary files are loaded (Creat_SDP.lib and Verif_Determ.lib contain specific functions of SIGNAL). 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 S1). In another way, the last statement shows that the state Err_empty is not reachable (i.e. property S2).

Now for invariant properties\(^{12}\) I1 and I2, 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 (I1) and (I2).

1. \((I1)\) 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:

\[
\begin{align*}
| \text{actual_write} := \true \text{ when(} ^{\new \text{ mess_in}} | \\
| \text{default false} | \\
| \text{inv1} := \text{actual_write when(z_s0 or z_s1)} | \\
| \text{when(} ^{\text{ mess_in}} \text{ default z_inv1} | \\
| \text{z_inv1} := \text{inv1 $ 1 init true}
\end{align*}
\]

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

2. In a similar way, \((I2)\) is encoded by the following SIGNAL code:

\[
\begin{align*}
| \text{actual_read} := \true \text{ when(} ^{\text{ mess_out}} | \\
| \text{default false} | \\
| \text{inv2} := \text{actual_read when(z_s1 or z_s2)} | \\
| \text{when(} ^{\text{ pull_mess default z_inv2}} | \\
| \text{z_inv2} := \text{inv2 $ 1 init true}
\end{align*}
\]

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 extension .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.

---

Figure 4. 2-FIFO queue abstraction.

Figure 5. Script for safety properties checking.
This is addressed by the so-called sages during information exchanges between sub-systems. 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.

Figure 6. Script for invariant properties checking.

\begin{verbatim}
Sigali;
read("safe_FIFO.z3s");
read("Creat_SDp.lib");
read("Verif_Determ.lib");
POSSIBLE(B_False(inv1)); \rightarrow True
POSSIBLE(B_False(inv2)); \rightarrow False
Always(B_True(inv1)); \rightarrow True
Always(B_True(inv2)); \rightarrow True
POSSIBLE(B_False(inv2)); \rightarrow False
\end{verbatim}
6. Acknowledgments

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

References


