Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

A Methodology for Construction of Composable Formal Models from SystemC in BIP

Ismail Assayad, Joseph Sifakis

Verimag Research Report n^o TR-2009-9

July 2009

A Methodology for Construction of Composable Formal Models from SystemC in BIP

Ismail Assayad, Joseph Sifakis

July 2009

Abstract

We present an approach that is intended to facilitate the integration of domain-specific languages and heterogeneous systems on a semantic level by mapping language constructs to concepts in an asynchronous formalism capable of describing heterogeneous systems. The approach is centered around the use of extended automata to model the behaviors, interactions to model the glue between the transitions of these automata, and finally priorities to choose amongst possible interactions. Using this approach, we present a methodology for modelling SystemC a widely used language for system-level modelling of heterogeneous system-on-chip processes. The methodology incrementally produces (a) readable results opening the way for enhanced verification, (b) models with understandable execution semantics, (c) composable formal models allowing consistent integration with potential domain-specific sub-systems. We show the feasibility of the approach through the simulation and verification of assertion and deadlock properties on a realistic complex system.

Keywords: SystemC, BIP(Behavior-Interaction-Priority), PINAPA

Reviewers:

How to cite this report:

```
@techreport { ,
title = { A Methodology for Construction of Composable Formal Models from SystemC in BIP},
authors = { Ismail Assayad, Joseph Sifakis },
institution = {  Verimag Research Report },
number = {TR-2009-9},
year = { },
note = { }
}
```

1 Introduction

There are scenarios where a combination of different sub-systems from different domains is useful. It is believed that many small, loosely-coupled models are easier to handle and improve readability in comparison with a single, monolithic model [War07]. Therefore, modelling such complex systems usually necessitates the use of multiple domain specific languages (eg. SystemC [sys] and Lustre [HLR92]) resulting in several domain-specific models, each describing one aspect of an application (e.g., sensors, user interfaces, etc) [HCW07]. When designing such multi-domain systems, one of the main challenges is to guarantee the correctness of the implementation. This raises the need for the analysis of their integration [KKR+06]. Most designers use a design approach, where one has an informal idea of how the design should behave, define processes specifications, implement and assemble the processes into a program. Therefore the analysis of these heterogeneous systems can be especially difficult for designs that are composed of concurrent processes with lot of interactions.

Integration issues Although many domain-specific languages have exposed APIs through programming libraries, and can be accessed from other programming languages without breaking the flow of execution, there are many issues for their integration:

- They are less comprehensive since their execution semantics is not exhibited at the designer's view.
- It is difficult to understand how their modelling constructs fit into the overall system description.
- Standard programming approaches do not properly support expressing semantic relationships and interdependencies between distinct modelling languages.

For instance, SystemC simulation engine introduces the important notion of delta-cycle as the fundamental simulation unit. In fact, a simulation procedure is a sequence of delta cycles. Delta cycles interactions are abstracted from the designers modelling view. Such an abstraction is intended to provide more coding simplicity for the designers assuming that all these interactions should work correctly. However, this is probably the most error-prone part of SystemC modelling, as analysis are difficult without considering what goes on inside delta-cycle updates. Furthermore, these interactions might be complicated or not understood by the designers.

Common semantics formalism Our idea is to use a formalism based on extended automata as a common denominator for the semantics of multiple models. To be useful the model must be composable and must support a set of abstract modelling concepts that are generic enough to be applicable to a wide range of domains. We claim that this formalism provides the key semantics concepts for modelling multi-domain and heterogeneous systems description and which can be leveraged for automatic analysis and verification.

Along this line we can:

- Compile designs in domain-specific languages to an easily understandable behavior and interactions model,
- Model and compose different systems belonging to different domain of applications which are often thought of as languages with very specific goals in design and implementation.

Unified semantics encompassing heterogeneity in systems design have been proposed in [EJL+03, BGK+06, BWH+03, Arb05]. Nevertheless, in these works unification is achieved by reduction to a common low-level semantic model. We need a framework which is not just a disjoint union of submodels, but one which expresses and preserves properties during model composition and supports meaningful analysis and transformations across multi-domain model boundaries.

This paper In this paper, we show how to brought to light the execution semantics of SystemC simulator using automata extended with (a) interactions between these automata, (b) priority rules describing scheduling policies for these interactions.

In a previous works [MMMC05, MMMC06], authors described a complete chain from SystemC to a synchronous formalism. It is based upon a systematic encoding of SystemC processes into a flavor of synchronous automata. In such a case, SystemC processes are encoded one by one into automata, communicating with an additional automaton that explicitly encodes the scheduler specification. The automata corresponding to the user processes are produced specifically to communicate with this scheduler automaton, using additional synchronous signals and the instantaneous

dialogue mechanism available in a pure synchronous semantics. However, the encoding into a synchronous formalism renders manual reading difficult, and requires significant amount of additional synchronizations to reflect the semantics of SystemC.

Unlike these works, our approach encodes SystemC models into an asynchronous formalism and has the advantage of producing

- *Readable results*. The global model is obtained by progressively composing its components while preserving their structure and data through translation. That is, it is possible to identify in the global model all its components with their behaviors and interactions, so that designer can easily track counter examples when checking properties for assertions or deadlocks for instance.
- *Composable models*. Instead of modelling processes to communicate with a scheduler, signals and other channels, our approach defines models with an explicit semantics, i.e., where the interplays of schedulings and communication are exhibited at the processes-level.Therefore, independent (generated) models are thus composable. Moreover, (direct) models translation into BIP [BBS06] allows for compositional verification.

The paper is structured as follows. Section 2 presents the abstract syntax of SystemC. Section 3 describes the modelling methodology of SystemC processes. The model construction is explained in Section 3.1. We present the System-CBIP tool and experimental results for a complex system in Sections 4 and 5 and conclude in Section 6.

2 SystemC

SystemC becomes nowadays a popular language for modelling complex hardware systems. Compared with other hardware description languages, SystemC is more feasible for designing large-scaled complex systems and modelling high level behaviors. It comes with many pre-defined constructs in its syntax, like channels, modules, clocks, etc., which make the compositional system design much easier.

SystemC is a set of C++ classes that allows modelling systems at different levels of abstraction from system behavioral to register transfer level. The SystemC Class library provides the constructs needed to model system timings, concurrency and reactivity. This Class library supports Modules, ports, Signals, Processes, data types, Clocks, Waiting and watching.

2.1 Delta cycle

SystemC is equipped with a simulation engine which allows to simulate a model once it is designed. SystemC simulation engine introduces the important notion of delta-cycle as the fundamental simulation unit. In fact, a simulation procedure can be seen as a sequence of delta cycles and all the interactions within a delta cycle are abstracted from the modelling perspective.

SystemC uses two-dimension time model: physical time and delta cycle. Physical time is represented by integer counters. Delta cycle is used to order the execution within the simulation cycle. The simulation cycles of SystemC consist of two phases evaluate and update. More than one delta cycle may occur in a particular time. Delta cycle is used to separate the evaluate phase and the update phase. When no more processes ready for evaluations, the update phase is performed. Therefore updates are strongly synchronized between processes.

3 SystemC modelling methodology

A simulation can be seen as a sequence of delta cycles whose interactions within the update phases are abstracted from the programmers modelling view.

To express the semantics of the delta cycles, for each process we distinguish between stable states, i.e., states corresponding to wait statements, and transient states corresponding to the beginning of all other statements. At the end of each delta cycle, a global synchronization between the processes (figures 2 and 1) allows to perform updates for processes needing data updates. This is represented by the label S and involves update of the new values computed during the delta cycle.

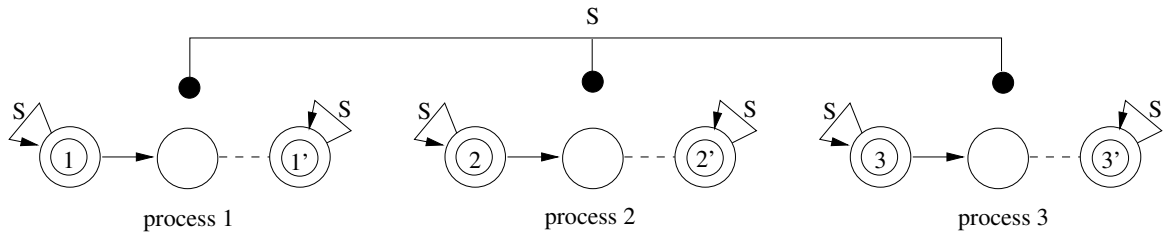

Figure 1: Global synchronization

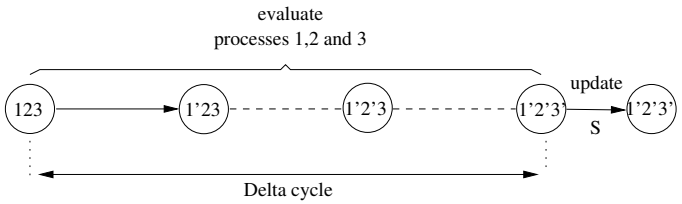

Figure 2: Delta cycle

As an example, for each process, two values of each signal s are needed: one is used to store the value calculated during the delta cycle. The second holds the value of the signal after the update phase. These values are represented by two variables $s.now$ and $s.next$ for each signal (figure 3).

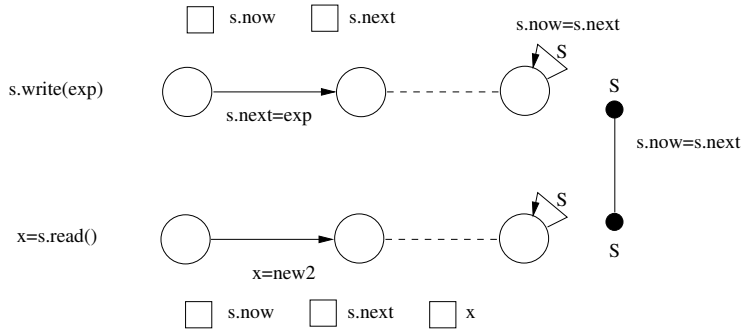

Figure 3: Updating a signal state

For ease of presentation, we do not consider any predefined channels or signals in presented modelling (eg. fifos, event queues, etc) but rather taking into account their state updates and event-driven interactions semantics. For instance, a fifo buffer is defined as a channel with two read and write actions and two internal read and write events, and modules connected to the fifo are allowed to call the channel methods but they are not aware of the existence of the channel events.

3.1 Modelling the behavior of processes

The idea is to model the behavior of processes by automata extended with data. We show by reasoning on the structure of the programs describing processes how to obtain compositionally an extended automaton representing the behavior of a process. That is for each atomic statement we show how to obtain the corresponding extended automaton; then we show how to compose these extended automata to obtain the global extended automaton.

Extended automata An extended automaton is a tuple $(L, X, Q, \longrightarrow)$ (figure 4) where:

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

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

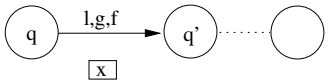

Figure 4: Extended automaton used for modelling statements

Hereinafter, we describe how we define the model of the main SystemC processes statements.

Statement For each statement on a set of variables we associate an automaton with an initial and a final control states (figure 5).

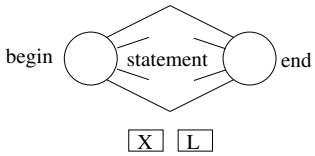

Figure 5: Statement

Control structure For a sequence of two statements $stmt1$ and $stmt2$, the corresponding automaton is obtained by composition as shown in figure 6.

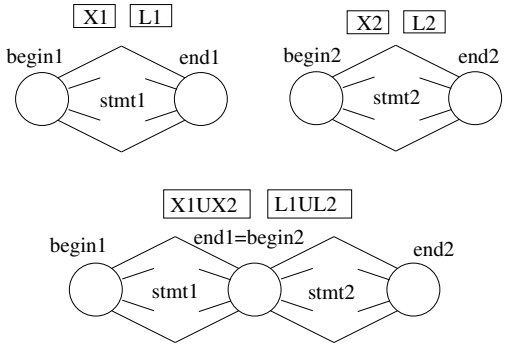

Figure 6: Sequence of two statements

For a conditional statement $if\ c\ then\ stmt1\ else\ stmt2$, the corresponding automaton is obtained as shown in figure 7.

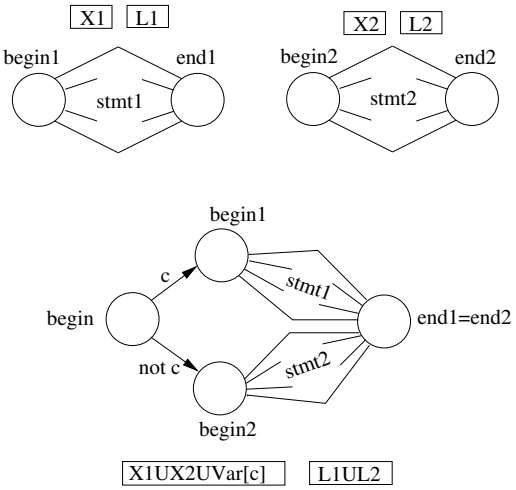

Figure 7: if

For a loop statement $while\ c\ do\ stmt$, the corresponding automaton is obtained as shown in figure 8.

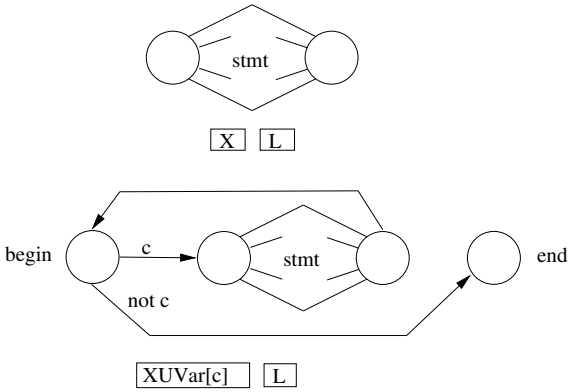

Figure 8: While

A function $f(p)\{stmts\}$ model:

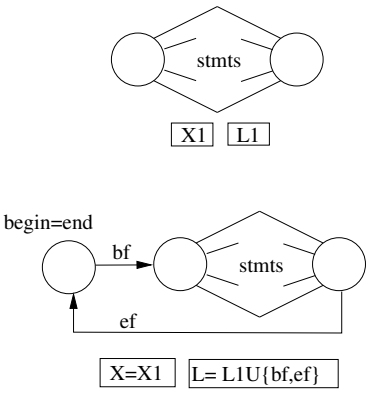

Figure 9: f(p)

Finally, it is worth noticing that when composing statements of a process, we can reduce the automata between two wait statements to one automata with all the transitions of the intermediate automata are collapsed to one single transition.

3.2 Basic statements model

A process may wait for an event e. The $wait(e)$ statement model is shown in the figure 10.

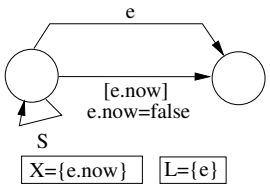

Figure 10: wait(e)

An immediate event notification $e.notify()$ statement notifies all the processes which are waiting on that event. This notification will result in running all the statements in between two continuous $wait$ within these processes. At the occurrence of this notification all other pending timed or delayed notifications are deactivated $e.active = false$ (see the corresponding statements models in figures 13 and 12). The $e.notify()$ statement model is shown in figure 13.

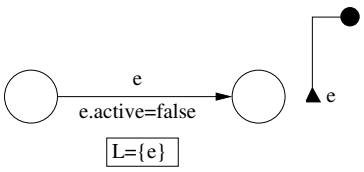

Figure 11: e.notify()

Timed notifications on an event e $e.notify(t)$ are used to emit notifications at a later time t which is relative to the current time. This time is stored by the corresponding time variable: $e.time = t$. This variable will be decremented at each tick. The event notification will then be deactivated after the notification of the waiting processes during the updates of the global synchronization S. That is when $e.now$ and $e.event$ are updated for each process waiting for e. The $e.notify(t)$ statement model is shown in figure 12.

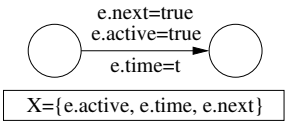

Figure 12: e.notify(t)

The delayed notification statement emits the notification in the update phase at the end of the current delta cycle. Thus a delayed notification occurs after all the immediate notifications. The $e.notify(0)$ statement model is shown in figure 13.

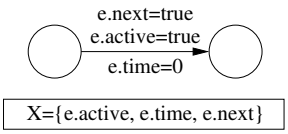

Figure 13: e.notify(0)

A process may also wait during a time t relative to the current time. The $wait(t)$ statement model is shown in figure 14.

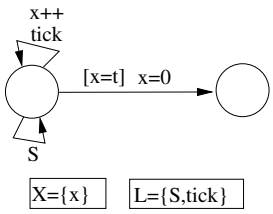

Figure 14: wait(t)

Waiting for a zero time means that the process waits for the next delta cycle. The statement model is depicted in figure 15.

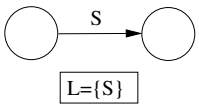

Figure 15: wait(0)

Time interaction When all processes are in a stable state and if there are no updates to do, simulation time ($tick$) is advanced. There are no updates to do if there is no active time or event variable $e.active$ to update at the current time. If the simulation time is exceeded, the processes reach the finish states.

Timeouts A process may also wait for an event e with a timeout t. If an immediate notification e or a delayed one $[e.now]$ is received before time t, the timer is reset along with notification. This reset is done by simply setting to zero the value of the timer variable x.

The $wait(e,t)$ statement model:

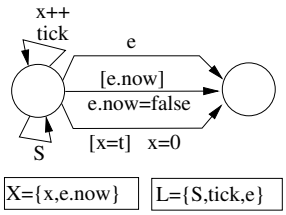

Figure 16: wait(e,t)

A SystemC signal s is persistent and is associated to a pair of variables $s.next$, $s.now$ and to an event $s.event$ to notify signal value changes. For a signal write statement, if the value exp being written is different than the current value $s.now$, the event $s.event$ is emitted in the update phase.

The $s.write(exp)$ statement model:

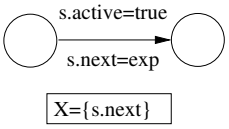

Figure 17: s.write(exp)

The $x = s.read()$ statement model:

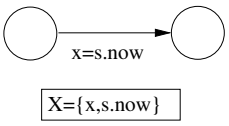

Figure 18: x=s.read()

A process may wait for a set of events notifications and timeouts defined by the semantics of the sensitivity of the process. The $wait()$ statement model is given in figure 19.

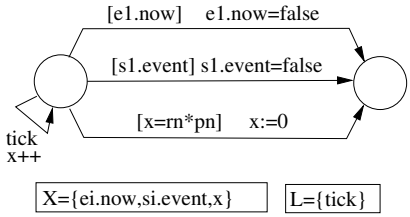

Figure 19: wait()

A process may change the events which will trigger it. By calling $next\_trigger(e)$, the process will be triggered by e. A timeout may also be used to trigger the process after that timeout. The $next\_trigger(e,t)$ statement model is shown in figure 20.

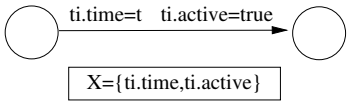

Figure 20: next_trigger(e,t)

The $f(x)$ statement model:

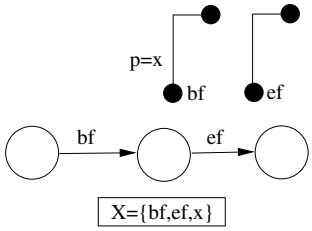

Figure 21: f(x)

Interactions During the interactions of process automata, variables are updated as follows.

- (a) For any label e introduce a broadcast connector from the process doing $e.notify()$ to the process doing $wait(e)$, and deactivate the other subsequent notifications $e.active = false$
- (b) At each stable state add loops labelled by $tick$ and for all active time variables $y.active = true$ do $y.time = y.time - 1$.
- (c) Strong synchronization of all the processes on S with transfer of data [1]
  1. For any pair of signal variables $s.next$, $s.now$ test if the signal is active $s.active = true$, do: $s.now = s.next$ activate the signal event: $s.event = true$, transfer $s.now$, $s.event$.
  2. For any pair of event variables $e.next$, $e.now$, test if the event is active $e.active = true$ and do: $e.now = e.next$, deactivate the event $e.active = false$, transfer $e.now$ to all processes.
  3. For each active time variable ($y.active = true$) test $y.time = 0$ and then update the variable value: $y.now = y.next$, deactivate the time variable $y.active = false$, transfer $y.now$ to all processes.

Ordering of the interactions In case of multiple event notifications on the same event, only the earliest notification to occur survives. Therefore (c.2) must be done before (c.3) for variables corresponding to delayed notifications.

Priorities When composing statements of a process, additional transitions corresponding to timeouts and events of the trigger and reset statements are added from each stable state to the end state of the sensitivity automaton or from the begin state to the end state. These transitions have higher priority than S and $tick$:

- Trigger timeout. At each stable state the transition

$$\xrightarrow{[ti.now]\ ti.now=false}$$

  to the end state of the process sensitivity automaton which has the guard the trigger time variable $ti.now$, deactivates the trigger event or timeout $ti.now = false$ for the process.

[1] It allows processes which need updates to transfer data when no more processes are ready to run in the current delta-cycle.

- Trigger event. At the begin state of the sensitivity automaton, the transition

$$\xrightarrow{[e.now]\ e.now=false}$$

whose guard is the trigger event variable $e.now$ has higher priority.

- Synchronous reset with respect to the clock. For each stable state, the transition

$$\xrightarrow{[si.event=true,si.now=v,x=r*n]\ si.event=false\ x=0}$$

which has as a guard a reset signal and a clock ticks timer, and deactivates these reset signal and timer has higher priority.

The priority of transitions advancing time $\xrightarrow{tick}$ also becomes higher than the one of S transitions when there is no active event or signal to update.

3.3 Modules

Processes and functions are modelled as components.

To exploit the modularity of hierarchical, modules are defined as components witch contain their own program variables, components containing extended automata of the functions, processes, and other components corresponding to other modules.

To avoid usages which result in a breakdown of the semantics of components, we export interactions of child components as interactions of the module (eg. figure 22).

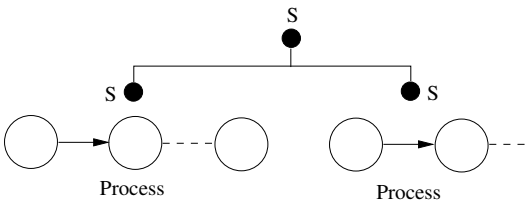

Figure 22: Exporting the interaction of two processes on S

4 Implementation

4.1 Parsing SystemC

The list of process instances of the program and the architecture bindings may be accessed using the SystemC library after the execution of the elaboration phase of the engine.

The idea is thus to link these processes to their corresponding GCC abstract syntax tree (AST) and parse the processes statements. For this purpose the PINAPA tool is used as this linking is done for processes and also for some SystemC primitive trees which are linked to SystemC objects.

While parsing the ASTs of process instances, the rules for constructing formal SystemC models presented before are applied to incrementally construct the processes models. For instance, when a wait node is recognized its model is built and added on the fly to the process instance automaton. Like processes, non-recursive functions are also modelled using one separate model per function instance. Then interactions are used to model the function calls and returns.

4.2 Tool flow overview

The complete tool-flow is depicted in figure 23. Using GCC [St] and PINAPA [MMMC05], we generate the AST trees of processes, functions and statements of the SystemC program and associate them to SystemC objects.

The exploration of these trees is then achieved to generate the SystemC model in two steps.

The first step ”Parsing” produces the structure and transitions of the automata of the program. The structure consists of the control flow of processes and the components hierarchy. Transitions are pairs of annotations: the primitives and operands of the statements. The operands are either other primitives or program variables which might be local to the process component or global variables whose declaration are defined in a parent component.

The second step ”Model generation” extends the preceeding model with (a) semantics variables, guards, and functions on these variables (b) labels (c) broadcast connectors and strong synchronization on these labels with corresponding transfer of semantics variables data.

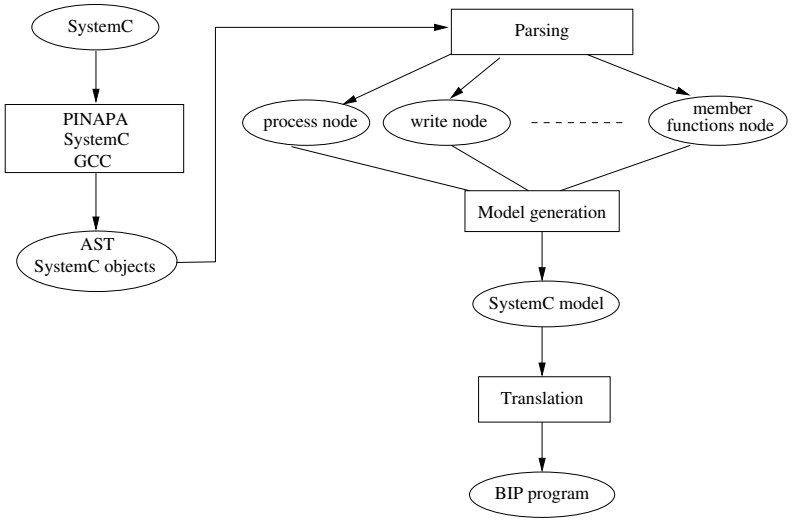

Figure 23: SystemCBIP Tool-flow

Finally, a direct translation produces the BIP program.

4.3 Translation to BIP

The BIP language [BBS06] supports a methodology for building components from:

- atomic components, a class of components with behavior specified as a set of transitions and having empty interaction and priority layers. Triggers of transitions include ports which are action names used for synchronization.
- connectors used to specify possible interaction patterns between ports of atomic components.
- priority relations used to select amongst possible interactions according to conditions depending on the state of the integrated atomic components.

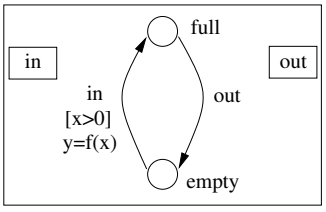

Figure 24: BIP atomic component

Figure 24 shows an atomic reactive component with two ports in, out, variables x, y, and control states empty, full. At control state empty, the transition labeled in is possible if 0 ¡ x. When an interaction through in takes place, the variable

x is eventually modified and a new value for y is computed. From control state full, the transition labeled out can occur. The omission of guard and function for this transition means that the associated guard is true and the internal computation microstep is empty. The syntax for atomic components in BIP is the following:

```
atom::=
component component id
  port port id+
  [data type id data id+]
  behavior
   {state state id
     {on port id [provided guard]
     [do statement] to state id}+}+
  end
end
```

Listing 1: Syntax of BIP component

That is, an atomic component consists of a declaration followed by the definition of its behavior. Declaration consists of ports and data. Ports are identifiers and for data, basic C types can be used. In the behavior, guard and statement are C expressions and statements respectively. We assume that these are adequately restricted to respect the atomicity assumption for transitions e.g. no side effects, guaranteed termination.

Behavior is defined by a set of transitions. The keyword state is followed by a control state and the list of outgoing transitions from this state. Each transition is labelled by a port identifier followed by its guard, function and a target state.

The BIP description of the reactive component of figure 24 is:

```
component Reactive
  port in, out
  data int x, y
  behavior
    state empty
      on in provided 0 < x do y:=f(x) to full
    state full
      on out to empty
   end
end
```

Listing 2: Program of the component of figure 24

Once the SystemC model of the program is generated by the tool as shown in figure 23, a straightforward translation is achieved to obtain the equivalent BIP program by producing for each component its ports, data, automaton states and transitions guards and functions, and finally producing inter-components connectors and priorities.

4.4 Benchmarks

Our test models consist of a set of 40 SystemC programs that allow for testing various SystemC primitives, different processes synchronization, useless notification, accessing modules data members, different processes hierarchy, etc.

Since the generated BIP programs are executable, the simple possibility to execute the constructed SystemC models and compare the resulting traces with the original SystemC program shows a confidence in the validity of the approach.

4.4.1 Simple Transmitter/Receiver example

The example is composed of two modules $m\_emitter\_name$ and $m\_receiver\_name$ including two thread processes named $emitter\_process$ and $receiver\_process$ respectively. The modules are connected to a boolean signal $signal\_0$ through output SystemC port my_ouyput and input port my_input respectively.

The transmitter writes a sequence of values 1 and 0 each 10 time units on the signal. The receiver is sensitive on the event issued by the signal and reads the signal value at each occurrence of this event.

The figure 3 shows the automaton generated for the transmitter: the set of labels of this automaton are indicated by the list of the BIP ports of the component, the variables are the set of BIP data.

```
component TOP__m_emitter_name__emitter_process
  port complete write_TOP__signal_0
  port complete not_le
  port complete le
```

```
  port complete truth_not
  port complete wait_time
  port incomplete syn
  port incomplete tick
  port complete plus_expr
  port complete end_while
  data bool val
  data int y
  data int TOP__signal_0_now
  data int TOP__signal_0_next
  data bool TOP__signal_0_event
  data bool TOP__signal_0_active
  data double x
  behavior initial do
      val = 0;
      y = 0;
      TOP__signal_0_event = false;
      TOP__signal_0_active = false;
      x = 0;
      to s2
      state  s2
         on  write_TOP__signal_0 do
             TOP__signal_0_next=1;
             TOP__signal_0_active=true;
         to  s3
      state  s3
         on  not_le provided (y > 9) do
         to  s9
         on  le provided (y <= 9) do
         to  s4
      state  s4
         on  truth_not do
             val = !val;
         to  s5
      state  s5
         on  write_TOP__signal_0 do
             TOP__signal_0_next=val;
             TOP__signal_0_active=true;
         to  s6
      state  s6
         on  tick do
             x=x+1;
         to  s6
         on  syn do
         to  s6
         on  wait_time provided (x=10) do
             x=0;
         to  s7
      state  s7
         on  plus_expr do
             y=y+1;
         to  s8
      state  s8
         on  end_while do
         to  s3
  end
end
```

Listing 3: Transmitter component

The following priorities ensure that time advances through interaction $tick$ by giving it more priority than the synchronization S between $emitter\_process$ and $receiver\_process$ whenever there are no variables to update during this synchronization.

```
priority
S_tick_1
if (emitter_process.TOP__signal_0_active)
  tick : receiver_process.tick,
```

```
        emitter_process.tick <
  S : receiver_process.S,
      emitter_process.S
S_tick_2
if (!(emitter_process.TOP__signal_0_active))
  S : receiver_process.S,
      emitter_process.S <
  tick : receiver_process.tick,
        emitter_process.tick
```

Listing 4: Priority of interactions S and $tick$

5 Realistic RISC CPU ISS

This example is an instruction set simulator which demonstrates a RISC CPU design using Synopsys's SystemC(TM) class library provided by Synopsys Inc.

The CPU itself is modeled using SystemC. The CPU reads in assembly program and execute it and write the result back to registers/data memory. The instruction set is defined based on commercial RISC processor together with MMX-like instruction for DSP program. It consists of a set of more than 39 instructions arithmetic, logical, branch, floating point and SIMD MMX-like instructions.

The CPU example structure is shown in figure 25. The generation of the BIP program for each of the different cases takes around 20 seconds on a Xeon 3GHz 2GB bi-proc SMP i686 GNU/Linux machine. The body of the produced C++ code has more than 48 thousands of C++ code lines.

It is possible to check a set of static and run-time properties on the system during either the code generation or during the exploration of the generated formal model. The static properties concerns the bindings of the architecture while the run-time ones are either the SystemC model assertions, execution deadlocks or delta-updates anomalies such as multiple accesses in the same cycle to SystemC signals.

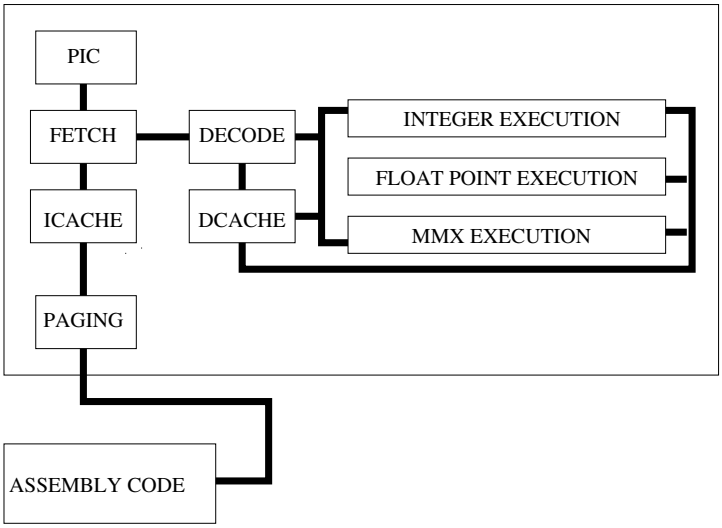

Figure 25: CPU architecture

Model properties (a), (b), (c) and (d) have been successfully checked by the verification tool using a depth-first exploration of model state space as shown in figure 1. Global path traces are given for the counter-examples which lead to the violation of the properties, so that the user may track the location of the error, the values of the variables when the error state is reached, and the path to the error state. For instance, for the case (a) the error path is "assign,decl,decl,decl,ne_expr,err" which indicates that the error happens starting from the ICache program after the first foor instructions, that is in the C assertion on variable j. This assertion is violated and the actual value of variable j

Property	location	states number	time (s)	speed (states/s)	Global path size
Assertion (a)	ICache	61	1	-	6
Assertion (b)	Bios	206356	1092	533	46678
Assertion (c)	Fetch	207841	1176	533	47017
Deadlock (d)	Fetch	255201	1548	350	229614
No error (e)	-	-	2 days	66	-
Bindings (f)	-	-	20	-	-

Table 1: Verification results for the example of figure 25 using depth-first exploration

is given in the last ICache state produced by the verification tool. The verification of the model (e) of the CPU which contains no error and has been stopped after 2 days of exploration.

Finally, the property (f) is a structural property concerning the bindings of the SystemC components. Any incomplete binding is signaled during code generation with the name of the corresponding component and an enumeration of the existing partial bindings.

6 Conclusion

We proposed a methodology to automatically create composable models from SystemC models using an asynchronous formalism. The methodology models SystemC processes using extended automata, interactions between transitions of these automata and priorities to choose amongst several possible interactions.

These formal models allow designers to verify the correctness of their design through the verification of its properties and track counter-examples to early ensure the quality of the design.

Our methodology captures delta-step and delta-cycle level behaviors of SystemC processes, and has the advantages of producing readable and composable models.

The methodology can be applied to building global models of heterogeneous systems, and we believe that it is convenient for the need to compositionally integrating models of multi-domain specific languages. Currently, we are applying it to Lustre.

We have developed the SystemCBIP compiler which produces BIP components starting from the GCC abstract syntax tree provided by PINAPA for SystemC programs. We have successfully simulated a benchmark of SystemC programs using our approach and have shown its scalability through the modelling and verification of a realistic complex CPU.

Acknowledgment

The authors thank M. Moy and A. Basu for constructive remarks.

References

[Arb05] Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *Sci. Comput. Program.*, 55(1-3):3–52, 2005. 1

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM '06*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society. 1, 4.3

[BGK+06] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema. Developing applications using model-driven design environments. *Computer*, 39(2):33, 2006. 1

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003. 1

[EJL+03] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia R. Sachs, and Yuhong Xiong. Taming heterogeneity?the ptolemy approach. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1):127–144, January 2003. 1

[HCW07] Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wasowski. *Guided Development with Multiple Domain-Specific Languages*. Springer Berlin / Heidelberg, 2007. 1

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992. 1

[KKR+06] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 43–46, New York, NY, USA, 2006. ACM. 1

[MMMC05] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip. In *EMSOFT*, September 2005. 1, 4.2

[MMMC06] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 2006. special issue on SystemC-based systems. 1

[St] Richard M. Stallman and the GCC developer community. *GNU Compiler Collection Internals*. Free Software Fundation. `http://gcc.gnu.org/onlinedocs/gccint/`. 4.2

[sys] SystemC. http://www.systemc.org. 1

[War07] Jos Warmer. *A Model Driven Software Factory Using Domain Specific Languages*. Springer Berlin / Heidelberg, 2007. 1