Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 146 (2006) 105–131
www.elsevier.com/locate/entcs

A Verification Approach for GALS Integration of Synchronous Components

F. Doucet, M. Menarini, I. H. Krüger and R. Gupta[1]

Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA

J.-P. Talpin[2]

IRISA/INRIA
Rennes, France

Abstract

Starting with modules described in Signal synchronous programming language, we present an approach to verification of GALS systems. Since asynchronous parts of a GALS system can not be described in Signal, we use a mixture of synchronous descriptions in Signal and asynchronous descriptions in Promela. Promela is the input language to the SPIN asynchronous model checker. This allows us to achieve globally asynchronous composition (Promela) of locally synchronous components (Signal). Here we present three key results: first, we present a translation from Signal modules to Promela processes and prove their equivalence. Second, we present a technique to abstract a communication bus designed for GALS, the Loosely Time-Triggered Architecture (LTTA) bus, to a finite FIFO channel. The benefit of this abstraction is improved scalability for model checking larger specifications using SPIN. Third, we prove the trace equivalence of the model of the GALS system in Promela and a hardware implementation of it. This allows the verification of GALS systems based on the Promela model. We then use our technique to verify a central locking system for automobiles built on a GALS architecture using the LTTA.

Keywords: Specification, Verification, Synchronous Languages, Model Checking, Signal, Spin, GALS.

[1] Email: {fdoucet,mmenarini,ikrueger,rgupta}@ucsd.edu
[2] Email: Jean-Pierre.Talpin@irisa.fr

doi:10.1016/j.entcs.2005.05.038

1 Introduction

Designing complex system-on-chip devices containing multiple cores and multiple clock domains presents challenging modeling and verification problems. Traditionally microchips are developed using synchronous languages that help reduce the complexity of the design verification tasks. However, the physical characteristics of deep sub-micron technologies make it very difficult to have predictable communication delays between computational modules. Therefore, it is difficult to achieve timing closure for the deployement of synchronous systems using those technologies. Globally asynchronous/locally synchronous (GALS) architectures are attractive for a number of reasons for use in on-chip system designs [7]. They provide asynchronous communication mechanisms that are not affected by communication latency. However, modeling and verification of such systems can be difficult due to the diversity of the component-level verification used.

In this paper, we address the problem of verification of GALS deployment of synchronous intellectual property (IP) modules specified using the Signal synchronous language [4]. Synchronous languages provide a way to specify systems as transition functions, where timing constraints can be put on when these transitions are taken. We use the Spin model checker [8] to verify synchronous modules in an asynchronous environment. Spin has been successfully used to verify asynchronous software systems. Our decision to use Spin instead of other model checkers is based on the two capabilities it provides: (1) efficient model checking algorithms for asynchronous semantics using partial order reduction (2) the ability to inline calls to external C functions in a Spin model, and treat them as black boxes. The asynchronous semantics of Spin enables us to efficiently model the GALS integration environment of several embedded synchronous components. We take advantage of partial order reduction to reduce the state space needed for the verification of the GALS model. Furthermore, for quick and efficient path to verification, we can take the output of the Signal compiler, a C function that simulates the synchronous reaction, and call it into a Promela process (Promela is the input language to Spin).

The contribution of this paper is a verification approach for synchronous component integration in a GALS architecture. First, we present a translation from a Signal module to a Promela process, and show the correctness of our translation by proving trace equivalence. Next, we describe the Loosely Time-Triggered Architecture (LTTA) bus used for communication in GALS architectures. We abstract this bus to a FIFO channel and use model checking to prove their equivalence. The abstraction increases the size of GALS architectures we are able to verify using Spin. Using the abstraction, we can

build and verify a Promela model of a GALS architecture that includes the synchronous components. Finally, we prove that our GALS model in Promela is trace equivalent to the hardware implementation of a GALS system using the LTTA bus. The combined use of Signal for specifications and of Spin for verification offers an effective and efficient combination for the verification of GALS architectures. Our results show that there is a possible automatic path that can be used to verify GALS deployment of synchronous specification.

This paper is organized as follows. In Section 2 we review the related work. In Section 3 we present the synchronous model of computation, as well as the Signal language and its semantics. In Section 4 we present our verification strategy for verifying synchronous components using Spin. In Section 5 we show our approach to verifying GALS architectures. In Section 6, we present an example of an automobile central locking system, and show how using our method it is possible to verify a central property of the system. We describe model checking performance statistics for this example. We then discuss future work and conclude in Section 7.

2 Related Work

A central challenge in designing system-on-chip using IP blocks is the problem of communication latency [11]. This fact makes the use of GALS architectures an appealing compromise between purely synchronous and asynchronous implementations. Many different architectures have been proposed over time to connect different synchronous components into a GALS system [12]. Carloni et al. [6] have proposed a latency-insensitive architecture which separates communication and computation. By means of a particular protocol, various synchronous components wait for all the data to be present on their input before executing any operation. In this architecture, problems due to the timing requirements of the physical clock are solved by logic blocks called relay stations. Those blocks contain latches, which are inserted on long communication lines that cross clock domains. A component that is insensitive to latency is called a patient process. If a process is not patient, it is possible to arbitrarily *stop its clock* by using a wrapper that waits until all data values are present, and latch them if necessary. Using this approach, component interactions are separated from computation and the system can be described using a synchronous language. However, as pointed out in [13], the latency-insensitive approach introduced by Carloni is limited to a single clock. If multiple systems with different clocks are interconnected, some delay must be inserted to slow down the faster ones to meet the slowest subsystem. This solution has potential for improvement because it slows down all the fast modules.

Ramesh et al. [10] present a tool set for modeling and verification of GALS architectures in the context of system-on-chip architectures. They use the Communicative Reactive Processes (CRP) language, a dialect of Esterel [5], and verify the architecture using Spin. To take advantage of the asynchronous model checking, they use a direct translation of CRP to Promela.

In this work, we propose an approach that builds on the work of system modeling using the Signal synchronous language [2]. We also use the Spin model checker, but we make use of a bus architecture abstraction to improve the scalability of the verification. We leverage the work of Carloni by using the wrapper and the relay stations proposed in the paper [6] for our equivalence between hardware implementation and Promela model.

3 The Synchronous Model of Computation

The synchronous model of computation implements the synchrony hypothesis [2] [5], which states that all components in a system synchronously (1) sample inputs, (2) fire, and (3) write outputs. All the reactions are instantaneous and, when observing the system, one sees the inputs at the same time as the outputs. In that sense, the synchrony hypothesis eases conceptualization because one does not have to think about all the possible process interleavings, potentially reducing the system state space. Note that, in the synchrony hypothesis, an observer cannot observe the causality relations between inputs and outputs. Examples of systems that can be specified efficiently in this paradigm are data-flow controllers used in embedded real-time applications.

Signal [4] is a language that implements the synchrony hypothesis. It is used for specification of data-flow equations over signals and the synchronous composition of these equations into a reactive system.

3.1 Tagged Model

The Signal semantics uses a tagged model, which is equivalent to the trace model, augmented with time annotations. In the tagged model, a signal is denoted by a sequence of tagged valuations (t, v) where v is a value and t is the symbolic time at which this value is sampled. This is a sequence where each event is tagged with a measure of the time at which it is recorded. The tag sequence $t_{1,..,n}$ of a signal s is called its clock, denoted by s_τ. For a signal s, the flow of values is sampled according to clock s_τ.

Example 3.1 *Figure 1 depicts a behavior b over three signals named x, y and z. The two frames depict two different timing domains. Signal x and y belong to the same timing domain: x is a down-sampling of y where its events are*

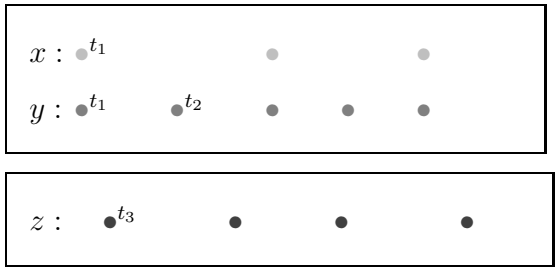

Fig. 1. Example of a multi-clocked behavior: the boxes represent two unrelated timing domains

synchronous to odd occurrences of events along y, sharing those specific tags. The signal z belongs to a different timing domain. Its tags are not ordered with respect to the clock of x, e.g. $x_\tau \not\leq z_\tau$ *and* $z_\tau \not\leq x_\tau$ *nor to the clocks of y.*

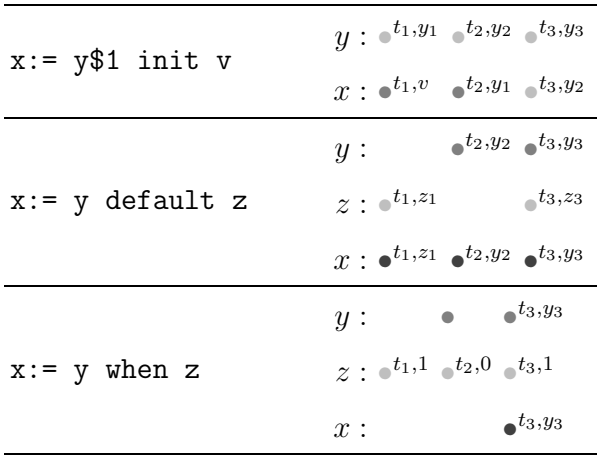

Fig. 2. Behaviors of single Signal equations

Let us now illustrate the behavior of a subset of the basic Signal equations. Figure 2 depicts the behavior of statements that are commonly found in specifications encoded with Signal. The first one is `x:= y$1 init v`, which initially defines `x` to v and subsequently by the previous value of `y`. The equation `x:= y default z` defines `x` by `y` when `y` is present and by `z` otherwise. Equation `x:= y when z` defines `x` by `y` when `z` is true. As one can see, the current value of the signals' clocks defines if the reaction can fire or not.

3.2 Transition System

We now define the semantics of the Signal language using a transition system.

Definition 1 (Transition System) *Let V be a set of variables over domain D. We define a transition system M over V to be a tuple $\langle S, T, s_0 \rangle$ where*

$S : V \to D$ is a set of states, $T \subseteq S \times S$ a transition relation and s_0 an initial state.

Each state represents a possible valuation of the variables. Two states $s \in S$ and $s' \in S$ are the same if the variables have the same exact values $s(V) = s'(V)$.

Definition 2 (Run) *Let M be a transition system. We define a run of a transition system to be a finite or infinite sequence of states $\pi = s_0 s_1 ....$*

A run is a sequence of states; a trace is a sequence of observations on a subset of the variables. These variables can represent input and output ports, while the internal variables would be hidden from an observer.

Definition 3 (Language) *Let M be a transition system. We define the language of M, denoted $\mathcal{L}(M)$, to be the set of all possible runs of M.*

3.3 Signal Semantics

We are now ready to express the semantics of the Signal language and its tagged model as a transition system. A Signal process is the conjunction of a set of basic Signal statements, listed Table 1 along with their interpretation semantics. Each variable v in the Signal specification is represented by two variables in the transition system: (1) variable v for holding the data, ranging over the same domain as in the specification, and (2) variable v_τ for the clock, ranging over a binary domain to indicate presence or absence of a value in v. In a run, we denote the k-th value in the sequence by v_k and the k-th clock by $v_{\tau k}$.

The statements in Table 1 are divided into three categories. Monochronous ("single-clocked") equations require all input and output signals to be present at the same time. Polychronous or multi-clocked equations do not require all signals to be present; they operate depending on which signals are currently present. The clock relations establish the timing constraints on signals; they define the presence and absence of signals.

3.3.1 Monochronous Operations

The synchrony hypothesis assumes that, for a reaction, all inputs and outputs are present at the same time and the reaction computes instantaneously. In Signal, monochronous operations implement the synchrony hypothesis. All arithmetic operations are monochronous. The statement $X_{\tau k} \leftrightarrow Y_{\tau k} \leftrightarrow Z_{\tau k}, Z_k := op(X_k, Y_k)$ expresses that at cycle k, by double implication, if one of the inputs or outputs is present, then all of them are present. This means that the clocks of the input and the output signals need to be the same, denoted

Table 1
Primitive constructs in the Signal language

Name	*Syntax*	*Interpretation Semantics*
Monochronous operations		
Arithmetic	`Z:= X op Y`	$X_{\tau k} \leftrightarrow Y_{\tau k} \leftrightarrow Z_{\tau k}, Z_k := op(X_k, Y_k)$ $X_\tau = Y_\tau = Z_\tau$
Delay/register	`Z:= X$1`	$X_{\tau k} \leftrightarrow Z_{\tau k}, Z_k := X_{k-1}$ $X_\tau = Z_\tau$
Polychronous operations		
Sampling	`Z:= U when B`	$B_{\tau k} \wedge B_k \wedge U_{\tau k} \leftrightarrow Z_{\tau k}, Z_k := U_k$
Choice	`Z:= U default V`	$(U_{\tau k} \leftrightarrow Z_{\tau k}, Z_k := U_k) \vee$ $(\neg U_{\tau k} \vee V_{\tau k} \leftrightarrow Z_{\tau k}, Z_k := V_k)$
Synchronous composition	`P | Q`	see Sub-Section 3.3.4
Clock relations		
Clock extraction	`Z:= ^X`	$Z_k \leftrightarrow X_{\tau k}$
Clock equality	`X ^= Y`	$X_\tau = Y_\tau$
Upper bound clock	`Z ^= X ^+ Y`	$X_{\tau k} \vee \wedge Y_{\tau k} \leftrightarrow Z_{\tau k}$
Lower bound clock	`Z ^= X ^* Y`	$X_{\tau k} \wedge Y_{\tau k} \leftrightarrow Z_{\tau k}$

by $X_\tau = Y_\tau = Z_\tau$. When the clock predicate is satisfied and all input values are present and known, the computation of Z_k can fire. The computation is instantaneous and the value of the output is present at cycle k and can be used in other synchronous reactions. The delay operation, also monochronous, assigns the previous value of X, X_{k-1} to Y_k. Again, the clocks are the same, if an input is present then an output is present, but in this case the values are delayed by one sampling instant. This is analogous to a synchronous register.

3.3.2 Polychronous Operations

Polychronous operations define relations on signal flows that are not synchronous. Clock presence or absence can be used to build control-flow in a specification where the clocks are used to describe the timing of specific events. In some sense, the clocks are guarding the possible polychronous transitions. The first kind of polychronous statement is the sampling operation. A signal Z samples a signal U when a sampling condition, represented by signal B, is both present and true. Otherwise, on signal Z at instant k, there is no value Z_k available and its clock, $Z_{\tau k}$ is false. When the sampling condition is false, we do not care about the presence or absence of U. Again, this is a

double implication. So if somewhere in the specification Z_k is required, then the input needs to be present and the sampling condition will be true. The polychronous second operation is the choice operation. At an instant k, a signal Z is assigned a value U_k if U_k is present; otherwise a value V_k if U_k is absent and V_k is present. Notice that U has priority over V. This operation is considered polychronous because an output can be present when only one of the two inputs is present.

3.3.3 Clock Relations

Clock relations are used to define the control flow through a Signal specification. The encoding of conditional statements ("if-then-else") into clock relations results in a "clock tree", a hierarchical organization of the clocks that control the branches of the nested conditionals. In the next two paragraphs we explain the notion of clock tree as well as the clock relation statements used to encode conditionals in more detail.

The clock tree expresses the relationships between the clocks of all signals in a program. The root of the tree ticks every time some clock is present; this is the most fine-grained clock. As one goes down the tree, each node is present only for a subset of the root ticks. The tree is built such that the control flow of a specification is described by the presence or the absence of control signals: the nodes in the tree are the control signals. For instance, an "if-then-else" will be represented by two complementary clocks: one will be present only when it is time to take the "then-condition" and the other will be present only when it is time to take the "else-condition". From the root, these two nodes are on different branches – meaning that both clocks will never be present at the same time.

Now, let us describe the Signal statements in Table 1 for clock relations. The statement for clock extraction, `Z:=^X`, is used to specify that signal Z describes the clock of X. This statement enables a designer to explicitly extract the clock of a signal X so that it can be used in conditional statements. For instance, one can attach some action on the presence ($Z == 1$) or absence ($Z == 0$) of X. The statement for clock equality, `X ^= Y` is used to specify a synchrony constraint on two signals. This means that both signals X and Y are exactly present or absent at the same time.

The "upper bound clock" and "lower bound clock" statements are used to define constraints on the clock tree. The first one, `Z ^= X ^+ Y`, defines the clock of signal Z to be the upper bound of the clocks of signals X and Y. This effectively is the union of two clocks, resulting in Z being present every time one of the two inputs X or Y is present. This statement is the timing behavior of the choice operation. The second statement, `Z ^= X ^* Y` defines

the clock of signal Z to be the lower bound of the clocks of X and Y. This effectively is the intersection of the two clocks, resulting in a signal Z present only when both signals X and Y are present. This timing behavior is close to the timing behavior of the sampling operation, except that the sampling statement requires the second input to have a true value.

3.3.4 Synchronous Composition

The synchronous composition operation `P | Q` is a polychronous operation in the sense that it can be multi-clocked. In Signal, each statement is a concurrent process; this means it is a small transition system. The behavior of a Signal specification is a transition system which is the synchronous composition of the transition systems of its components. To translate a specification, every statement is translated and then added in a conjunction that forms the total system behavior. This is effectively the synchronous composition of all statements. The synchronous composition of all Signal statements is obtained inductively and the result is the transition system of the Signal specification.

Definition 4 (Synchronous Composition) *Let* $M_1 = \langle S_1, T_1, s_{1_0} \rangle$ *and* $M_2 = \langle S_2, T_2, s_{2_0} \rangle$ *be transition systems over a set of variables* V. *We define their synchronous composition as being a tuple* $\langle S_\otimes, T_\otimes, s_{\otimes_0} \rangle$ *where* $S_\otimes = S_1 \times S_2$, $T_\otimes = \{(s_1, s_2) \rightarrow (s_1', s_2') \mid \exists (s_1, s_1') \in T_1 \wedge \exists (s_2, s_2') \in T_2\}$, *and* $s_{\otimes_0} = (s_{1_0}, s_{2_0})$.

In the composed system, to have some resulting behavior, the clocking constraints and the data values shared by P and Q have to agree. This means that if the two transition systems share variables, composition can eliminate many possible behaviors. Considering that clocks are variables, one has to consider the effects of composition for the temporal behavior. The presence or absence of a signal is defined by the relation of its clock with the clock of other signals in the system. When this relation is a function, the clock can be calculated (e.g. starting from the status and value of input signals). This calculation is used to build a control-flow graph and to generate sequential code [1]. The constraints are specified using these clock relations.

3.4 Translation to Transition System

We now consider the translation of a Signal specification to a transition system. Recall that the control flow through the Signal program is encoded as a clock tree. This clock tree defines a "firing schedule" for the transitions of the transition system we create. The firing schedule is the hierarchical composition of all clock relations in the system. Every time a synchronous reaction is triggered, the clock tree is evaluated to find which statements have to fire.

In a specification, each synchronous module has its own local clock tree, with its root defining the most general firing condition for the module. Now, if we take a module and compose it with another module, it is possible that the first module has to stay idle while the second module fires. The standard example, again, is the conditional statement, encoded as the synchronous composition of two synchronous "modules" – one for each branch of the conditional. Precisely one of these modules can fire – the branch that is taken in the execution of the conditional; the other module representing the other branch has to idle since its local root node in not in the branch.

To enable synchronous modules to be idle, we extend the semantic model to include stuttering steps. A stuttering step is represented by $\perp$ and presents itself to an observer as a reaction where all signals are absent. To accommodate this we need to add stuttering transitions in the transition system that set all the clock variables to zero – meaning that there is no input and no output to a reaction.

Definition 5 (Stuttering Step) *Let M be a synchronous transition system, and $clk(M)$ be the set of clock variables in M. We define a stuttering state as being a state where all variables in $clk(M)$ are zero. A stuttering step is a transition entering a stuttering state.*

Therefore, our transition system will allow stuttering runs, where a module can take an arbitrary number of stuttering steps.

Definition 6 (Stuttering Run) *Let M be a transition system. A stuttering run for a M is a run which allows an arbitrary number of stuttering steps to be taken between two states.*

We define each stuttering run $\pi_s = s_0, \perp, \perp, \perp, \perp, s_1, \perp, ..., \perp, s_2, ...$ to be equivalent to the run $\pi = s_0, s_1, s_2, ...$ obtained from π_s by removing all stuttering ($\perp$) states.

We now describe how each equation of Table 1 can be translated to a transition system. Figure 3 shows, for each Signal statement, an automaton describing the operation along with its clock constraints. Consider first the arithmetic operation `Z:= X op Y`. The corresponding automaton has two states: one for firing the reaction and one where the reaction is in a stuttering state. The equations in Table 1 ensure that the firing state can be entered only when all clocks are present, and that the stuttering state can be entered only when all clocks are absent. These constraints express the assumption that, for arithmetic operations, the only allowable behaviors for the environment are the monochronous behaviors, with respect to signals X, Y and Z. In the translation of the delay operation, the automaton also restricts the allowable behaviors for the environment to monochronous behaviors. For polychronous

operations, the automaton also expresses timing assumptions on the behavior of the environment, although less strict than the monochronous behaviors. There are two possible firing states and one stuttering state.

One can note that for the delay operation there is a causality relation inside the firing state. It is necessary to execute the assignment of the last value of X to the output Z, before registering the current value of X; else there would be loss of value to be stored. Since the reaction is instantaneous, in the figure we show the dependency graphically as a set of internal steps to be taken inside the firing state. Taking these internal steps does not change which clock guards are enabled for the next transition.

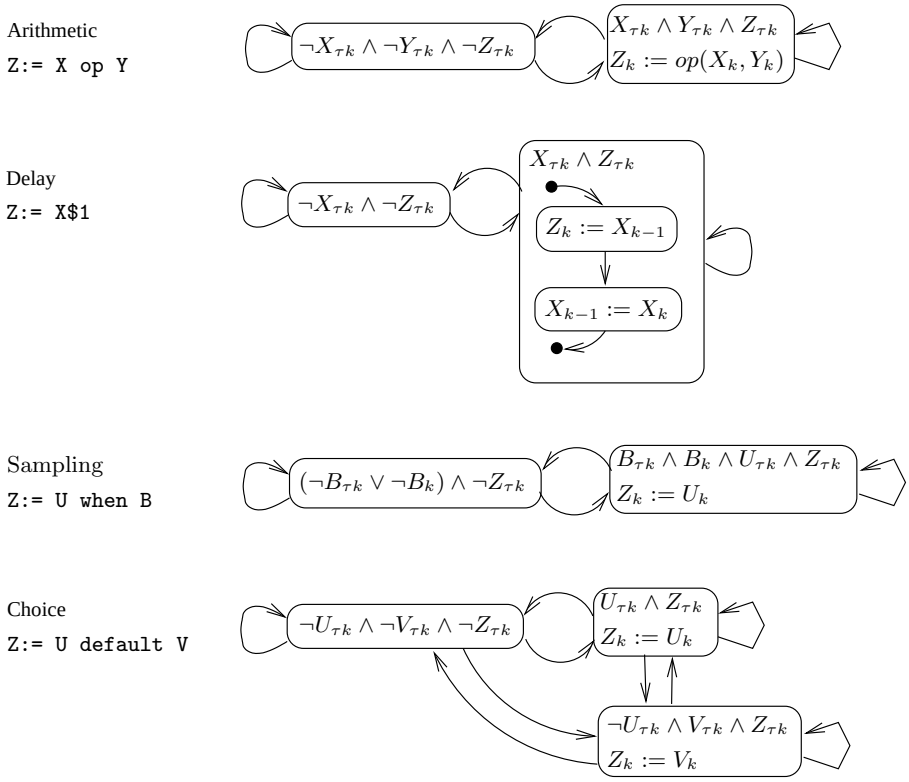

Fig. 3. Translation of Signal statements to transition systems. The automata transition only on assertion of the clock constraints, which describe the environment assumptions for correct monochronous or polychronous behavior. To be correct the delay operation requires a specific statement ordering.

4 Verification Strategy for Individual Components

In this section, we explain how to verify a single synchronous component in an asynchronous environment. To this end, we use the Spin model checker to

verify the correctness of the component. We first describe Spin; then we outline how to integrate a synchronous component in its environment. We discuss the issue of interfacing a synchronous component to an asynchronous environment, and then address the question of whether the model in the asynchronous environment is equivalent to the original Signal specification.

4.1 The Spin Model Checker

The Spin model checker [8] is used to verify distributed software systems. Spin models are described using the Promela language. In essence, Spin specifications consist of a set of sequential processes communicating by means of channels (or, as special cases, variables). Channels act as finite FIFO buffers. The body of a process is a sequence of guarded actions. Reading from/writing to a channel, assigning values to/reading values from a variable are examples of actions. Guards are predicates on the state space of the process, as well as synchronization predicates. A process can block when reading from an empty channel or writing to a channel that is full, but shared variable access does not block processes.

The Spin compiler translates each process to a transition system, and composes every process into the model to be checked. The processes that are not blocked can be executed in any arbitrary order. Each statement in a process can be interleaved with any other statement from any other process. Inside a process, specific statement sequences can be enforced to be atomic, meaning that they will not be interleaved with statements of other processes.

To reduce the state space explored by the model checker, Spin offers to perform a partial order reduction [8] to remove the interleavings that are observationally the same for an observational property. This means that if the order of execution of asynchronous processes is not important for a certain kind of property, the model checker will only consider one of these executions and prune all the other equivalent executions.

4.2 Translation of Signal Module to Promela Process

In this subsection we describe how we convert a Signal module into a Promela process.

4.2.1 Generation of Simulation Model of Synchronous Reaction

For a Signal module, the Signal compiler converts all the Signal statements of the module into transition systems, forms their synchronous product, and generates a C function that can simulate the synchronous reaction. The C

function reads the input signals, computes the reaction and writes values to output signals, all in accordance to the synchrony hypothesis, and equivalent to executing a single step of the synchronous transition system of the Signal module.

As described in Section 3, Signal modules often define assumptions about the environment through statements about the clocks of interface signals. In the C function generated by the Signal compiler, there is some code to check if the clock assumptions are satisfied. The return value of the C function is a Boolean value indicating if the reaction has fired successfully or not – meaning the assumptions were satisfied or not.

4.2.2 Translation Template

To verify a single Signal module, we wrap the C function generated by the Signal compiler inside a Promela process. The process generates values for the environment signals and fires the reaction by calling the C function. When integrating the C function into the Promela process, we know if the assumptions are met by firing the C function and checking the return value. If the clock constraints are not met, it will reset all signals to introduce a stuttering step. This way, when illegal values are generated on environment signals, the process executes a stuttering step and one does not observe any change on the interface signals.

```
1: proctype some_signal_module() {
2:   loop:
3:      atomic {
4:              /* 1. input generation  */
5:              /* 2. reaction:
                        call c code,
                        introduce stuttering step if required */
6:              /* 3. collect output */
7:      }
8:      goto loop;
9: }
```

Fig. 4. Translation template to wrap a Signal module inside a Spin process

Figure 4 shows the template we use to wrap the compiler-generated C function inside a Promela process. A synchronous Signal model is converted to a Promela *proctype*. The *loop* of line 2 coupled with the *goto* in line 8 is used to repeat the synchronous reaction forever. The synchronous reaction is emulated inside to *atomic* block (lines 3 - 7). The synchrony hypothesis, in fact, requires inputs and outputs to be observed at the same time. There is no causal relationship between input and output generation. We use an atomic step to reproduce this behavior. In fact, we will observe the behavior of the component using the never claim construct of Promela. This is a Büchi

automaton synchronously composed to the rest of the system; it takes a transition each time the rest of the system takes one. The use of an atomic step makes its content to look like a single step. Therefore, the never claim cannot observe what happens inside it, but just the variable configuration at the end of the atomic step. Lines 4, 5 and 6 of Figure 4 simulate the synchronous reaction. Line 4 generates all possible input values. Line 5 calls the C function generated by the Signal compiler and if it returns true (meaning that the clock conditions where satisfied) sets the output signals to the correct values; if it returns false all signals are reset to generate a stuttering step. Finally, when some observed propery is violated, line 6 prints the signal configuration to generate an error trace.

4.2.3 Semantics

We now describe the semantics of the Promela process using a transition system. Spin will add some variables to the transition system to enforce the semantics of the Promela progam. For instance, each `proctype` will have a process identifier associated with it. Moreover, a global variable called *exclusive* will be added to enforce the atomic step semantic.

The `proctype` delimits the Promela process to be translated into an automaton $M = \langle V, T, s_0 \rangle$. In the translation, an explicit program counter (pc), a unique process identifier (pid), and a global variable named "*exclusive*" are added to the variable set $\{pc, pid, exclusive\} \in V$. The "*exclusive*" variable is used to guard each transition with the following condition: "$exclusive == pid$" or "$exclusive == 0$".

The atomic block, starting on line 3 of Figure 4, is translated to a transition where *exclusive* is assigned to *pid*. This way, no other process can be interleaved with the atomic section. This is used to model that the synchronous reaction is instantaneous.

On line 4, the input generation is implemented using non-deterministic transitions. This builds a model of the environment, to close the system for model checking. The model of the environment can generate any possible input combination for the reaction.

On line 5, the reaction is invoked by calling the C function obtained from the Signal compiler. If the function returns false, we need to reset the input and output values to produce a stuttering step. This is because the synchronous process assumptions about the environment were not met by the input generation step. If the function returns true, it means that the reaction fired correctly.

On line 6 there is a set of printout statements to print the values of all signals. These are used to print a counterexample if necessary. They are

ignored in a verification run.

When leaving the atomic section in line 7, the process resets the *exclusive* variable to 0, allowing any other process execution to be interleaved.

In this template, one can observe the reaction only after the atomic section. This means that an observer will see the input values at the same time as the output values, and will not be able to infer causality relations without additional information. If the environment generates valid inputs, then the observer will observe the inputs and the outputs at the same time. This template can be used to model-check the Signal module for LTL properties. This is the implementation of the synchrony hypothesis in our Promela translation.

4.3 Equivalence of the Descriptions

In this subsection, we prove the equivalence of the Signal module with the Promela process by using observational equivalence [9]. We show that the first is trace equivalent to the second, and since we consider deterministic systems, the equivalence is compositional for LTL properties.

Definition 7 (Trace Equivalence) *Let M_1 and M_2 be two transition systems with the same alphabet. M_1 and M_2 are trace equivalent, denoted $M_1 \approx M_2$, if they have the same language: $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.*

In our case, trace equivalence means that one can observe the same events on the Promela process and the Signal module. Since the model of the Signal module is contained within and invoked from inside the Promela process, the proof is by construction.

We define observers on the Promela process and on the Signal process as processes that are synchronously composed with their respective models. Let us denote V_s^{if} to be the set of input and output signals for the interface of the Signal process. We define a Signal observer as a transition system O_s over V_s^{if} that can observe any possible run over V_s^{if}. The observer O_s is synchronously composed with the transition system of the Signal module and observes after each synchronous reaction, sampling both inputs and outputs.

We define V_p^{if} as the set of variable pairs (value/clocks) in the Promela process that correspond to V_s^{if} in the Signal module. These are the observable input and output of the Promela process, corresponding to the interface of the Signal module.

We define an observer O_p on the Promela process as a transition system over V_p^{if} that can observe any possible run over V_p^{if}. The observer O_p is composed synchronously with the Promela process, observing only after the synchronous reaction. Indeed, the observer cannot observe inside the atomic section since it is guarded by the expression: `exclusive== 0` or `exclusive==`

`observer_pid`, and variable "*exclusive*" is not in the alphabet of the observer. We can now state the following theorem.

Theorem 4.1 *Let $M_s = \langle S_s, T_s, s_{s_0} \rangle$ and $M_p = \langle S_p, T_p, s_{p_0} \rangle$ be transition systems for a Signal module and its Promela translation respectively . Let O_s be an observer over V_s^{if} on M_s, O_p be an observer over V_p^{if} on the M_p, and let L_s and Lp be the languages they observe, respectively. Then the following holds: $L_s = L_p$ and by Definition 7, $M_s \approx M_p$ with respect to O_s and O_p.*

Proof sketch: This is a proof by construction. The two observers observe observables with the same alphabet. In fact, we can define a bijective function that maps from signals in V_s^{if} to pairs value/clock in V_p^{if}. Therefore, we can map the observations of the two observers to the same alphabet. By definition, observer O_s can observe any legal behavior of the Signal module. Observer O_p can observe the configuration of the pairs in V_p^{if} only when the "proctype" execution reaches line 8 of Figure 4. This is because the "exclusive" variable guards each step of O_p so that the guard is false when the "proctype" is executing inside the atomic block ("exclusive" variable is set to the process identifier of the "proctype" containing the atomic instruction and reset to 0 when the block ends)

In line 4 of Figure 4, any possible configuration of input signals of V_p^{if} can be generated. After line 5 is executed, the configuration of V_p^{if} is either a valid signal configuration for the Signal module or a stuttering step. Line 5 calls the C function generated by the Signal compiler. This function is deterministic and, if provided the right input, returns true and simulates the correct reaction for the Signal program. If the input configuration is not correct (does not satisfy the clock tree) it returns false. In the latter case all signal clocks are set to 0, generating a stuttering step.

The execution of line 6 does not modify V_p^{if}. By construction, line 6 just prints the current state of input and output variables to provide an error trace. (This happens only when replaying an error trace in Spin; during a verification run all print instructions in this line do not execute at all).

We prove that observer O_p can observe any word observed by O_s. All inputs are generated in line 4. After line 5, if the input signals generated were consistent with the clock tree the configuration of V_p^{if} is the same as V_s^{if} and because line 6 does not modify it, this is still true in line 8 when O_p carries out its observation. Given that at any time all possible correct inputs can be generated, then $s \in \mathcal{L}(M_s) \implies s \in \mathcal{L}(M_p)$.

We now prove that observer O_s observes all words observed by O_p: $s \in \mathcal{L}(M_p) \implies s \in \mathcal{L}(M_s)$. After line 5 we have the same configuration of V_p^{if} and V_s^{if} or a stuttering step for V_p^{if}. In subsection 3.4, we extended the

synchronous transition system to allow for stuttering steps. In a stuttering run we can add of remove an unbounded number of stuttering steps. Therefore, all words observed by O_p can be observed by O_s. This proves that $s \in \mathcal{L}(M_p) \leftrightarrow s \in \mathcal{L}(M_s)$ and therefore the equivalence of the two languages. ■

Now, given the synchronous composition, the observers observe the same behaviors by construction. The difference between the Signal module and the Promela process is the stuttering step, but because stuttering steps do not change the semantics of the transition system, we can state that the two languages observed are the same. Given that the two modules are trace equivalent, LTL properties verified on the Promela model hold for the Signal model. In Promela, an observer can be implemented using a never claim. Spin converts an LTL property to a never claim [8], which observes the same language as the observer we used in the theorem.

5 Verification Strategy for GALS Architecture

We now extend our focus from synchronous modules to their asynchronous composition into a Globally Asynchronous, Locally Synchronous (GALS) architecture. We will use asynchronous model checking to enumerate all execution interleavings of the synchronous blocks in the GALS system to avoid unexpected problems of incorrect behaviors or deadlocks. The Spin model checker can find out if these kinds of problems arise, and can return the execution path that leads to the error, if one exists.

The use of a Loosely Time-Triggered Bus Architecture (LTTA) is one way to implement globally asynchronous communications around synchronous islands of computation [3]. We abstract the LTTA bus to a Promela channel of size 1, and prove their equivalence through model checking. This abstraction will allow us to verify more complex systems than possible using the standard translation of the LTTA. We then show that the GALS architecture model, where we replace LTTA bus structure with Promela FIFO channels, is equivalent to a real hardware implementation.

5.1 Communication Channel Abstraction

The basic structure of the LTTA is illustrated in Figure 5. It is composed of three devices, a writer, a bus, and a reader, indicated by the superscripts $()^w$, $()^b$ and $()^r$ respectively. Each device is activated by its own, approximately periodic, clock. At the n^{th} clock tick (time $t^w(n)$), the writer generates the

value $x^w(n)$ and an alternating flag $b^w(n)$ such that:

$$b^{\mathbf{w}}(n) = \begin{cases} \mathit{false} & \text{if } n = 0 \\ \mathit{not}\ b^{\mathbf{w}}(n-1) & \text{otherwise} \end{cases}$$

Both values are stored in its output buffer, denoted by y^w. At any time t, the writer's output buffer y^w contains the last value that was written into it. At $t^b(n)$, the bus fetches y^w to store in the input buffer of the reader, denoted by $y^{\mathbf{b}}$. At $t^r(n)$, the reader loads the input buffer y^b into the variables $x(n)$ and $b(n)$. Then, in a similar manner as for an alternating bit protocol, the reader extracts $x(n)$ if and only if $b(n)$ has changed, meaning that the value also changed. In order to prove the correctness of the protocol, we need to prove that, under some hypotheses on the clocks, the sequences x^w and x^r must coincide, i.e., $\forall n \cdot x^{\mathbf{r}}(n) = x^{\mathbf{w}}(n)$.

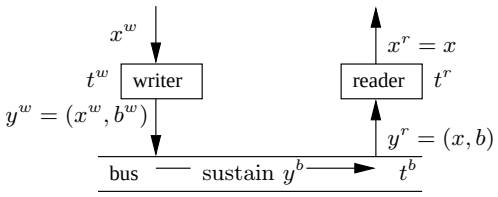

Fig. 5. Basic structure of the LTTA

In [3], it is shown using symbolic model checking that a discrete Signal model of the LTTA protocol satisfies the desirable requirement of ensuring a coherent distribution of clocks. However, the assumptions ensuring correctness of the actual LTTA protocol are quantitative in nature (tolerance bounds for the relative periods, and time variations, of the different clocks). For the protocol to be correct, the clocks must be quasi-periodic (periods can vary within certain specified bounds), and must relate to each other within some specified bounds.

In order to allow for standard model checking techniques to be used, two kinds of abstractions of the protocol are necessary. The first one is to use only boolean data types for bus payload. It is clear that this protocol and the property to be verified are data-independent with respect to the type X of data which is transmitted. Therefore, it is sufficient to verify this protocol with a finite set of finite instantiations of the type X. It is then possible to deduce the correctness of the protocol for any instantiation of the type X. However, in [3], only the instantiation of X by the type of booleans is considered.

The second abstraction is about the relative rates of the clocks. The condition that the bus must be faster than the writer is an abstraction on the

ordering between events. This is abstracted by a predicate that never two writes occur between two bus transfers. Similarly, there must not be two bus transfers between two reads. These assumptions are encoded as fairness conditions in the environment model, i. e. the clocks of the reader and writers.

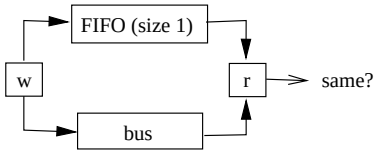

Fig. 6. Verification setup for equivalence of LTTA with FIFO buffer of size 1

Figure 6 shows the model checking setup for verifying that the LTTA bus is equivalent to a FIFO buffer of size one. We created a reader and a writer process and connected them to the translation of the Signal LTTA bus model. The writer process generates data values, writes them both to the bus and to the FIFO, and the reader process compares the values for correctness. We placed an assertion in the reader process that the values read are always equal.

Theorem 5.1 *Let M_L be an LTTA bus and M_C be a Promela channel with capacity of size 1. Assuming that the constraints about the clock rates for the reader and writer processes hold, then M_L is trace equivalent to M_c.*

Proof sketch: By model checking. The LTTA bus is checked, as a black box, to have the same behaviors as the FIFO buffer, in all possible environments that can be generated by the reader and writer processes. That is, the writer generates every possible value on the interface signals and we show that the values read are the same. ■

The clock assumptions with the handshaking inside the LTTA bus can be modeled by the blocking behaviors of the FIFO channels.

5.2 Communication Abstraction and Scalability Benefit

In this subsection, we comment on benefits that the abstraction of the LTTA bus to a channel provides on the scalability of the verification.

To investigate this, we created three additional trace equivalence experiments, shown in Figure 7(a), (b) and (c) where we pipeline the busses. We verify the system at the end of the pipeline to prove that it still behaves correctly with respect to the correctness condition of trace equivalence with a pipeline of FIFOs. The performance numbers of the model checker for the three experiments are listed in Table 2.

The first entry in the table shows the results for verifying the equivalence of a single bus with a single FIFO buffer, the experiment described in the

previous subsection. The second entry lists the results for the two-stage setup of Figure 7(a). A reader/writer process is connected between the busses to read the output from one bus and write it to the next one. For this setup, one can easily notice major increases in state space and memory usage. For the pipeline with three stages, shown in Figure 7(b), we again connect the busses with intermediate processes to forward the data. In this case, we were not able to check the full model state space because the amount of memory required was too high. We used state-space compression in Spin and were able to check the program but without an exhaustive coverage of all possible states. Still, with this technique, the results show major increases – especially for memory usage, which is above 1.5 GB.

To be able to exhaustively model-check a three bus system, we simplified the program using a technique suggested in the Spin book [8]. We directly connected the output of one bus to the input of the following one, removing the additional processes. With this configuration we obtained a system made up of 3 processes described in signal (LTTA bus) and 2 processes written in Promela (reader and writer). The resulting system was simple enough to be exhaustively verified, and the statistics are listed in the table as the simplified model. The required memory is slightly above 1GB.

Finally, as shown in Figure 7(c), we replaced all LTTA modules with Promela channels and checked for trace equivalence. Using the channel abstraction, the experiment ran in less than half a second with a state vector of 64 bytes, requiring about 2MB of memory.

These results show a tremendous benefit in using the abstraction of LTTA bus modules to Promela channels (as shown from the data of Table 2). We ran our experiments using Spin Version 4.1.3 (April 24 2004), on a dual processor machine with two 3.2 GHz Xeon processors and 6 GB of RAM, running Linux Fedora Core 2.

Table 2
Performance number for the model checking of the LTTA

# of buses	State -vector	Depth reached	States stored	Transitions	Atomic steps	Memory used	Time Used
1	168 bytes	3294	1097	3190	14230	1.78 MB	0.01 s
2	324 bytes	628960	235681	1.11e+06	5.27e+06	145 MB	6.30 s
3 (compression)	480 bytes	14,165,911	3.04e+06	2.19e+07	1.05e+08	1,649 MB	189 s
3 (simplified)	296 bytes	4,219,560	1.56e+06	9.59e+06	4.55e+07	1,073 MB	56 s
3 (channels)	64 bytes	1331	4661	20226	14119	1.78MB	0.04 s

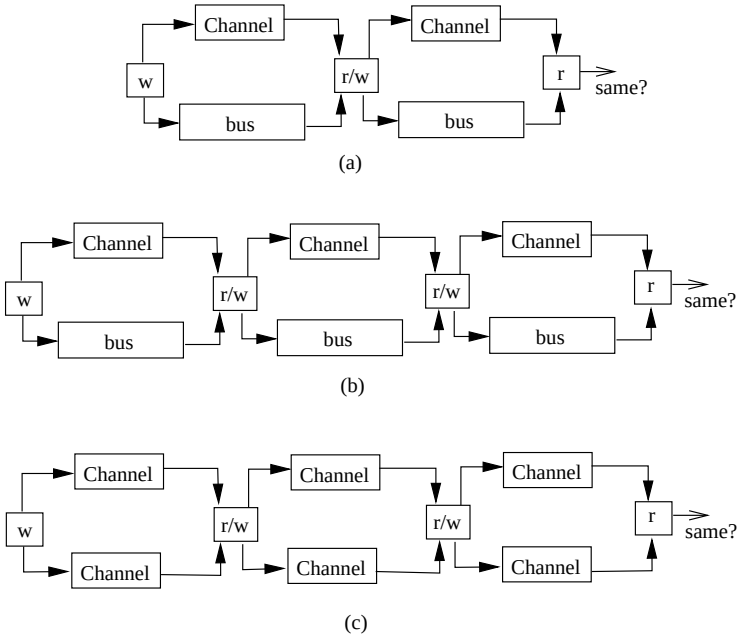

Fig. 7. Scalability experiments for the LTTA example: (a) 2 stage pipeline using LTTA translation (b) 3 stage pipeline using LTTA translation (c) 3 stage pipeline using only Promela channels.

5.3 Verification Model for GALS Architecture

In this section, we prove that our Promela model is equivalent to a hardware implementation using synchronous modules connected by LTTA busses. In this architecture, each synchronous block is wrapped into logic that gates the input clock to have control on when to fire the reaction. This happens when all inputs are present. Thus, this logic implements the firing condition – the clock relation guard – for the synchronous module.

Figure 8 shows the equivalence relation we want to prove, where the upper part is the structure of our Promela model and the lower part of the figure represents the hardware structure.

To integrate Promela models of Signal components into this kind of GALS deployment, we need to slightly modify the translation template defined in Figure 4. Since the input is now generated by other components in the GALS architecture, the original input generation step is replaced by a step where we convert the FIFO channels into signals with variables for their value and clocks. The process is then fired by calling the Signal-compiler-generated C function, and if the return value is true, the input channels will be emptied; the output values will be read from the signals and written to the channel. If the call return false, then inside the Promela process the reaction will be converted to a stuttering step, and the input channels will not be emptied. Thus, we

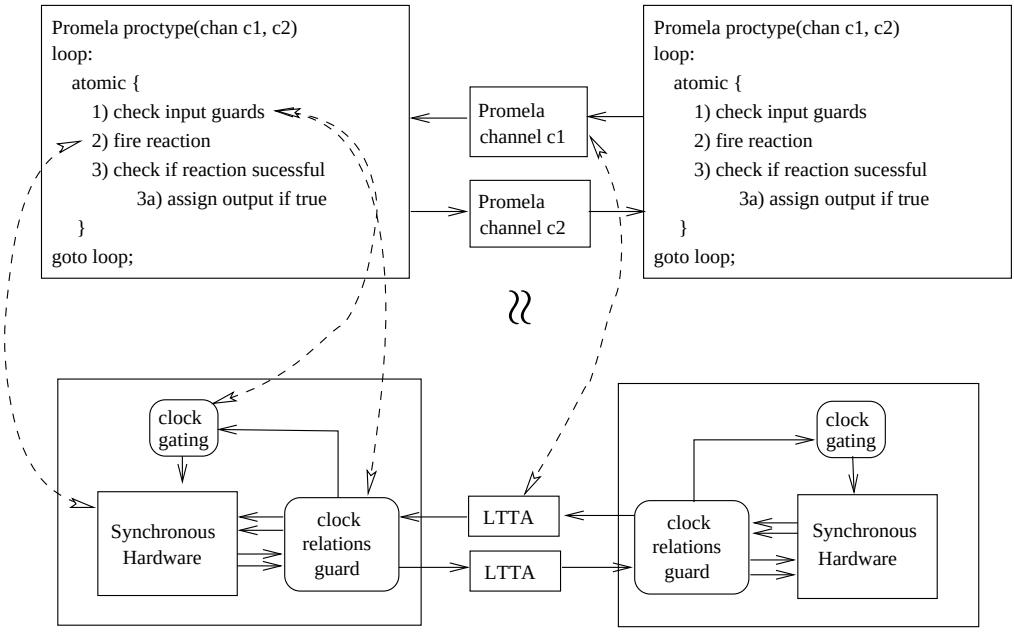

Fig. 8. Hardware implementation of the GALS model.

use the return value of the Boolean function to evaluate if the clock relations are satisfied, at which condition the input channels can be emptied and values can be written to the output channels. Now, based on these conditions, we can state that using the return value of the C function properly simulates the input guard – the clock constraints – and correspond to the implementation.

Theorem 5.2 *Let $S_s = \langle M_s, C_s, G_s \rangle$ be a system model where $M_s = \{m_{s_0}, ..., m_{s_n}\}$ is a set of synchronous modules modeled inside our Promela wrapper, $C_s = \{c_{s_0}, ..., c_{s_m}\}$ is a set of Promela channels linking the synchronous modules and $G_s = \{g_{s_0}, ..., g_{s_n}\}$ is a set of guards on the set of Promela channels C_s. Let $S_i = \langle M_i, C_i, G_i \rangle$ be an implementation of the system, where $M_i = \{m_{i_0}, ..., m_{i_n}\}$ is a set of synchronous modules, and $C_i = \{c_{i_0}, ..., c_{i_m}\}$ is a set of channels equivalents to Promela channels, like the LTTA is, and $G_i = \{g_{i_0}, ..., g_{i_n}\}$ is a set of guards on the set of implementation channels C_i. If $\forall x \in 0..n | \mathcal{L}(g_{i_x}) = \mathcal{L}(g_{s_x})$ then $S_s \approx S_i$ for observers on the communication channels.*

Proof sketch: We assume that all hardware modules are correct with respect to the corresponding Signal specifications. By Theorem 5.1, we know that the LTTA bus is equivalent to a Promela channel, with their languages being the same. By Theorem 4.1, we know that the Promela translation of the synchronous component is equivalent to the component. By construc-

tion, the guards in the hardware model implement the same Boolean function of the guards in the Promela model. Then, by composition of languages, the Promela model is trace equivalent to the implementation. Or, given the equivalence of $\forall x \in [0..n]$ $(\mathcal{L}(m_{s_x})||\ \mathcal{L}(g_{s_x})||\ \mathcal{L}(c_{s_o})||...||\mathcal{L}(c_{s_m}) \approx \mathcal{L}(m_{i_x})||\ \mathcal{L}(g_{i_x})||\ \mathcal{L}(c_{i_y})||...||\mathcal{L}(c_{i_m})$, then, by implication $S_s \approx S_i$. ■

Since the systems are trace equivalent, if an LTL property ϕ holds on the Promela specification it also holds on the hardware implementation: $S_p \models \phi \leftrightarrow S_i \models \phi$.

By using our integration approach, in the Promela model, a designer does not have to manually code the guards verifying the clock relations. The usage of the stuttering step inside the translation template models the possible interleavings for asynchronous arrivals of data from different sources, and enables practical system verification. However, in the hardware implementation, the guards will have to be manually implemented if the hardware components do not generate stuttering steps when fired with inputs that do not respect the clock constraints.

6 Verification Example

In this section, we use an automobile Central Locking System (CLS), illustrated in Figure 9, as a GALS system example. It has several components, each described by an independently clocked Signal process, connected to each other using LTTA channels; by means of the techniques introduced in the preceding sections we can model the LTTA channels by means of Promela channels for the verification task. The system specification includes a "Key" control, that can issue three commands: lock, unlock and open trunk.

A second component of the CLS is the "CLSController". It is a control unit that is connected to all the other components and issues the appropriate commands to them. Another component of our system is the "ImpactSensor". This device is connected to the "CLSController"; in the case an accident is detected, the "CLSController" must unlock all the doors overriding any other command from the "Key". The last two components present in our model are the "DoorMotor" subsystems. They execute the physical opening and closing of the two doors, inform the "CLSController" when the operation is completed and report the current status.

We present the results for the verification of a very important property: under any circumstances, if an accident occurs, the doors of the car are automatically unlocked. We ran the model checking with different options to verify the property $\Box(\mathit{impact\ happens} \rightarrow \Diamond\Box(\mathit{doors\ unlocked}))$.

Table 3 shows the interesting performance numbers for these checks. We

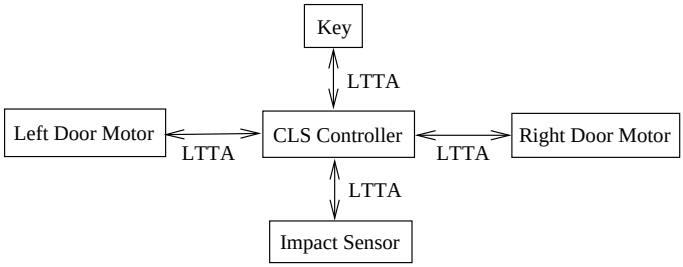

Fig. 9. Block diagram of Central Locking System example

Table 3
Performance number for the model checking of the CLS

Simulation description	State -vector	Depth reached	States stored	Transitions	Atomic steps	Memory used	Time used
Full model 1	616 bytes	19897839	6.218e+06	4.506e+07	3.177e+08	Out of memory	2 hours+
Full model 2	520 bytes	21618700	7.614e+07	6.904e+08	4.735e+09	1970 MB	5:24:11
Without ImpactSensor 1	496 bytes	6109897	2.593e+07	2.942e+08	2.360e+09	3435 MB	2:26:13
Without ImpactSensor 2	496 bytes	6109897	2.306+07	2.636e+08	2.125e+09	Out of memory	2 hours+

discuss each of the verification runs captured in the rows of Table 3, in turn. The first run is the verification of the full model with the partial order reduction and all the possible compressions activated, where we ran out of memory after more than 2 hours. The SPIN model checker cannot use more that 4GB of RAM.

The second verification run for the full model uses the BITSTATE hashing algorithm of Spin. This algorithm does not store the full description of the states it explores, but uses a hash function on the state description instead to identify already explored states. In the event of a hash clash, two different states are identified as equal, and some execution path is not explored. We report the succesful verification of the property using this method for the system with five processes.

To be able to completely model-check our system, we introduced an abstraction. We noticed that the "ImpactSensor" process was a source process [8] responsible only for random generation of impacts. We changed the model a little bit by removing the "InpactSensor" and instead sending impact signal directly to the "CLSController".

This reduction in the number of processes reduces the number of possible interleavings, and using the partial order reduction and, the system model has

become small enough to perform an exhaustive verification on it. The third entry in the table shows the results after this change. While it still requires large amounts of memory, this instance of the system can be fully verified for the accident property. One can note that the partial order reduction is critical in verifying this instance, and without it we cannot verify the system. The fourth entry shows how one runs out of memory when trying to verify the system without using the reduction.

One could wonder why this design example, which is of reasonable size, requires so much memory. We are investigating ways to reduce the memory required to verify GALS architectures using our translation. One problem arises from the use of 32 bit integers variables for boolean values in the C code generated by the Signal compiler. The use of a single bit for booleans could potentially greatly reduce the size of the state vector. There is room for optimization of the translation to make sure that the variables used in the C translation only range over the necessary minimal domain.

7 Conclusion and Future Work

We presented an approach for model checking integration of synchronous Signal components into a GALS system architecture. The integration of the Signal synchronous modules into the Spin model checker requires to convert the Signal specification to a transition system that we then translate to Promela. By using the C output of the Signal compiler and in-lining it into the Promela specification, it is possible to create a verification model in very little time. We proved the equivalence of this translation to the original Signal module. We then abstracted GALS communication by showing that the LTTA bus is equivalent to a FIFO buffer. Then we performed the verification of an asynchronous integration of synchronous modules into an automotive central locking system, and described our performance experiments.

The results obtained so far are promising towards accelerated verification of GALS architectures. Promising areas of applications and avenues of extension comprise the use of this framework for conformance checking GALS designs at different design abstraction layers, as conducted in [14]. While the core of this work is about integrating synchronous components, one could also extend this as the refinement of a synchronous component into a set of asynchronous modules. One may desire to insert desynchronization buffers into such a refinement and model check the correctness of the refinement using flow preservation refinement and real-time constraints. Future investigation efforts will be on closing that gap. Although the work presented in this paper does not imply a method to proof refinement it can be used to verify the de-

ployment of the specification on an asynchronous architecture. Another very interesting element of future work following this path would be to study different types of interconnections between the synchronous components. This would allow for verification of more complex GALS systems going beyond point-to-point connections. Also, since our proof is largely independent of the input language, we could apply it for the conversion of components specified in other languages, as long as they correctly implement the synchrony hypothesis, and generate a C model where we can invoke the reaction.

Acknowledgments

Our work was partially supported by the UC Discovery Grant and the Industry-University Cooperative Research Program, by the Semiconductor Research Corporation, as well as by funds from the California Institute for Telecommunications and Information Technology. We are grateful to the anonymous reviewers for their insightful comments.

References

[1] T. P. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[3] A. Benveniste, P. Caspi, P. L. Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In J. Sifakis and A. Sangiovanni-Vincentelli, editors, *Proc. of 2002 Conference on Embedded Software, EMSOFT'02*, volume 2491 of *LNCS*, pages 252–265. Springer Verlag, 2002.

[4] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[5] G. Berry. *The Foundations of Esterel*. MIT Press, 2000.

[6] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.

[7] D. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, 1984.

[8] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[9] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[10] S. Ramesh, S. Sonalkar, V. D'Silva, N. Chandra, and B. Vijayalakshmi. A toolset for modelling and verification of gals systems. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 506–509. Springer, 2004.

[11] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proc. IEEE/ACM Design Automation Conf.*, June 2001.

[12] M. Singh and M. Theobald. Generalized latency-insensitive systems for gals architectures. In *FMGALS*, 2003.

[13] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*. IEEE Computer Society, 2004.

[14] J.-P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet. Polychrony for formal refinement-checking in a system-level design methodology. In *Special Issue of Fundamenta Informaticae on Applications of Concurrency to System Design*. IOS Press, Aug. 2004.