A Novel Architecture Prototyping Framework With Generic Properties Verification for Sub-architectures

Ismail Assayad*, Lamia Eljadiri, Moez Krichen, Abdelouahed Zakari, Wilfried Adoni, Tarik Nahhal

Abstract—**Formal verification has become very useful and popular in last decade in area of embedded systems design and in analysis of critical systems. It can reveal common errors, check system invariants, but also verify more complex properties defined by temporal logic formulas. To reduce the time-to-market for embedded architectures and assist SystemC designers in the complexity of verification process at design time, we advocate a novel approach where (a) generic safety properties are used for sub-architecture verification during architecture prototyping, and (b) sub-architecture models are built according to the presented (Behavior, Interactions, and Priority) framework, in order to ensure that models verification results still hold for subsequent architecture prototype candidates. This approach best helps the designer at two levels. At the prototype dimensioning level, it introduces a sets of pre-defined properties for common sub-architecture classes. At the verification level, it enables to check safety properties of a sub-architecture without the need to redo the verification process for next prototypes comprising it. We present the framework and show its feasibility on several examples.**

Index Terms—**Modelling, Behavior Interaction Priority, Safety, Embedded Architectures, Generic properties, Verification**

I. Introduction

THE verification of embedded sub-architectures plays an important role in the design flow of embedded architectures. The formal verification is a powerful technique because it is based on mathematical proofs to describe the absence or existence of errors, most of them demonstrated by the counter-example method.

The SystemC language is a defacto-standard for embedded systems design and became the basic language of most of industrial production companies [1]. This allows research works to focus on checking architectures specifications of SystemC programs which are first translated to equivalent formal model such as timed automata models and then checked using model checking tools such as SPIN [2], SMV [3] and BIP [4].

We have successfully used the state-of-the-art automatic formal verification methods. More particularly, we use the model checking technique [5][6][7], which takes into account all the possible behavior of the design. The essential prerequisite of the model checking technique is formal specification of specification properties. For this purpose, we use suitable temporal logics [8][9][10][11]. Strong expressiveness of these logics allows us to express all the typical requirements on a sub-architecture behavior. However, a nontrivial effort is required to specify these formulas and to use model checking tools to verify them for the different architecture prototypes under verification. For this reason, model checking is not yet the standard method of verification.

On the other hand, SystemC simulation suffers the drawback of being incomplete. SystemC-level formal verification of implementations with respect to the specifications stated in this paper brings designers another tool which can help them to purify the code and to ensure its correctness. Furthermore, unlike simulations, such a verification process can lead to backward refinements of sub-architecture designs through provided counter-examples, which is helpful to future reuse of the designs.

There is another different way of using formal verification in architecture design which consists in the creation of high-level abstract models of the design instead of verifying low-level SystemC programs. The problem of this approach which is out of the scope of this paper is to ensure that an abstract model is sound. Moreover using the advocated SystemC-level way of verification has the advantage of being cycle-accurate and thus it allows for checking standard fine grain properties and also for compositional verification of sub-architectures. These simple generic verifications are not always possible with the high-level abstract verifications.

The structure of the paper is the following. Sections II and III present the contributions and related works respectively. Section IV presents the modeling approach and the conversion method. Section V presents the operated sub-architecture model transformations. In Section VI, we first show the effect of sub-architecture model reductions on the final architecture model size for a modulo four counter; then we present the verification results for a CPU sub-architecture comprising four other sub-architectures; afterwards, we present the set of pre-defined properties and verification results for a FIFO and an AMBA AHB sub-architectures. Finally, Section VII concludes the paper.

Manuscript received June 16, 2020; revised January 16, 2021. This work was partially supported by LIMSAD Lab, Faculty of Sciences, School of Electrical and Mechanical Engineering, ENSEM and Hassan II University of Casablanca.

I. Assayad is a Professor of the National School of Electrical and Mechanical Engineering, ENSEM and LIMSAD Lab, Faculty of Sciences, Hassan II University, Casablanca, Morocco. IAENG member (Corresponding author E-mail: iassayad@gmail.com).

L. Eljadiri is a PhD candidate of Faculty of Sciences, LIMSAD Lab, Hassan II University, Casablanca, Morocco. E-mail: lamia.eljadiri@taalim.ma.

M. Krichen is a Professor of Faculty of CSIT, Albaha University, Saudi Arabia and ReDCAD Lab, University of Sfax, Tunisia. E-mail: moez.krichen@redcad.org.

A. Zakari is a Professor of University of Lorraine, LGIPM Lab, Metz, France. E-mail: abdelouahed.zakari@univ-lorraine.fr.

W. Adoni is a PhD candidate of Faculty of Sciences, LIMSAD Lab, Hassan II University, Casablanca, Morocco. E-mail: adoniwilfried@gmail.com.

T. Nahhal is a Professor of Faculty of Sciences, LIMSAD Lab, Hassan II University, Casablanca, Morocco. E-mail: nahhalprof@gmail.com.

II. Contribution

We present a framework that supports the methodology depicted in figure 1. First of all, when the architecture prototype has been set, simulation is used to find execution errors and performance bottlenecks such as buses available bandwidth, processor utilization and memory conflicts performance. A new architecture prototype is to be selected if performance analysis results are not satisfactory. The identified bottlenecks helps the designer to consider more alternatives for the new architecture dimensioning. Then, the methodology consists of four phases:

- Designer Mapping,
- BIP model construction,
- Transformation and reduction,
- Verification

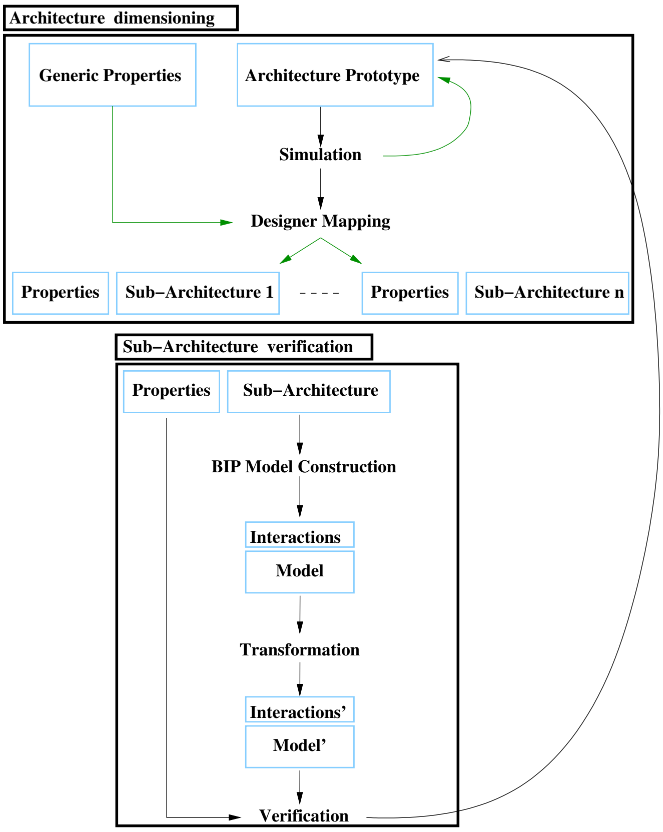


Fig. 1: Design methodology

a) Designer mapping.: This phase focuses on sub-architectures specifications. Our methodology relies on sets of generic safety properties pre-defined for each class of similar sub-architectures. For instance generic sets of safety properties are defined for synchronous FIFOs, asynchronous FIFOS, AMBA buses, RISC CPUs, and SRAM memories. Designer defined properties may also be added. In order to be able to start the next phases of our verification methodology, the designer has to write a list of pairs for each identified sub-architecture. The list defines the mapping between the variables in the generic properties formulas of sub-architecture class, and the names of the signals in that sub-architecture.

b) BIP model construction.: Details of the sub-architecture SystemC Program are involved in this phase. A Sub-architecture process behavior is modeled by a set of states and transitions. Each state is followed by a list of outgoing transitions with their corresponding guards and data computation, which are C expressions and C statements respectively. A new state is reached after the occurrence of a transition. Finally, different types of processes synchronization are modeled by the introduction of the concept of interactions between transitions.

c) Transformation and reduction.: In this phase, the transformation step adds additional interactions to the sub-architecture model so that sub-architecture verification results still hold for other architecture prototypes. These new interactions are incomplete ones which are added whenever a process of the sub-architecture is binded to a communication signal. with these incomplete interactions, the model takes into account in the current design verification iteration the possible occurrence of new communications, when considering different architecture prototypes involving this sub-architecture in new design iterations of the global loop in figure 1. The reduction step is an optimization step whose objective is to increase the verification speed. It significantly reduces the model size by replacing sequences of transitions between stable states with meta-transitions whenever it is possible.

d) Verification.: This phase checks the model against the pre-defined generic properties associated with the sub-architecture class. A new version of the sub-architecture has to be coded when verification results reveal specification errors. Then, after any change in the sub-architecture, simulation of the global architecture prototype is run again to check non regression w.r.t performance requirements of the architecture. Finally, either the current architecture prototype is kept and a new iteration is started for the verification of the new version of the sub-architecture; or a new architecture prototype meeting performance requirements is selected (global loop in figure 1).

We propose in this paper the use of SystemC-level generic verification of sub-architectures. To further assist designers in the verification process, we specify pre-defined properties for several classes of sub-architectures using PROMELA language. More particularly, properties are encoded as formula of LTL, i.e., Linear Temporal Logic [12]. These pre-defined properties are taken from standard specifications of sub-architectures behaviors.

The advantages of our verification approach are fourfold :

1) It has the advantage of being accurate because it is performed at code-level which is cycle accurate and bus accurate.
2) Our approach allows generic verifications of safety properties for sub-architectures programs. Moreover verifications are valid for all architecture prototype candidates comprising them. This is achieved using the signals-driven incomplete-interactions transformations on the generated models.
3) Verification process is improved by using pre-defined properties for each class of sub-architectures, i.e., those having common behavior specifications;
4) Verification speed is improved by reducing sequences of transitions in the models of sub-architectures processes whenever it is possible. Such reductions cannot be

applied in classical verification approaches that are not aware of stable/unstable states.

III. Related work

Formal verification techniques and tools are used in various domains in the literature [13][14][15][16]. Many related works for SystemC-level verification exist for embedded systems [17][18]. In this paper, the model we use for the conversion of SystemC architectures is the one presented in [19]. The reasons why we made this choice are multiple. First, this model separates the behavior layer from the interactions layer. With this separation it will be easier to implement our concepts of generic verifications in architecture prototyping by only making some transformations on the interactions layer of sub-architectures.

Second, unlike classical conversion methods in the literature which model the signals and the scheduler as normal processes, this separation makes it possible for us to propose a different method which does not use such processes. It therefore allows us to easily reduce some sequences of transitions into meta-transitions, something which become complex to do in classical methods due to the processes synchronizations with the automata of the scheduler.

Third, each signal is modeled with two variables and transition guards associated to the processes using them, and the variables are updated at the end of delta cycles if needed. Similarly, the scheduler effects on processes executions are included in the specification of the interaction and priority layers of sub-architectures. Our method does not need dedicated processes neither for the signals nor the scheduler and, hence, it decreases by construction at the beginning of the verification phase the product model size of sub-architectures.

A lot of works are also done in the field of abstract-level way of verification [7][20][21]; however these approaches are not suitable for our framework, because some strong abstracted models may be too different from low-level cycle accurate models and may not allow the verification of standard specifications properties. In addition to this loss of accuracy, they make the use of generic properties more difficult if not impossible.

Finally, the presented framework, is at our best knowledge, the first reported one on the use of generic verification of safety properties at cycle-accurate code-level in the context of architectures prototyping.

IV. Sub-Architecture BIP modelling

A. SystemC library

SystemC is a widely used C ++ library for the modeling of software/hardware embedded systems. It includes low-level descriptions of hardware such as RTL; and high level of SystemC-TLM descriptions which are functional and timed abstractions of hardware which require dedicated extension to the basic SystemC library. SystemC-TLM is a new level of description which is not present in other hardware description languages like Verilog that only support bit and signal types but no functional transaction data types.

The main constructs of the SystemC language are:

- Modules are the fundamental building block in a SystemC program. Modules support multiple processes inside them. Modules can also be used for describing hierarchy: a module can contain sub-modules, which allows to break complex systems into smaller more manageable pieces. Modules and processes can have a functional interface, and implementation details of IP blocks.
- Processes are used to describe functionality. SystemC provides three different processes to be used by hardware and software designers: methods (asynchronous blocks), threads (asynchronous processes) and clocked threads (synchronous processes).
- Ports of a module are the interface passing information to and from the module, and triggering actions within the module. Ports can be single-direction or bidirectional.
- Signals create connections between modules allowing them to communicate. SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver while unresolved signals can only have a single driver.

Before presenting the general principle of the conversion of sub-architectures programs to the BIP formal models, we give below a summary of the various language constructs in the form of an abstract syntax. The conversion is implemented for a large subset of SystemC as shown in the following abstract syntax :

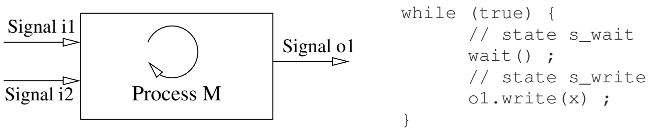
```
Program     := (Module)*, (Decl)*, (Init)*, (Bind)*
Decl        := Signal|Clock|Event|Var
Signal      := Signaltype, DataType, SignalId
Clock       := CloclType, ClockId
Event       := EventType, EventId
Var         := DataType, VarId
Init        := SignalId, SignalVal|
               ClockId, ClockVal
Bind        := ModuleId.portId, ModuleId.portId|
               ModuleId.portId, (SignalId|ClockId)|
               PortId, ModuleId.port|
               PortId, (SignalId|ClockId)


Module      := ModuleId, (Port)*, (Decl)*,
                         (Proc,CStat,Bind)*,
                         (ProcFunc)*,
                         (ModuleFunc)*
Port        := PortType, Datatype, PortId
PortType    := ScIn|ScOut|ScInout
Proc        := MethodProc|ThreadProc|CThreadProc
ProcFunc    := ProcId, (VarId)*, Stmt
ModuleFunc:= FuncId, Stmt
Stmt        := wait(EventId)|wait(EventId,int,Unit)|
               wait(int,Unit)|SignalId.wait()|
               wait()|EventId.notify()|
               EventId.notify(int, Unit)|
               VarId = PortId.read()|
               PortId.write(DataType d)|PortId.wait()|
               VarId = signalId.read()|
               signalId.write(DataType d)|Cstmt

MethodProc  := ProcId, (Sensitivity)*,
               (Initialize|dontInitialize)
ThreadProc  := ProcId, (Sensitivity)*,
               (Initialize|dontInitialize)
CThreadProc:= (ProcId, (PortId.pos()|PortId.neg())),
               (ResetSignal|dontResetSignal)
ResetSignal:= (SignalId|PortId), (true|false)
Sensitivity:= EventId|SignalId|ClockId|PortId
```

B. Principle of the modelling method

Let us consider a process M, two input signals i1 and i2, and one output signal o1. M sensitivity list contains the two signals i1 and i2. M pseudo-code is the following:



```
while (true) {
        // state s_wait
        wait() ;
        // state s_write
        o1.write(x) ;
}
```

Fig. 2: Process pseudo-code.

Let us see its conversion to state-transition automaton shown in the following figure.

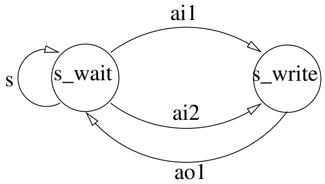


Fig. 3: State transition automaton for Process M.

Hereafter a brief explanation is given for this example:

- There are two states, one for the first SystemC construct i.e. state s_wait, and one for the second one, i.e. state s_write. The end state for the first construct is the start state of the subsequent construct.
- Transition labeled by ai1, respectively ai2, will be taken when event ei1, respectively ei2, of signal i1 respectively signal i2, notifies that new data is written on the signal. Notifications happen at the end of delta cycles where corresponding writes are made. In other words, in the product model, first transition is taken along with interaction ei1 • ai1 and second transition is taken along with ei2 • ai2. Both of them correspond to the end of wait() statement.
- Transition labeled by action ao1 is an immediate transition. The corresponding interaction is composed only of ao1 because signal write is not blocking. However the written value is accessible after the end of delta cycle only. In our conversion method, two variables ao1_next and ao1_now are associated to the signal, then during the transition, the value is stored on ao1_next, then it is assigned to ao1_now at the end of delta cycle. The read statement gets the value of ao1_now which is now equal to the old value of ao1_next defined in previous cycle.
- Self-loop transition labeled by S is taken when all available processes are waiting and corresponds to the end of current delta-cycle. Thus, the interaction composed of the set of processes S actions, must have a lower priority than interactions ei1 • ai1 and ei2 • ai2, so that it will occur only when all available processes are waiting.

C. BIP Modeling for the statements

We model the behavior of processes as described in previous Section. The modeling approach is centered around the use of extended automata whose interactions may be restricted if needed : either by simply removing unused ones, or by keeping all interactions and using priorities to choose some of them among the set of possible ones which depends on the processes states (figure 4). For each statement in the abstract syntax, we show how to obtain the corresponding model in the form of extended automata.

An extended automaton is a tuple $(L, X, Q, \longrightarrow)$ where L is the set of labels, X is the set of variables, Q is the set of control states and $\longrightarrow$ is a transition relation $\longrightarrow \in Q \times (L, G, F) \times Q$ such that G is the set of boolean function on X and F is the set of functions on X. An element $q \xrightarrow{l,g,f} q'$ of $\longrightarrow$ is defined as follows where v is the valuation function:

$$q \xrightarrow{l,g,f} q' \quad \wedge \quad g(v(X)) = true \quad \Rightarrow \begin{cases} (q, v(X)) \xrightarrow{l} (q', v(X)) \\ v(X) = f(v(X)) \end{cases}$$

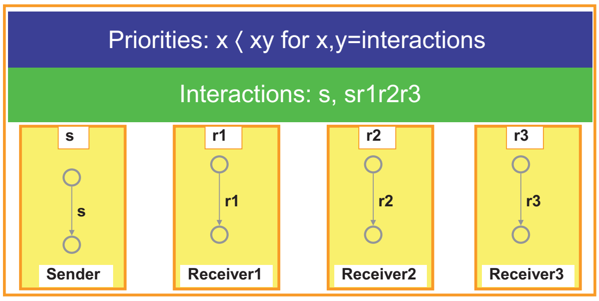


Fig. 4: BIP model example

While parsing a sub-architecture program, the following rules are applied to incrementally construct processes models. For instance, when a wait node is recognized its model is built and added on the fly to the current model. Like processes, non-recursive functions are also modelled using one automaton per function, then interactions are used to model function calls and returns. The occurrence of an interaction denoted $i_1 \bullet \ldots \bullet i_k$ means the simultaneous occurrence of processes actions whose labels are $i_1 \ldots i_n$. Processes interactions satisfy the following dominance property : $\forall I = i_1 \bullet \ldots \bullet i_k \; \forall (j_1, j_k) \neq (1, k)$ such that $1 <= j_1 <= k$ and $1 <= j_k <= k$ we have $i_{j_1} \bullet \ldots \bullet i_{j_k} <_{dom} I$ and we say that $i_{j_1} \bullet \ldots \bullet i_{j_k}$ is dominated by I. We call that interactions order $<_{dom}$ the dominance priority. Hereinafter, we present the models of some usual process statements.

1) Sequences of processes statements: For each statement on a set of variables we associate an automaton with initial and final control states as depicted in figure 5a. For a conditional statement $if\ c\ then\ stmt1\ else\ stmt2$, the corresponding automaton is obtained as shown in figure 5b. Loops are modeled in a similar way as shown in figure 5c whereas a function $f(p)\{stmts\}$ model is shown in figure 5d.

As will be explained later in Section V, it is worth noticing that when appending statements of a process we reduce the set of transitions between two waiting states into a single meta-transition labeled with the union of their actions, whenever it is possible, i.e., when the reduction is neutral for the verification results. Processes run without interleavings with other processes between the first and the second stable states. Although these reductions are faithful to SystemC executions semantics, they are not applied when inter-processes shared variables are involved in these transitions in order not to miss some interesting results of the verification.

2) Processes synchronization statements: A process may wait for an event e. The $wait(e)$ statement model is shown in the figure 5e, one transition corresponds to the immediate notification and the other to the delta cycle delayed notification. An immediate event notification $e.notify()$ statement notifies all the processes P_i which are waiting on that event. The $e.notify()$ statement model is shown at the top of figure 5f for a process P_1. The connector depicts the set of interactions $P_1.e \bullet I$ such that I is any interaction dominated by $P_1.e \bullet P_2.e \bullet \ldots \bullet P_n.e$.

The $e.notify(sc\_zero\_time)$ model is shown at the middle of the figure where the variable $e.active$ is set. It will be deactivated at the end of current delta-cycle, i.e., at the occurrence of the available processes S interaction along which these delta-cycle notifications are handled by setting the variables $e.event$. There is one variable $e.event$ for each process using e, and each of them is reset every time a waiting state transition is passed by corresponding process. Notice that, in the models of figure 5f, immediate notifications override pending delta-cycle and timed notifications.

Similarly, $e.notify(t)$ model is presented at the bottom of figure 5f. These timed notifications $e.notify(t)$ are used to emit delayed notifications at time t relative to current time. Delay value is stored by corresponding time variable $e.time$ which is decremented at each occurrence of the available processes $tick$ interaction. When it reaches the value 0, variables $e.event$ are set for appropriate processes. In the model, earlier notifications override ones scheduled to occur later.

3) Time interaction: Simulation time is advanced when the end of current cycle is reached with no more updates to do or no available process is waiting for such updates, and at least one process is waiting on time. In other words, time variable clk_tick is incremented when all available processes are in a stable state, and either there are no active events ($e.active \wedge time = 0$) neither active signals to notify ($s.events$), or no available process is waiting for them. In this case, the dominant interaction $P_1.tick \bullet \ldots \bullet P_n.tick$ is given higher priority than the dominant interaction $P_1.S \bullet \ldots \bullet P_n.S$, i.e., $P_1.S \bullet \ldots \bullet P_n.S <_S P_1.tick \bullet \ldots \bullet P_n.tick$. It is given lower priority otherwise, i.e., when new updates activate some processes transitions guards. We call the interactions order $<_S$ the delta-cycle priority.

4) Timeouts: A process may also wait for an event e with a timeout t. If an immediate notification e or a delayed one $[e.event]$ is received before time t, the timer is reset along with the notification. This reset is done by reseting the value of the timer variable x. The $wait(e, t)$ statement model is shown in figure 5g.

5) Processes communication statements: A SystemC signal s is associated with a pair of variables $s.next$, $s.now$, and a set of variables $s.event$ which aure used to notify signal value changes for the processes during their S interaction. There is one variable $s.event$ for each process using s, and each of them is reset every time a waiting state transition is passed by corresponding process. For a signal $s.write(exp)$ statement, if the value being written exp is different than the current value $s.now$, the variables $s.event$ are set during the processes interaction. $s.next$ will contain only the last exp value written before the interaction. Thus, writing repeatedly overwrites the previous values. The $s.write(exp)$ statement model is shown in figure 5h. The $x = s.read()$ statement model is shown in figure 5i.

A process overrides its events sensitivity by calling $next\_trigger(e)$. If done, the process will be triggered by event e. In addition to events, a timeout may also be used to specify a duration t after which the process will be triggered. The model of $next\_trigger(e, t)$ is presented in figure 5k, where variables $nt.active$, $e.ntactive$, and $nt.time$ are set. $nt.active$ is used as a guard in the $sensitive$ statement model transitions. When it is set, then $sensitive$ model follows the trigger sensitivity transitions which end by reseting $nt.active$ in order to get back to the default sensitivity transitions for next execution. When it is reset it only follows the default sensitivity transitions. Model of $next\_trigger()$ statement without arguments simply resets $t_i.active$. Model of $sensitive << e, t$ with only one event and a time value is given in figure 5m. Notice that for processes of type "thread", the sensitivity statement is used for the $wait()$ statements model, whereas for processes of type "method" the sensitivity and $next\_trigger$ statements are used for the initial state transitions model.

Finally, the $f(x)$ method call statement model is shown in figure 5l for a process P_1. connectors denote the interactions between this model and the method body model, i.e. $P_1.bf' \bullet P_{1_{body}}.bf'$ and $P_1.ef \bullet P_{1_{body}}.ef$ with an assignment of x to formal parameter p in first interaction.

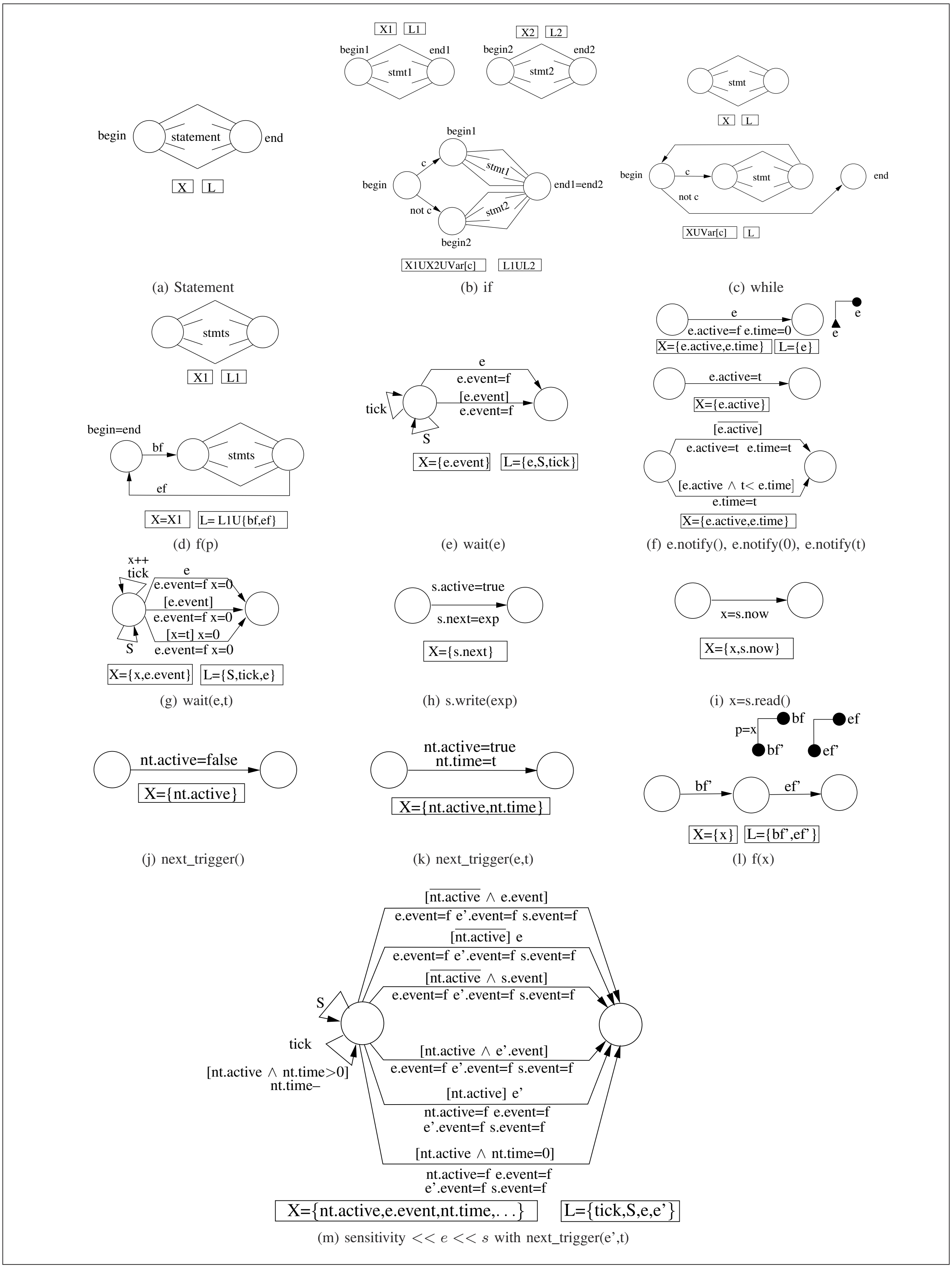
D. Sub-architecture BIP Model

In this section we present the processes modelling composed of their behaviors, their interactions and corresponding updates, and the priorities over these interactions.

A process execution goes through a sequence of delta-cycles. To express the semantics of the delta-cycles, we distinguish for each process between stable states, i.e., states corresponding to waiting statements, and transient states corresponding to the beginning of all other statements. The end of each delta-cycle requires a global synchronization between all the processes (figure 6). This is represented by the interaction over the processes S actions and involves update of the new values for either the next delta-cycle or the next tick-cycle. Any other interaction I which is not accompanied by an advance of time has higher priority than the former interaction, i.e, $\forall$ interaction I such that $tick \notin I$ and $S \notin I$ $P_1.S \bullet \ldots \bullet P_2.S <_\delta I$. We call the interactions order $<_\delta$ the delta-step priorities.

An occurrence of the S's interaction means the end of the current delta-cycle. Thus variables updates are executed with this occurrence and new values are ready for the next delta-cycle as follows.
For each pair of variables $(e.active, e.time)$ such that $e.active = t \wedge e.time = 0$:

- For all the available processes using $e.event$: $e.event = t$
- Cancel later notifications by variables reset : $e.active = f$ and $e.time = 0$



(a) Statement (b) if (c) while

(d) f(p) (e) wait(e) (f) e.notify(), e.notify(0), e.notify(t)

(g) wait(e,t) (h) s.write(exp) (i) x=s.read()

(j) next_trigger() (k) next_trigger(e,t) (l) f(x)

(m) sensitivity $<< e << s$ with next_trigger(e',t)

Fig. 5: Part of statements models

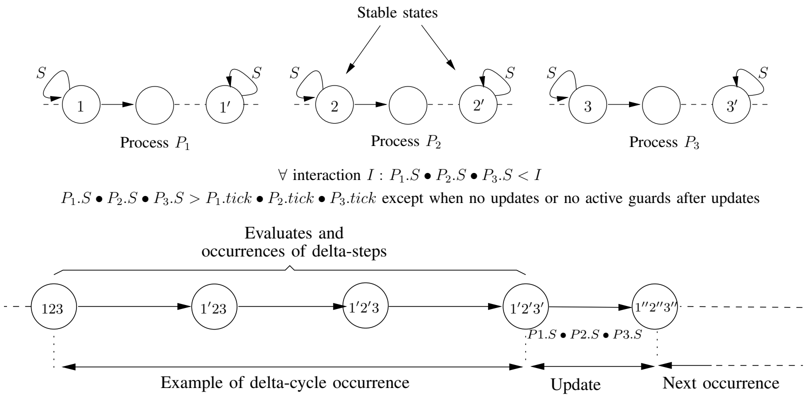

Stable states
S
S
1
$1'$
Process P_1
2
$2'$
Process P_2
3
$3'$
Process P_3
$\forall$ interaction I : $P_1.S \bullet P_2.S \bullet P_3.S < I$
$P_1.S \bullet P_2.S \bullet P_3.S > P_1.tick \bullet P_2.tick \bullet P_3.tick$ except when no updates or no active guards after updates
Evaluates and occurrences of delta-steps
123
$1'23$
$1'2'3$
$1'2'3'$
$1''2''3''$
$P1.S \bullet P2.S \bullet P3.S$
Example of delta-cycle occurrence
Update
Next occurrence

Fig. 6: Sub-architecture delta-cycle example

For each pair of variables $(s.now, s.next)$ such that $s.now \neq s.next$:

- $s.now = s.next$
- For all the available processes using $s.event$: $s.event = t$

An occurrence of the $tick$ interaction means the end of the current time cycle. Thus variables updates are executed with this occurrence and new values are ready for the next time-cycle as follows.

For each pair of variables $(e.active, e.time)$ such that $e.active = t \wedge e.time > 0$: topsep=0pt

- $e.time = e.time - 1$

For each pair of variables $(e.active, e.time)$ such that $e.active = t \wedge e.time = 0$: topsep=0pt

- For all the available processes using $e.event$: $e.event = t$
- Reset notifications : $e.active = f$

Let the priority order $<$ defined as the union of orders $<_\delta$, $<_S$, and $<_{dom}$. The model of the restricted sub-architecture product is $(Q, I, \rightarrow_<)$ where Q is the set of states of processes models product, I is the set of interactions, and $\rightarrow_<$ is the restricted transition relation defined as follows :

$(q, i, q') \in \rightarrow_<$ iff $(q, i, q') \in \rightarrow$ and $\forall (q, j, q'') \in \rightarrow$ we do not have $i < j$

The priority order $<$ ensures the correct occurrences of delta-steps, delta-cycles and prohibits dominated interactions in the sub-architecture model.

In order to validate the modelling approach by experiments, we considered a benchmark of forty SystemC programs comprising the various SystemC primitives, and different scenarios of processess synchronizations. For each program we then generated the corresponding model and we successfully examined the conformity between models and programs execution traces. For that, we instrumented the programs in order to produce test traces. Then we checked whether the behavior of the traces are valid behaviors of the models.

V. Sub-architecture model transformations

At the property-level the verification is generic, which means that the pre-defined safety properties for each class of sub-architectures are generic for all implementations in this class. Each set of generic properties depends only on the standard specifications of the associated sub-architectures class.

At the prototyping level the verification may also be generic at some conditions. Combining several sub-architectures may result in large architectures which may be time consuming to model check directly. To avoid this issue using the presented BIP modeling approach, we ensured that checked safety properties for a given sub-architecture are true in any architecture prototype candidate containing it. To do that we identified all the potential interactions between the current sub-architecture and other ones, then we added incomplete interactions to the model before starting the verification. We assumed that sub-architectures may communicate using only signals, and thus we modelled incomplete interactions for only sub-architecture elements calling signal statements.

Notice that if we don't make this transformation we can't guarantee that checked safety properties will still hold for the next architecture prototype. Let us consider for instance a sub-architecture composed of two buffers of size 1 and 2 respectively. If we make the hypothesis that these buffers always perform an output when not empty, then the property ”maximal number of tokens stored in the sub-architecture is 2” is true. The property, however, is not true anymore if we consider an architecture prototype composed of a second sub-architecture performing outputs to second buffer.

Let us now consider a sub-architecture model for a channel composed of two one-data buffers A and B. For simplicity, and since we are interested here in the set of interactions, the shown product model in figure 7 is abstract, i.e., it does not contain the signals protocol details nor the cycle accurate behavioral details.

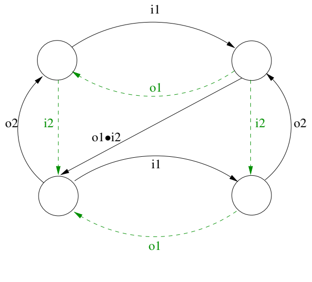


Fig. 7: Abstract model of the channel sub-architecture.

It is worth noticing that in the BIP model generation phase interactions $i1$, o1•i2 and $o2$ are produced which correspond to inputs of A, outputs from A to B, and outputs of B. The incomplete interactions which appear in the product model of figure 7, $i2$ and $o1$, are added in the model transformation phase.

The second model transformation is the transitions reduction. The objective is to reduce delta-step transitions into one meta-transition at the extent possible, hence losing all track of interleaving between processes on these transitions. Processes communicate through signals and are supposed to not use shared variables, so every sequence of transitions between two waiting states, with less than one output transition, is reduced into one meta-transition.

$$\frac{q_1 \xrightarrow{a_1} q_2 \ldots \xrightarrow{a_n} q_n}{q_1 \xrightarrow{a_1|a_2|\ldots|a_n} q_n}$$

Because the last output into a given signal determines its stored value, we excluded the sequences with several outputs in order to take into account the impact of processes interleaving on outputs during the verification phase.

Finally, in this phase the sub-architecture model is also reduced by removing incomplete interactions for a list of signals given by the designer, and which specifies those signals which are supposed to be useless for other sub-architectures.

VI. Experiments

We now present the different experiments we conducted on a counter, a fifo, an AMBA AHB, SRAM and a RISC CPU sub-architectures. Incomplete interactions which might be useful for new architecture prototype candidates were added in the sub-architectures models. LTL is a temporal logic where a formula is composed of atomic propositions, logical operators, and temporal operators. Currently we do not have a GUI interface for visualization purposes. Nevertheless, LTL properties associated to sub-architectures are inserted as LTL formula in the Promela file and may be easily identified at the top of the file following the syntax:

```
Ltl  <Idf1>  {<Subarch1_Formula1>}
Ltl  <Idf2>  {<Subarch1_Formula2>}
...
Ltl  <IdfN>  {<Subarch1_FormulaN>}
```

Other LTL properties as needed are inserted corresponding to the specification to check for that sub-architecture. The verification tool will then check them all one by one. To do that, each formula is translated to generate a never claim which is an automaton representing the negation of the formula so that the tool will seek to find a counter-example for it. At last, when a counter-example is found, it may be use by designer to debug and/or modify the sub-architecture behavior.

For the counter example, transitions corresponding to internal signals are cut since the functionality of those signals are not used by other sub-architectures.

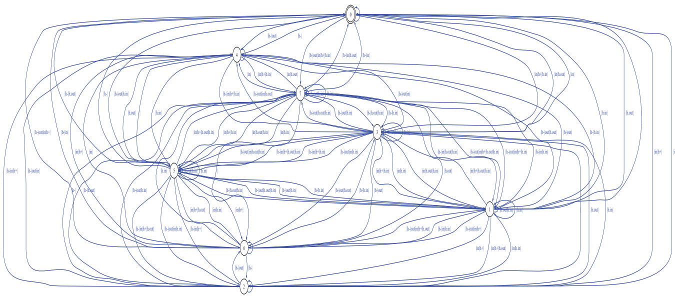


Fig. 8: Counter model before reduction

Figures 8 and 9 show that the cuttings operated on internal signals of a modulo four counter results in a reduced product model whose size is four times smaller than original one.

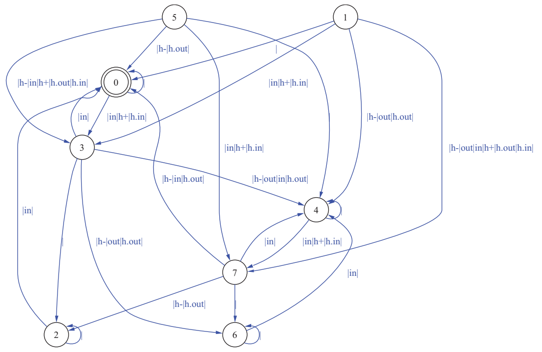


Fig. 9: Counter product model after reduction

The second example is an instruction set program for the Synopsys RISC CPU architecture The instruction set is a RISC one augmented with MMX-like instruction for DSP programs. It includes about forty arithmetic, logical, branch and SIMD MMX-like instructions. For this architecture, the BIP model construction took 20 seconds on a Xeon 3GHz 2GB bi-proc SMP i686 GNU/Linux server.

For this example we only used local checks implemented through assertions within the program. These C++ assertions are conditions on the values of sub-architectures variables. Each assertion is modelled by an error transition in the model construction if it is violated, and it is ignored if the corresponding condition holds. Assertions violations were successfully checked and a summary of the results is given in Table I.

The third example is a synchronous FIFO sub-architecture. Data can be written continuously until the FIFO is full. Data that was written first will be the first to be read. Subsequent readings will return data that has been written successively in time. The standard block scheme of a synchronous FIFO is depicted in Figure 10.

TABLE I: Depth-first verification results

Location	Explored states	Time (s)	Speed (states/s)	Error trace length
ICache	61	1	-	6
Bios	206356	1092	533	46678
Fetch 1	207841	1176	536	47017
Fetch 2	255201	1548	350	229614

TABLE II: Generic properties for synchronous FIFO

Property formula	Description
AG (full $\Longrightarrow \neg$ push)	If the fifo is full, FIFO should not have a write request.
AG (empty $\Longrightarrow \neg$ pop)	If the fifo is empty, FIFO should not have a read request.
AG (wr_idx $<$ depth $\wedge$ wr_idx $\geq$ 0)	There is a write overflow if the write pointer exceeds the max and the min of the fifo.
AG (rd_idx $<$ depth $\wedge$ rd_idx $\geq$ 0)	There is an overflow if the read pointer exceeds the max and the min of the fifo.
AG (rd_idx $\neq$ wr_idx $\Longrightarrow \neg$ full $\wedge \neg$ empty	The fifo cannot be full or empty if the pointers are different.
AG $\neg$ (full $\wedge$ empty)	The fifo cannot be full and empty at the same time.
AG (pop $\Longrightarrow \neg$ (tick_h $\wedge$ (wr_idx=rd_idx)) U full)	If a read request causes the two pointers to become equal in the next clock edge, then the fifo will be empty.
AG (push $\Longrightarrow \neg$ (tick_h $\wedge$ (wr_idx=rd_idx)) U empty)	If a write request causes the two pointers to become equal in the next clock edge, then the fifo will be empty.
AG ($\neg$ full $\wedge$ push $\wedge$ wr_idx $<$ depth $\Longrightarrow$ wrincr)	If the fifo is not full and there is a write request and the write pointer has not yet reached the maximum fifo size then the write pointer is incremented.
AG ($\neg$ empty $\wedge$ push $\wedge$ wr_idx = depth $\Longrightarrow$ X(wr_idx=0))	If the fifo is not full, and there is a write request, and the write pointer has already reached the maximum size of the fifo then write pointer is reset to value 0.
AG ($\neg$ empty $\wedge$ pop $\wedge$ rd_idx $<$ depth $\Longrightarrow$ rdincr)	If the fifo is not empty and there is a read re-quest and the read pointer has not yet reached the maximum fifo size then the read pointer is incremented.
AG ($\neg$ empty $\wedge$ pop $\wedge$ rd_idx = depth $\Longrightarrow$ X(rd_idx=0))	If the fifo is not empty, and there is a read re-quest, and the read pointer has already reached the maximum size of the fifo then read pointer is reset to value 0.
AG (full $\wedge$ push) $\Longrightarrow$ ($\neg$ wrincr U pop)	If the fifo is full and we have a write request; as long as we do not have a read request, then the write pointer does not increment.
AG (empty $\wedge$ pop) $\Longrightarrow$ ($\neg$ rdincr U push)	If the fifo is empty and we have a read request; as long as we do not have a write request, then the read pointer does not increment.
AG (full $\wedge$ pop) $\Longrightarrow$ ($\neg$ wrincr U push)	If the fifo is full and we have a read request; as long as we do not have a write request, then the write pointer does not increment.
AG (empty $\wedge$ push) $\Longrightarrow$ ($\neg$ wrincr U pop)	If the fifo is empty and we have a write request; as long as we do not have a read request, then the read pointer does not increment.

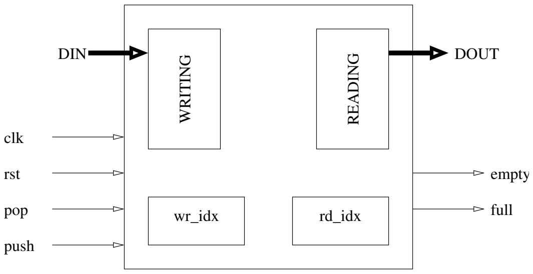


Fig. 10: Generic FIFO sub-architecture

FIFO sub-architectures include the following set of control signals and registers :

- push is a request signal for inserting data into the FIFO.
- pop is a request signal for extracting data from the FIFO.
- full is a flag indicating that FIFO is at its maximum capacity.
- empty is a flag indicating that FIFO has no valid data.
- wr_idx is a counter indicating where data will be stored in the FIFO. It is incremented through wrincr.
- rd_idx is a counter indicating where data will be read from the FIFO. It is incremented throug rdincr.
- The clk clock is the synchronous clock signal for the FIFO for both the read and write transactions, active on the positive edge of the clock.

FIFO specification states that, initially, rd_idx and wr_idx are set to value 0. The empty signal is set to value 1 and full remains at low state during this time, i.e. at value 0.

Read operations from the FIFO are not allowed when the FIFO is empty. On a write operation, wr_idx is incremented and empty is set to 0. Pointers are managed in a cyclic manner. When wr_idx reachs depth-1, a subsequent write operation will cause wr_idx to get back to value 0.

The same condition defines the transitions to full and empty states of the FIFO which is the equality between wr_idx and rd_idx. It is necessary to distinguish between the two transitions using the type of the operation. If a write operation is the origin of the equality then the transition is to the full state. Otherwise if a read operation is the origin of the equality then the transition is to the empty state.

According to the specifications, we are able to define the following set of generic properties that must be verified by any synchronous FIFO sub-architecture as shown in Table II.

The fourth example is the AMBA AHB bus sub-architecture. Several masters and slaves may connect to the bus, but only one master is allowed access at a time. The slave servicing the transfer depends on the address being read or written. AHB supports pipelining of data and address phases, slave waiting cycles, and the split and retry protocol. Figure 11 depicts the standard AHB scheme.

AMBA AHB specifications are hardware and operating system independent. Like the previous example, Table III summarizes the sub-architecture pre-defined generic properties.

Finally, we provide in Table IV the verification results for the SRAM memory component. Property P$_1$: AG (sramenqueued $\Longrightarrow \neg$ sramdone U sramdequeued) states that a request cannot be done before it is dequeued. Property P$_2$: AG (sramenqueuedaddr = sramcpuaddr $\wedge$ sramdata =

TABLE III: Generic properties for AMBA AHB

AG (HBUSREQ $\wedge \neg$ MASK $\Longrightarrow F$ HGRANT)	If master m requests the bus and m is not masked by the arbiter then m is eventually granted bus access.
AG (HBUSREQ $\Longrightarrow F$ HGRANT)	If master m requests the bus by asserting the signal HBUSREQ then it is eventually granted bus access by asserting the signal HGRANT.
AG (MASK $\Longrightarrow F$ HSPLIT)	The master that has been split by a slave always eventually recovers from the split.
$AG \neg$ MASK $\vee \neg$ HSPLIT	The master status cannot be marked as masked and recovered at the same time.
$AG \neg$ HGRANT1 $\vee \neg$ HGRANT2	Only one master at a time is granted access to the bus.
$AG \neg$ MASK $\wedge$ HBUSREQ $\wedge$ IDLE $\Longrightarrow$ GRANT	The highest priority master requesting the bus is to be granted the bus, provided the bus is idle and the master.
AG (GRANT $\wedge$ READY $\Longrightarrow$ HMASTER)	The granted master is to be given bus ownership when the last active slave has signaled READY, which indicates transfer completion.
AG (SPLIT $\wedge \neg$ READY $\wedge$ HMASTER $\Longrightarrow$ MASK)	The split master should be masked.
AG (READY $\wedge$ HSPLIT $\Longrightarrow \neg$ MASK)	The split master should be unmasked if the slave, that signaled the split, signals the end of that split using HSPLIT.
AG (BUSREQm $\Longrightarrow X$(GRANTm) $\vee$ F(HSPLITm)	The highest priority master is either granted immediately (if it was not masked), or is masked (which means it is waiting on a split transfer) and will be granted when the split transfer it is waiting on is followed by an HSPLIT signal.

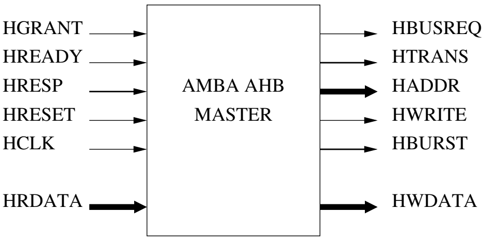


Fig. 11: AMBA AHB interface

sramcpudata) states that data read and the memory address referenced must be similar, and all the references are made in order. Property P_3 : AG (sramenqueuedaddr = sramaddr) states that memory address enqueued is equal to the committed request address.

TABLE IV: Verification results for the SRAM component

Location	automata states	automata transitions	Time
P_1	5739	$7 * 10^6$	24
P_2	10267	$3 * 10^5$	6
P_3	5710	$7 * 10^6$	60

These experiments show the feasibility of our approach, i.e., the ability to check safety properties for sub-architectures, while achieving good performance results on common architectures in term of automata size and verification time.

VII. Conclusion

In this paper we proposed a novel cycle accurate code-level verification approach to check safety properties of sub-architectures. In this approach we not only separately check correctness properties for identified sub-architecture programs, but also guarantee that these properties still hold in architecture prototype candidates comprising them. To do that we extended sub-architecture models using interaction-based transformations, which allow for generic verification of safety properties.

To our knowledge this is the first work which presents a prototyping framework that uses generic verifications of sub-architectures. The presented framework enables to check pre-defined generic safety properties for common classes of sub-architectures once at design time, without the need for additional time to redo the verification phase every time sub-architectures are involved in new architectures prototype candidates.

Furthermore, two reductions are operated on sub-architecture models to improve verification speed. The first one is the cutting of transitions corresponding to incomplete interactions if they cannot be used by new architecture prototypes according to designer point of view. The second one is the reduction of some sequences of transitions into single meta-transitions if such a reduction is neutral with regards the verification step. Finally, we have successfully applied the verification approach on four sub-architecture examples.

References

[1] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Pub., Hingham, MA, 2002.

[2] G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.

[3] K. L. McMillan, *Symbolic model checking*. Kluwer, 1993.

[4] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *SEFM '06*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 3–12.

[5] J. Queille and J. Sifakis, "Iterative methods for the analysis of petri nets," in *Application and Theory of Petri Nets, Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets, Stasbourg 23.-26. September 1980, Bad Honnef 28.-30. September 1981*, 1981, pp. 161–167. [Online]. Available: https://doi.org/10.1007/978-3-642-68353-4_27

[6] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '85. New York, NY, USA: ACM, 1985, pp. 97–107. [Online]. Available: http://doi.acm.org/10.1145/318593.318622

[7] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986. [Online]. Available: http://doi.acm.org/10.1145/5397.5399

[9] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[10] R. Alur and T. A. Henzinger, *Logics and models of real time: A survey*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 74–106. [Online]. Available: http://dx.doi.org/10.1007/BFb0031988

[11] S. Almagor, U. Boker, and O. Kupferman, *Discounting in LTL*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 424–439. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54862-8_37

[12] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[13] W. V. C. Dechsupa and A. Thongtak, “Compositional formal verification for business process models with heterogeneous notations using colored,” in *Lecture Notes in Engineering and Computer Science: Proceedings of the International MultiConference of Engineers and Computer Scientists 2019, IMECS 2019*, Hong Kong, 13-15 March 2019, pp. 565–570.

[14] B. Srongsil and W. Vatanawood, “Compositional verification of data invariants in promela using slicing technique,” in *Lecture Notes in Engineering and Computer Science: Proceedings of the International MultiConference of Engineers and Computer Scientists 2018, IMECS 2018*, Hong Kong, 14-16 March 2018, pp. 484–488.

[15] A. T. Kanut Boonroeangkaow and W. Vatanawood, “Formal modeling for persistence checking of signal transition graph specification with promela,” in *Lecture Notes in Engineering and Computer Science: Proceedings of the International MultiConference of Engineers and Computer Scientists 2017, IMECS 2017*, Hong Kong, 14-16 March 2017, pp. 161–165.

[16] H. K. K. Shirshendu Das, Shounak Chakraborty and K. L. Man, “Formal modelling and verification of compensating web transactions,” *IAENG Transactions on Electrical Engineering*, vol. 1, pp. 123–136, 2013.

[17] P. Herber, J. Fellmuth, and S. Glesner, “Model checking systemc designs using timed automata,” in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, ser. CODES+ISSS ’08. New York, NY, USA: ACM, 2008, pp. 131–136.

[18] I. Radojevic, Z. Salcic, and P. Roop, “Modelling heterogeneous embedded systems: from Esterel and SystemC to DFCharts,” *IEEE Design & Test of Computers*, vol. 23, no. 5, pp. 348–357, May 2006.

[19] A. Ismail, E. J. Lamia, Z. Abdelouahed, and N. Tarik, “The “behavior, interaction and priority” framework applied to systemc-based embedded systems,” in *13th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2016, Agadir, Morocco, November 29- December 2, 2016*, 2016.

[20] S. Campos, E. Clarke, W. Marrero, and M. Minea, “Verifying the performance of the pci local bus using symbolic techniques,” in *Computer Design: VLSI in Computers and Processors, 1995. ICCD ’95. Proceedings., 1995 IEEE International Conference on*, Oct 1995, pp. 72–78.

[21] D. Große, H. M. Le, and R. Drechsler, “Proving transaction and system-level properties of untimed systemc tlm designs,” in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, July 2010, pp. 113–122.