Fundamenta Informaticae XXI (2001) 1001–1025 1001
IOS Press

Formal Analysis of SystemC Designs in Process Algebra

Hossein Hojjat
Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

MohammadReza Mousavi
Eindhoven University of Technology, Eindhoven, The Netherlands

Marjan Sirjani
University of Tehran, Tehran, Iran
Reykjavik University, Reykjavik, Iceland

Abstract. SystemC is an IEEE standard system-level language used in hardware/software co-design and has been widely adopted in the industry. This paper describes a formal approach to verifying SystemC designs by providing a mapping to the process algebra mCRL2. Our mapping formalizes both the simulation semantics as well as exhaustive state-space exploration of SystemC designs. By exploiting the existing reduction techniques of mCRL2 and also its model-checking tools, we efficiently locate the race conditions in a system and resolve them. A tool is implemented to automatically perform the proposed mapping. This mapping and the implemented tool enabled us to exploit process-algebraic verification techniques to analyze a number of case-studies, including the formal analysis of a single-cycle and a pipelined MIPS processor specified in SystemC.

Keywords: SystemC, Process Algebra, Formal Verification, mCRL2

1. Introduction

A growing trend in the microelectronics field is to integrate a complete and sophisticated system on a single chip (System on Chip). One of the most challenging requisites of the SoC approach is to define a description language which is capable of specifying the whole design at a higher level of abstraction. At this level of abstraction, usually known as system level design [22], the main concern is to simultaneously and homogeneously describe both hardware and software subsystems. Among the different languages for

Address for correspondence: Department of Computer Science, TU/Eindhoven, P.O. Box 513, Eindhoven, NL-5600MB, The Netherlands

system level design, SystemC has become very popular and widely adopted by several leading companies in this field. Moreover, SystemC 2.1 has been standardized as IEEE standard 1666 [1].

SystemC is a library of classes and macros implemented in C++ that are to be used in the system level specification. As SystemC is an add-on to the core language, all the features of C++ are available in modeling but the designer has to adhere to the particular style given by SystemC. The set of data types in SystemC contains the necessary elements for system level designs, such as bits, bit vectors, four state logic (low, high, undefined, tri-state), and data types for fixed-point arithmetic of arbitrary size. The notion of concurrency is realized by introducing processes. Concurrent processes are scheduled in a non-preemptive fashion. A process may suspend itself on a particular event, after the occurrence of which it will be ready to resume. The sensitivity of processes to the events can be static or dynamic. The communication of different components is represented in SystemC by the concepts of channels and ports.

One of the goals of SystemC is to provide verification at higher levels of abstraction. Several attempts have been made in this direction [12, 21, 23, 25, 26, 35] but this aspect of the language is still in its early stages [37].

In this paper, we describe a mapping from SystemC programs into mCRL2 [14], a process algebra enhanced with data types, which provides a common framework for analyzing and verifying systems. By translating the SystemC code to mCRL2 we can carry out various kinds of analysis using its backbone toolset. First of all the μ-calculus model-checker of mCRL2 makes it possible to verify the safety and liveness properties of the system. Particularly using model-checking we can ensure the absence of race conditions. For this purpose, we check a temporal formula to the effect that no two conflicting assignments to a signal or variable are performed (in two paths) during a delta cycle. To prove that the race conditions do not exist in the model we can alternatively use the confluence checker tool. We hide all the irrelevant actions which do not have an impact on the value of the desired output. Under this setting, we show the absence of race condition by proving that the system is confluent.

We implemented a tool that made this translation automatic. The translation and the implemented tool enabled us to verify a number of case studies. These case studies range from simple specifications of combinatorial and sequential circuits (e.g., adders, flip-flops) to a simplified single-cycle (comprising about 1000 lines of SystemC code) MIPS processor to a rather complicated pipelined MIPS processor (comprising 2500 lines of SystemC code).

Related Work. The authors of [12] propose a model-checking technique for a subset of SystemC, comprising fundamental constructs of sequential circuits. Our work improves upon [12] by treating a much larger subset of SystemC and supporting modal μ-calculus as the specification language.

In [21], a translation from SystemC to an extension of Petri Nets is defined. Their translation requires an upper bound on the size of dynamic constructs, which is not required in our approach. However, [21] supports quantitative time, which we do not support. We plan to extend our translation with quantitative time since it can be modeled using mCRL2.

In [23], SystemC programs are partitioned into software and hardware and are verified separately. This separation is not necessary in our case, since we try to transform both hardware and software parts into mCRL2 at the same time.

In [25] a special process algebra is introduced for SystemC designs. This process algebra is used only for formalizing SystemC. For verification, the code is translated into Promela, the input language of Spin [20]. In our approach, we benefit from the existing tool-set for mCRL2 and thus, provide a single framework both for formal semantics and formal verification of SystemC.

The authors of [26] have described a toolbox, called LusSy, for analyzing the transaction level models written in SystemC. An automaton-based intermediate language (with guards, messages and assignments on transitions) is used between SystemC and the language of the back-end verifier tool. This language is also used for describing the scheduler. One of the goals of LusSy is that "the user should not have to learn a timed logic", so the properties are expressed in terms of assertions. Although this makes the verification of safety problems more convenient, this technique has its limitations when it comes to liveness properties. To overcome the limitations, LusSy additionally supports some general liveness properties such as "a process cannot terminate". In comparison with [26], our approach has the advantage of allowing for modal μ-calculus for property specification which is more expressive and can code other (less expressive) temporal logics such as PSL as syntactic sugar. In [35], the intermediate language of LusSy is translated into Promela, the input language of SPIN. We are not aware of an implementation of this translation so that we can compare its performance with our implementation.

In [29], the authors present an approach for model-driven testing of SystemC designs. For this purpose, the modeler formally specifies the intended specification of the system in the AsmL [10]. The result is accompanied with another AsmL description, which explains the discrete event semantics of the simulator. Spec Explorer [8] uses the two AsmL code to generate test cases for the system. The conformance of the test cases with the implementation is checked subsequently. As the back-end tool supports C# programming language, an interface is written between C# and the actual SystemC code. [29] formalizes the scheduler in AsmL and the rest of SystemC constructs are directly called through a wrapper. In our approach, however, we formalize the whole design in the process-algebraic formalism. Also, [29] focuses on testing while in this paper, we focus on formal verification. Similar to [29], in [15], a translation from SystemC to AsmL is given and PSL properties are also embedded in the same formalism as monitors. Then the combination can be both simulated and model-checked.

In [3, 31], a translation from SystemC to the object-based language Rebeca [32] is defined. In order to encode SystemC constructs efficiently, the Rebeca language had to be extended (in terms of an intermediate language) with global variables and wait statement. The generality of process algebraic constructs in mCRL2 allowed us to do without any extension. Furthermore, we could apply existing process-algebraic analysis techniques to SystemC designs as we illustrate in this paper, whereas in [3, 31] a customized model checking engine is proposed.

In [17] the authors have given a translation from SystemC to UPPAAL, a model checker based on timed automata [24]. For this purpose the scheduler is stated as timed automaton. For every event, channel and process instances in the original design a corresponding automaton is constructed in the UPPAAL model. The main advantage of [17] comparing to our work is modeling the quantitative time with the clock capability in UPPAAL. The benefit of our mapping is using the strength of the mCRL2 toolset in state space reduction and visualization. Furthermore the liveness properties in UPPAAL are bounded, i.e., they can only be verified within a certain time bound. We do not have such constraints with mCRL2 and can ensure liveness in infinite executions using μ-calculus model checking.

An improved simulation technique is introduced in [5] to prevent the simulator from exploring the unnecessary interleavings. Prior to actual simulation a model checker is executed on the pairs of concurrent processes. If the model checker fails to prove the consistency in race conditions then an error is reported. After locating the spurious executions by the model checker the simulator is informed on unnecessary traces. The simulator uses partial order reduction to simulate only the necessary paths. The partial order reduction of [5] is equivalent to our tau-confluence reduction in Section 7. However the reduction is used in [5] to improve the simulation of the SystemC designs. In this paper we eliminate the

spurious executions for the purpose to reduce the state space and make the huge designs accessible for further analysis.

This paper is an extended and improved version of [18].

2. Background

2.1. SystemC

In this section, we give an overview of the fundamental structures in SystemC. The basic building block of SystemC programs are modules (SC_MODULE). The actual behavior of a module is performed by the SystemC processes, running concurrently in a module. There are essentially three types of processes in SystemC: methods, threads and cthreads. In the initialization stage, all the defined processes in the program are executed once. The method processes cannot be suspended during their execution. So, after being enabled, a method executes its body from beginning to end. A method is reactivated again whenever an event occurs in the system, to which the method is sensitive. Threads are activated only once, and it is their own duty to suspend themselves on the desired points, by calling `wait()`. The simulator has to implicitly save the internal state of threads, in order to use them when they resume. Cthreads or clocked threads, are special kinds of threads, which can be sensitive to only one edge of one clock in their sensitivity list of signals.

There is a scheduler in the model to manage the concurrent execution of the processes. The logic of the scheduler is determined by the simulator. Processes are non-preemptive, which means that a process, in its turn, runs a piece of code and voluntarily releases the control. There are two different cases by which a process releases the control: when it executes a wait statement, or when it terminates. In Figure 1, a simple SystemC model with two processes, namely, `get_input` and `count`, is given for a D-type Flip-Flop. The first process, i.e., `get_input`, is a method, and the second one, `count`, is a thread which has a wait statement on an event.

Each process can be made sensitive to a set of events. When a process, which is sensitive to a particular event, is suspended, it will be activated again as soon as that event occurs. This mechanism helps synchronize different processes, and also it is useful in creating responsive elements that react to the events in their environment. Sensitivity can be static or dynamic. In static sensitivity the processes are made sensitive to a fixed set of signals at their declaration by the keyword `sensitive`. Whenever a change is spotted in a signal, the corresponding event is generated so that all the sensitive processes to that signal can be activated. Each process can be made sensitive to the positive edge of a signal (`.pos()`), the negative edge (`.neg()`) or both. For example `get_input` is sensitive to the positive edge of `clk`. Whenever the value of `clk` has a transition from `false` to `true`, an event is generated automatically by the system so that if `get_input` is suspended, it will continue its execution. In dynamic sensitivity the process explicitly mentions at the waiting point the event for which it is waiting. In Figure 1, `count` tells the scheduler that it is waiting for the event `e`. So, whenever another process explicitly triggers the event by calling the method `notify` on `e`, `count` will be again switched to a ready process.

The communication mechanism is based on two concepts: ports and channels. A port is a kind of pointer to a channel. For example, `clk` is an input and `dout` is an output port. Usually when the modules are instantiated, channels are used to interconnect the modules together. *Signal channel* is a special kind of channel, which does not change its value immediately when a new value is written to it. When all the ready processes are evaluated, the simulation enters a new step called the update phase. In this phase

```
SC_MODULE(dff) {
  sc_in<bool> din;        // data input port
  sc_in<bool> clk;        // clock input port
  sc_out<bool> dout;      // data output port
  sc_event e;           // communication event
  sc_int<16> num;   // counting the changes in din
  SC_CTOR(dff) {         // constructor
    num = 0;
    SC_METHOD(get_input);  //create a method
    sensitive_pos << clk;    //static sensitivity
    SC_THREAD(count);      //create a thread
  }
  void get_input() {
    dout.write(din.read());  //change output
    e.notify(SC_ZERO_TIME);
  }
  void count() {
    while( true) {
      wait( e);          // wait on e
      num = num + 1; }    // increasing num
    }
};
```

Figure 1. SystemC module for DFF

the old values of the channels are updated, the contentions are resolved and the suspended processes that have to be awaked are notified. The evaluate-update round is usually known as the *delta cycle*.

Simulation Semantics. The formal semantics of simulation in SystemC is described using distributed abstract state machines in [27]. Here we overview the simple phases in the simulation kernel, which take place after the elaboration phase in which modules are instantiated and channels are connected among modules. These are the steps in the simulation of the SystemC designs [28]:

1. *Initialize*: Execute all processes to initialize the system.

2. *Evaluate*: Execute a process that is ready to run. Iterate until all ready processes are executed. Events occurring during the execution could add new processes to the ready list.

3. *Update*: Execute any update calls made during step 2. This step is used by the signal channels as a synchronization point. When a signal is modified during the program execution, its current value does not change. The new value can only be observed in the update phase. The contentions are also resolved.

4. *Notify*: If delayed notifications are pending, determine the list of ready processes and proceed to Evaluate phase (step 2).

Finally, when the simulation is finished a clean-up phase destructs the created structure and releases the allocated memory.

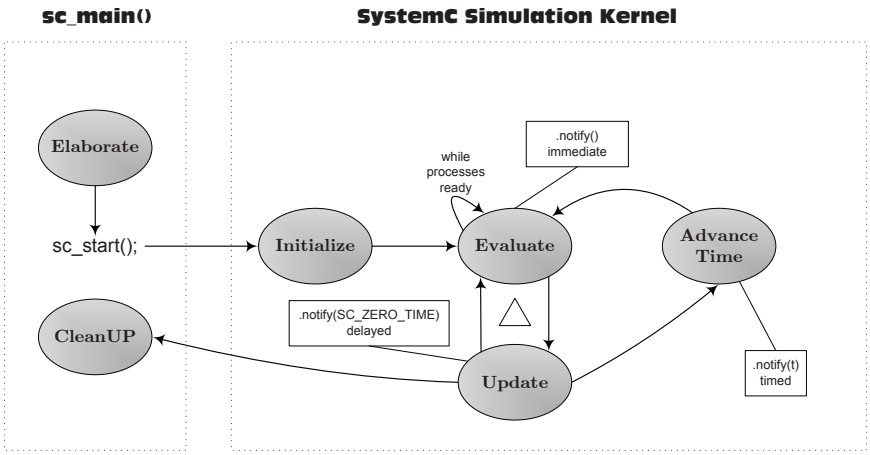

Figure 2. SystemC Simulation kernel ([4])

There is a further step in the original algorithm to advance the time and perform time-notification, but we ignore quantitative time here. Each evaluate-update phase is called a delta cycle in terms of simulation semantics. Notice that the number of delta cycles can be more than one, since in the evaluate phase some events (such as notifying an event or changing a signal) can occur that awake suspended processes. Immediate notifications are the ones that deliver their events as soon as they occur. In contrast, delta delayed notifications deliver their events in the next delta cycle. The syntax for delta cycle notification is `notify(SC_ZERO_TIME)`.

Figure 2 shows the transition diagram of the simulation kernel, which is inspired from [4].

2.2. mCRL2

mCRL2 is a successor to μCRL, and extends it by including features such as true concurrency (in terms of multi-actions), real time, higher-order functions and concrete data types. μCRL, in turn, extends the Algebra of Communicating Processes (ACP) [2] with abstract data types. We refer to [14] for a more elaborate description of mCRL2 features. The choice of mCRL2 as our formal language is motivated by the existence of a rich set of abstract data types as first-class entities in mCRL2. The strong tool-set also provides a powerful mechanism for analyzing specifications. The summarized syntax of mCRL2 processes is given below.

$$p ::= a(d_1, \ldots, d_n) \mid \tau \mid \delta \mid p{+}p \mid p{\cdot}p \mid p \parallel p \mid \tau_I(p) \mid$$
$$\partial_H(p) \mid \nabla_V(p) \mid \Gamma_C(p) \mid \textstyle\sum_{d:D} p \mid c \rightarrow p \diamond p$$

A basic action a of a process may have a number of arguments $d_1, \ldots, d_n$. These arguments correspond to the data elements. Action τ (which does not take any parameter) denotes the internal (unobservable) action. Process δ denotes the deadlocked process in which no further transition is possible. Non-deterministic choice between two processes is denoted by the "$+$" operator. Processes can be composed sequentially and in parallel by means of "$\cdot$" and "$\|$", respectively. The abstraction operator $\tau_I(p)$

renames actions in I into τ and thus makes them unobservable. To enforce synchronization, the encapsulation operator $\partial_H(p)$ specifies the set of actions H which are not allowed to occur. Conversely, the allow operator $\nabla_V(p)$ indicates the only actions that are allowed to occur. To show possible communications in a system and the resulting actions, communication operator $\Gamma_C(p)$ is used. The elements of set C are of the form $a_1 \mid a_2 \mid \cdots \mid a_n \rightarrow c$ (for $n \geq 2$), which intuitively means that action c is the result of the multi-party synchronization of actions a_1, a_2, ..., and a_n. The non-deterministic operator is generalized by the parameterized sum operator $\sum_{d:D} p$, where the variable d may appear (and is bound by $\sum$ in the process p. A conditional statement is represented by $c \rightarrow p_0 \diamond p_1$, where process p_0 will start if the condition c evaluates to true, or otherwise p_1 will take over.

There are a number of built-in data types in mCRL2, such as integers, reals, lists, sets and functions which are quite useful for our implementation.

An mCRL2 specification contains a set of definitions; each of them are prefixed with a keyword determining their purpose. By a **sort** definition one can define a new data type. The constructors of a data type are given by **cons** (for constructor) and **struct** declarations. Constructors defined using **struct** get their equality and inequality axioms generated for free. A data type can equipped with **map**s, which are user-defined mappings for manipulating data. A set of equations (**eqn**) can be defined to specify the definition of a mapping. A new process is declared by **proc**. The initial process is designated with keyword **init**.

The mCRL2 tool-set contains tools for state space generation, reduction, simulation, visualization and model-checking. Furthermore, it can be smoothly integrated with the CADP tool-set [9].

3. Kernel Data Types

There is a generic pattern of specification which is repeated in each translation from SystemC to mCRL2. This pattern concerns the definition of common data types, e.g., for variable names, signal names, process names, bits and vectors, and the kernel processes for the simulation semantics of SystemC. In this section, we illustrate the generic definition of data types. The simulation kernel processes of SystemC are described in Section 4.

The set of data types that are commonly used in all translations from SystemC to mCRL2 are summarized in Figure 3. For the sake of brevity, we only describe the data type mappings and do not present the specification of the equations manipulating the data types here. The interested reader may find the complete implementation in [34]. The first part of the definitions, given in Figure 3 - Part 1, represents the identifiers and names used in the program. These identifiers and names are extracted from the program code by the tool. Six sorts are defined for this purpose, ID: the unique identifers which are assigned to processes (cf. Section 5), SigName: the names of the signals used in the program, PortName: different names which are used as ports, VarName: the names of variables, EvnName: the names of events, and ModuleIns: module instance names.

We assign a natural number to each delta cycle, which helps us differentiate delta cycles; this number is called *round* in our model. So, if we are in the delta cycle i, a process which is ready in the delta cycle $i + 1$ cannot be executed at the moment. Round is defined in Figure 3 - Part 2.

Following the above-explained definitions, there are three groups of data type definitions: variables, signals and process information. In Figure 3 - Part 3, a list of variables (VarList) is defined, which offers two maps for changing and finding the value of a variable. For each variable, other than its name, value,

```
%– Part 1 - Program definitions
sort
  ID = struct · · · ; % process names
  SigName = struct · · · ; % signal names
  PortName = struct · · · ; % port names
  EvnName = struct · · · ; % event names
  VarName = struct · · · ; % variable names
  ModuleIns = struct · · · ; % module names
%– Part 2 - Rounds
sort  Rnd = Nat;
%– Part 3 - Variables
sort
  Scope = struct local(getProc:ID)| field(getModule:ModuleIns);
  Value = struct boolean(getValBool:Bool)?isBool|
   integer(getValInt:Int,getIntSize:Nat))?isInt|
   array(getValArray:Array)?isArray|NullVal?isNull;
  Var  = struct variable(getVarName:VarName, getVal:Value,getScope:Scope);
  VarList = List(Var);
map
  changeVar:VarList × Var → VarList;
  findVarVal:VarList × VarName × Scope → Value;
%– Part 4 - Signals
sort
  Sig = struct sig(getName:SigName,
   getCurVal:Value,getNewVal:Value);
  SigList = List(Sig);
map
  findCurSigVal:SigList × SigName → Value;
  changeSig:SigList × SigName × Value → SigList;
  updateSig:SigList × SigName × Value → SigList;
  findChangedSigs:SigList → SigList;
%– Part 5 - SystemC process information
sort
  Evn = struct event(getEvnName:EvnName,
   getModuleIns:ModuleIns);
  Stat = struct finished?is_finished | ready(getRnd:Rnd)
   ?is_ready | suspended(getEvn:Evn)?is_suspended;
  SensType = struct both | none | negedge | posedge;
  SigSens = struct sigsens(getName:SigName,getType:SensType);
  SigSensList = List(SigSens);
  ProcInf = struct procInf(getID:ID,getStat:Stat,getSensitiveList:SigSensList)|NullInf;
  ProcQueue = List(ProcInf);
map
  changeStat:ProcQueue × ID × Stat → ProcQueue;
  notifyEvn:ProcQueue × Evn × Rnd → ProcQueue;
  findSensType:SigSensList×SigName→SensType;
  notifySigChange:ProcQueue×SigName×Rnd×SensType→ProcQueue;
  getFirstReady:ProcQueue × Rnd → ProcInf;
  isAnyReady:ProcQueue → Bool;
```

Figure 3. Generic data types used in the translated mCRL2 specifications

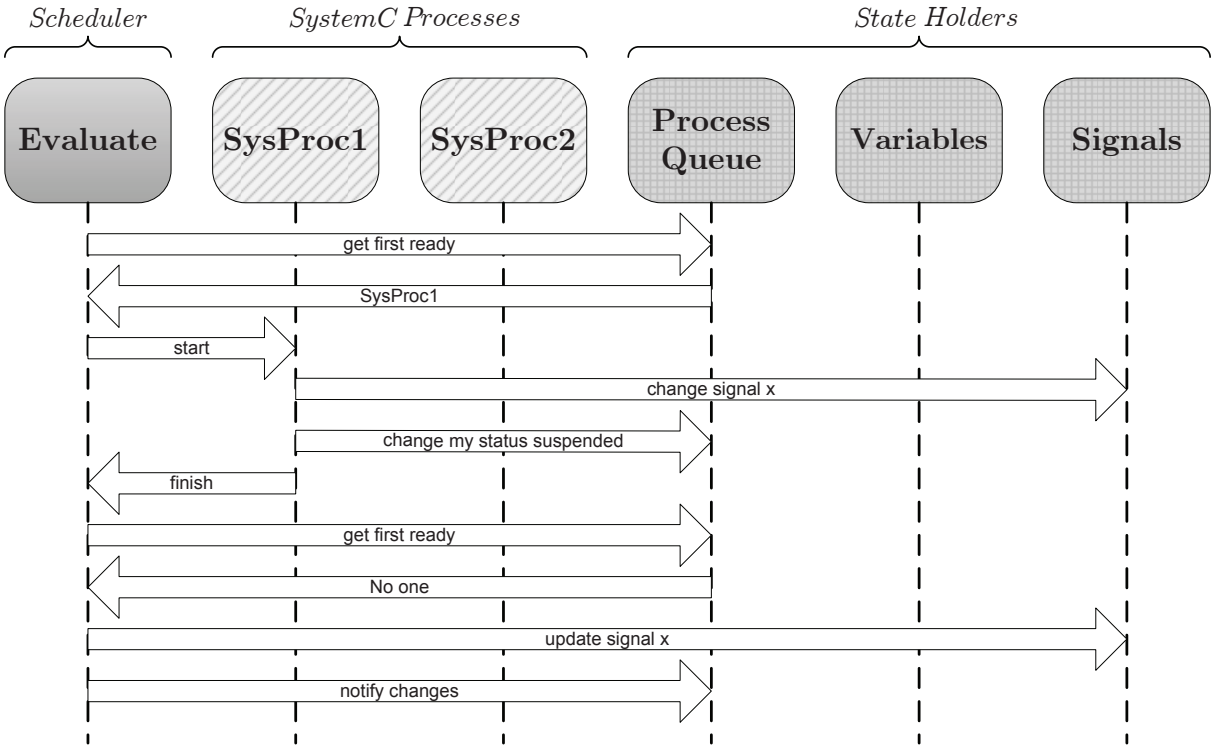

Figure 4. A schematic view of the generic mCRL2 processes for a SystemC specification

and bit width, a scope discriminator is also designated to distinguish the variables in different scopes. A scope of a variable is the place where the variable is defined, and can be local to a process or field of a module.

Generic data type definitions for signals are given in Figure 3 - Part 4. There are two values for each signal, one for the current and one for its new value. The map changeSig can be used to give a new value to a signal. This map is used in translating the regular assignments to signals in the SystemC code. The signal does not get the new value immediately after this call, only by calling updateSig the new value is copied to the current one. This is due to the special treatment of signal modifications in SystemC. The only use of updateSig is in the update phase, where the signals get their new values. The map findChangedSig returns the signals of which the current and the new values are different.

In Figure 3 - Part 5, a list data type (ProcQueue) for process-related information is declared. For each process in this list, three facts are recorded: ID, status and the sensitivity list. ID is a unique identifier, which is used to differentiate between processes. The status can be finished, ready, or suspended. Ready has an argument which shows the current round. For the sensitivity to a signal, it should also be specified if the sensitivity is to a positive edge, a negative edge or to any changes in the signal. There are several maps declared for changing the status of a process, notifying an event, and gathering information from the queue.

4. Kernel Processes

In this section, we present the generic mCRL2 processes which implement the simulation semantics of SystemC. A schematic view of the generic mCRL2 processes resulting from our translation is presented in Figure 4. Initially, we create a scheduler process, one process corresponding to each SystemC process and three processes for saving and updating the latest values of the process queue, variables, and signals. The scheduler process synchronizes with process queue in order to find out, which process is the next ready process to be executed. Note that any scheduling algorithm (including a chaotic non-deterministic choice or any other fixed- or dynamic-order) can be implemented in the process queue process to return the next ready process conforming to the desired scheduling algorithm. We revisit this subject in Section 7. Afterwards, the next ready process is signaled, which may change the value of signals and variables in the course of its execution (by synchronizing with with the variables and signals processes) and eventually suspends itself by synchronizing first with the process queue (in order to change its status) and then with the scheduler (to return the control flow). The process of choosing and scheduling ready processes continues until no more process is enabled. At this stage, the scheduler changes its state to the update phase. During this phase, the scheduler process performs all delta-delayed changes to signals and variables and updates the status of processes. When all these synchronization have taken place a new evaluate round starts. Finally, when the repetition of evaluate and update rounds reaches a fixed-point, a new set of inputs are applied, triggering another round of delta cycles.

A summary of the mCRL2 specification of the processes in Figure 4 is given in Figure 5. The first three processes which are prefixed with "Handler" are containers for the variables, signals and process information lists. The lists are kept in the arguments of the process. Apart from usual actions for retrieving and setting the values from/to the list, the following actions are provided by processes. In the variable handler, the action r_changeVar is used for changing the value of a variable. SigHandler offers two actions for changing and updating the values of a signal: r_changeSig and r_updateSig. There are actions for changing the status of a process (r_changeStat) and notifying an event (r_notifyEvn) or a change in a signal (r_notifySigChange) in the process ProcQueueHandler. Other than the three handlers, the process Evaluate and the translated processes of the model are initiated at the beginning. Each of the threads or methods that are defined in the model is mapped into one mCRL2 process. This mapping is described in more detail in the following section.

An alternative approach to translating variables (signals) would be to include a parameter for each variable (signal) in its handler process. This approach is tried in [19] for a different source language, but it does not scale well to large SystemC specifications (in the order of a thousand lines of code), since the signature and the specification of the handler process becomes too huge and unmanageable.

The process Evaluate is the central process of the system. It has an argument showing the current round. The major work of Evaluate is to choose a ready process from the process information list. The selected process should be ready in the current delta cycle. Then it commands the corresponding mCRL2 process to commence. This is performed by the action s_start. As the scheduling is non-preemptive, Evaluate has to wait until s_finish is issued by the recently started mCRL2 process. This means that the running process has finished its work, and another process can be initiated. Before a process finishes, it should change its state to suspended or finished in its corresponding entry in the process information list. The evaluate phase will be repeated till no more processes are ready in the current delta cycle. At this moment, the process Evaluate will convert to another process, Update. This process recursively checks all the signals to see if there is a signal, of which the current value is different from the new value. If so, it

```
proc VarHandler(vl:VarList)=
  ∑ v:Var r_changeVar(v).
  VarHandler(changeVar(vl,v)) +
  ∑ n:VarName ∑ s:Scope r_getValue(n, findVarVal(vl,n,s),s).VarHandler(vl) +
  r_getVarList(vl).VarHandler(vl);
proc SigHandler(sl:SigList)=
  ∑ n:SigName ∑ v:Value r_changeSig(n,v).SigHandler(changeSig(sl,n,v)) +
  ∑ n:SigName ∑ v:Value r_updateSig(n,v).SigHandler(updateSig(sl,n,v)) +
  r_getSigList(sl).SigHandler(sl);
proc ProcQueueHandler(pq:ProcQueue)=
  ∑ i:ID ∑ s:Stat r_changeStat(i,s).
   ProcQueueHandler(changeStat(pq,i,s))+
  ∑ e:Evn ∑ r:Rnd r_notifyEvn(e,r).
   ProcQueueHandler(notifyEvn(pq,e,r))+
  ∑ n:SigName ∑ r:Rnd ∑ st:SensType
   r_notifySigChange(n,r,st).
   ProcQueueHandler(notifySigChange(pq,n,r,st))+
  r_getQueue(pq).ProcQueueHandler(pq);
proc Evaluate(round:Rnd)=
  ∑ pq:ProcQueue s_getQueue(pq).
  (getFirstReady(pq,round)≠NullInf)→
   (s_start(getID(getFirstReady(pq,round)),round).
   r_finish(getID(getFirstReady(pq,round))).
   Evaluate(round))◇
  (∑ sl:SigList s_getSigList(sl).
  Update(round,findChangedSigs(sl)));
proc Update(round:Rnd,sl:SigList)= (sl≠[])→
  (s_updateSig(getName(head(sl)),getNewVal(head(sl))).
   ((isBool(getNewVal(head(sl))) ∧
   getValBool(getNewVal(head(sl))) == false ∧
   getValBool(getCurVal(head(sl))) == true) →
     s_notifySigChange(getName(head(sl)),round + 1,negedge)◇
     s_notifySigChange(getName(head(sl)),round + 1,posedge))
  .Update(round,tail(sl)))◇
  (∑ pq:ProcQueue s_getQueue(pq).
  (isAnyReady(pq)→Evaluate( round+1)◇Validate));
```

Figure 5. The kernel mCRL2 processes

updates the signal and also notifies the lists, so that if a process is waiting for a signal change, it will be converted to ready for the next delta cycle. The condition in this process is to check the changed signal to see if it is Boolean and also if there is a positive edge occurring in it. If so, it passes a posedge argument with the s_notifySigChange in order to awake the processes which are only sensitive to a positive edge of the signal. Otherwise, the argument is negedge. The s_notifySigChange message is received by the queue handler which then marks all processes waiting on the corresponding signal (signal edge) as ready.

At the end of Update, the queue of processes is checked to see if any ready process is remaining. If so, another round of evaluation (with an increased round number) is begun. When all the processes are finished, the process Validate is run to check the system for another set of inputs. This is explained in more detail in Section 6.

```
proc get_input(pID:ID,container:ModuleIns)=
  ∑ ro:Rnd.r_start(pID,ro).∑ sl:SigList.s_getSigList(sl).
  s_changeSig(signalConnect(dout,container),
  boolean(getValBool(findCurSigVal
    (sl,signalConnect(din,container))))).
  s_notifyEvn( event( e, container), ro).
  s_changeStat( pID, suspended( event( NullEvn,container))).
  s_finish(pID).get_input( pID, container);
proc count(pID:ID,container:ModuleIns)=
  ∑ ro:Rnd.r_start(pID,ro).loop1(pID,container).
  s_changeStat(pID,finished).s_finish(pID);
proc loop1(pID:ID,container:ModuleIns)=
  s_changeStat(pID,suspended(event(e,container))).
  s_finish(pID).
  ∑ ro:Rnd.r_start(pID,ro).∑ vl:VarList.s_getVarList(vl).
  s_changeVar(variable(num,integer(getValInt(
  findVarVal(vl,num,field(container)))+1,16),field(container))).
  loop1(pID,container);
```

Figure 6. Translation of the proc. in Fig. 1

5. From SystemC to mCRL2 Processes

Generic definitions given in Sections 3 and 4 form the bases for our translation from SystemC to mCRL2. To translate specific SystemC code, it only remains to translate its specific processes and statements therein. In this section, we give an overview of this translation scheme.

Each specific SystemC process is mapped into an mCRL2 process. Each such mCRL2 process has two arguments: the first one is the ID of the process which differentiates it from others. The second one is the module instance that contains this process. The reason for differentiating among processes in such a manner is that a module can be instantiated more than once in a program, and a SystemC process can be instantiated more than once in a module. The translation of the body of the process goes in between the start and the finish actions. Figure 6 shows the mCRL2 translation of the SystemC processes in Figure 1.

As it can be seen in the translation of the processes, the translated processes are basically sequences of different actions for communicating with the kernel processes. These include modifications to the signals and variables. The map signalConnect, used in the description of the process get_input, is a mapping of the form PortName $\times$ ModuleIns $\rightarrow$ SigName. It determines the signal to which a port of a module instance is connected. As the process get_input is a method, in its termination it changes its state to wait on a dummy event. This will also occur when a thread executes the command `wait()`. This means that the process has ended but can be reactivated when one of the events in its sensitivity list occurs.

Statements. By interacting with the kernel processes, the values of signals and variables can be obtained and changed. This makes the translation of assignments and similar statements straightforward. In hardware modeling normally bitwise operations are frequently used, which are not supported in mCRL2. For this purpose we implemented the required functions for converting a value to its binary form and also perform bitwise operations. Further details can be consulted from the code in [34].

Concerning the mapping for control flow statements, the conditional statements such as if-then-else and switch-case are mapped to mCRL2 conditionals. Loops are translated into recursive processes guarded by their conditions, if any.

```
proc Validate = ∑f0:Bool.s_changeSig(clk_sig, boolean(f0)).
∑f1:Bool.s_changeSig(din_sig,boolean(f1)).Evaluate(0);
```

Figure 7. Verification proc. for Fig. 1

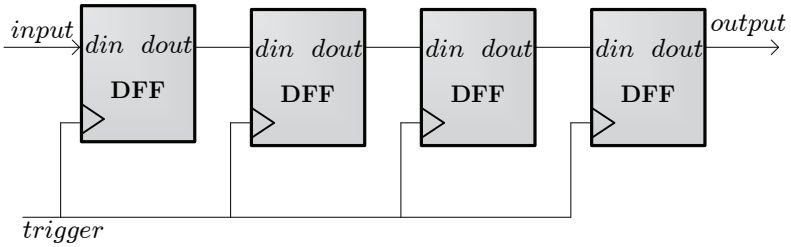

Figure 8. A simple shift register

6. Model checking the code

In order to perform model-checking on a model, all the possible values for the signals form the environment have to be examined. We add a particular process to the system in order to generate all these possibilities. In the remainder of this section, we first describe this process and then illustrate our approach by model-checking a small example.

6.1. Process Validate

To verify a SystemC module in our model, we include the process Validate in the mCRL2 code. This process generates all the possible inputs for the module. For example, Figure 7 shows this process for the SystemC module in Figure 1. The signal din_sig is connected to the port din and clk_sig is connected to clk. If one of the inputs of the module is an integer, we should also specify the range of the variable in generating different values for the input. In addition to this approach, we use an alternative approach in Section 8.2 (inspired by [6]) in order to verify the data semantics of a pipelined architecture.

6.2. Example

We take four modules of the D Flip-Flop (as described in Figure 1, without the `count` method) to build a simple shift register. The design is depicted in Figure 6.2.

There are two input signals from the environment to this circuit: *trigger* and *input*. By altering the values of these signals in the Validate process, various executions are possible. The state space of this system contains 2248 states and 2440 transitions. A visualization of the state space using the visualization tool ltsview (included in the mCRL2 toolset) is depicted in Figure 6.2 . We also verified the correctness of the system by using the the μ-calculus model-checker of mCRL2, which translates the combination of an mCRL2 specification and a modal formula into a Parameterized Boolean Equation System (PBES) (by using the tool lps2pbes) and solves it (by means of the tool pbestool). The correctness

requirement for the shift register is that the value of the input should be communicated after four clock cycles (positive edges of trigger signal) to the output signal. This can be expressed in the modal μ-calculus in terms of the following formula:

$$\begin{aligned}
&\forall b : Bool.[(true^*).\text{changeSig}(din, boolean(b)).\\
&\quad(\neg\text{updateSig}(trigger, boolean(true)) \wedge \neg\text{changeSig}(din, boolean(\neg b)))^*.\\
&\qquad\text{updateSig}(trigger, boolean(true)).\\
&\quad(\neg\text{updateSig}(trigger, boolean(true)))^*.\text{updateSig}(trigger, boolean(true)).\\
&\quad(\neg\text{updateSig}(trigger, boolean(true)))^*.\text{updateSig}(trigger, boolean(true)).\\
&\quad(\neg\text{updateSig}(trigger, boolean(true)))^*.\text{updateSig}(trigger, boolean(true)).\\
&\quad(\neg\text{updateSig}(trigger, boolean(true)) \wedge \neg\text{changeSig}(out, boolean(b)))^*.\\
&\qquad\text{updateSig}(trigger, boolean(true))]false
\end{aligned}$$

The above property states that whenever a change is spotted in the signal din which lasts till the first positive edge of trigger, it should be propagated in the shift register in such a way that after four positive edges of trigger, it reaches the out signal (changeSig$(out, boolean(b))$, note that we have checked deadlock-freedom before hand and hence the above formula is only true if the expected change action is observed before the fifth trigger). The positive edges in trigger are identified here by the action updateSig$(trigger, boolean(true))$), which is issued only when the value of trigger is modified from false to true.

6.3. Tool

We have implemented a tool for automatically translating the SystemC code into mCRL2. It supports a reasonable subset of SystemC including the features described above. This includes most of the constructs which are necessary in an RTL design. In the C++ related features, we support different kinds of loops, conditionals and assignments. From the SystemC constructs we support modules, processes, ports and the primitive channels. The tool is written using the Java programming language and the ANTLR compiler generator.

7. Non-determinism in delta-cycles

Non-determinism is not allowed for synthesizable hardware. Hence, many existing SystemC simulators pre-assume that the specification does not include non-determinism, i.e., race conditions, and thus schedule the ready processes in a deterministic manner, usually according to their order of definition in the original code. For example in the initial phase and at each clock cycle of the system in Section 6.2, there are different orders in which the D Flip-Flops propagate their inputs to the output. However, a typical SystemC simulator only considers one path of execution.

Considering only one path of execution sweeps possible race conditions under the carpet. The modeler cannot see whether/when the system has a form of non-determinism and thus different orders of scheduling result in observably different states. An initial attempt to detect such pathological situations is to replace the Evaluate process of Figure 5 with that of Figure 13. The latter Evaluate process chooses the next ready process non-deterministically and thus generates all possible orders of scheduling. Applying this approach to the example of Section 6.2, results in a state-space with 6670 states and 7423 transitions.

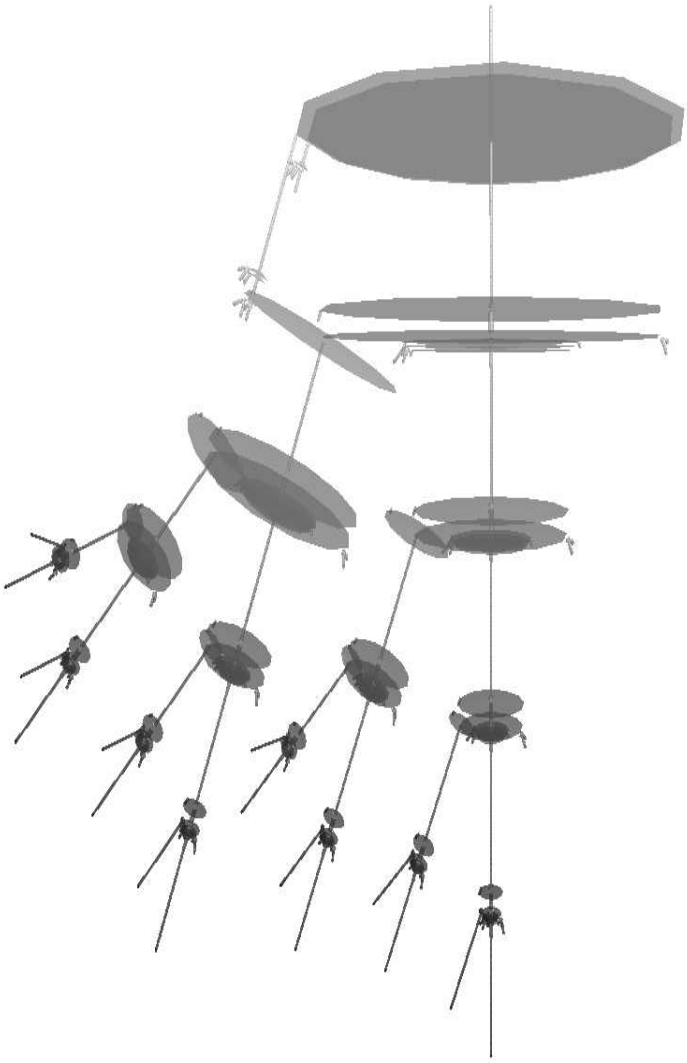

Figure 9. A visualization for the state space of 6.2

The state space of this problem does not show the (very much expected) drastic increase in its size after introducing non-determinism in delta cycles. There are mainly three factors in this system responsible for this incident. First of all, notice that the system has a sequential behavior: only after a process writes on its output signal, the next process is notified to get the data. Thus, after the initialization phase in which all the processes are ready, the value written on the input signal advances step by step in the processes. Secondly, the only shared signal between the processes, which can influence the readiness of all processes, is the trigger signal. This signal cannot awake the processes more than four times, since after four positive edges of trigger the input reaches the output. Finally, this circuit has an important characteristic, called *confluence*. This means that different orderings in delta cycles always end in a single state. Therefore different branching in a delta cycle cannot affect the branching in the next delta cycle.

In most practical cases, however, different interleavings of actions lead to immensely huge state spaces. This is an incarnation of the well-known state-space explosion problem. For instance, with 15 enabled processes the number of all schedules goes beyond one thousand billion. Process algebraic reduction techniques, implemented in the mCRL2 tool-set, can mitigate this problem. By first abstracting from unobservable action, i.e., actions that do not change variables, one can check whether different schedules are confluent (using the tool lpsconfcheck) before attempting to generate the state space. After finding the confluent paths, the state-space generation can be done by only generating one instance among confluent paths. After generating the state-space, the state-space can be reduced, e.g., modulo weak trace equivalence. The result is usually a small state-space, which can be examined using the visualization or the model-checking tools.

To better illustrating this technique, we analyze a rather straightforward case study. (Note that we use a smaller case study to show the actual state-space; the combination of reduction and model-checking technique explained in the remainder of this section, however, scales up to much larger case studies.) Consider a two-bit shift register, similar to the four-bit counterpart of Section 6.2. This shift register consists of merely two D Flip-Flop modules, as depicted in Figure 10. In order to distinguish between different delta cycles better in the final state space, we use two specific actions in the model to show the start and the end of a delta cycle, called $str(x)$ and $fin(x)$, for the start and the end of delta-cycle x, respectively. The beginning of a delta cycle can be marked in the process Evaluate, after the condition that tests if there are any processes ready in the process queue. The end of a delta cycle can be best identified when the process Validate is initiated, since at this position we can ensure that all the possible execution paths have been finished.

Since the changes in the values of the signals are of interest for us, we hide all the actions except for the ones that assign a new value to a signal. Also we keep the actions that are added to the system for designating the delta cycles. Under these conditions, the state space of the problem after branching bisimulation reduction is shown in Figure 11. The states in which non-deterministically a ready process is chosen are represented with a darker color. These states form a confluent (i.e., a diamond) structure in the state space. Thus, the order of executions do not matter and we can minimize the diamonds using tau-confluence reduction. This has been done in Figure 12.

In the case of a two-bit shift register, the absence of race condition can be checked manually by examining the reduced state space. However, for larger designs, the above technique must be combined with model-checking to detect race-conditions. The model-checking problem is about checking a temporal formula to the effect that no two conflicting assignments to a signal or variable are performed (in two paths) during a delta cycle. The following formula shows the μ-calculus property which has been

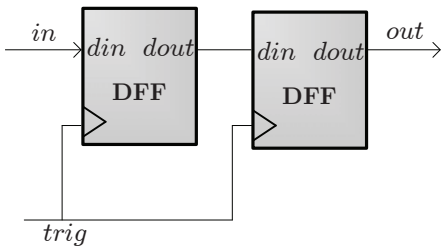

Figure 10. A two bit shift register

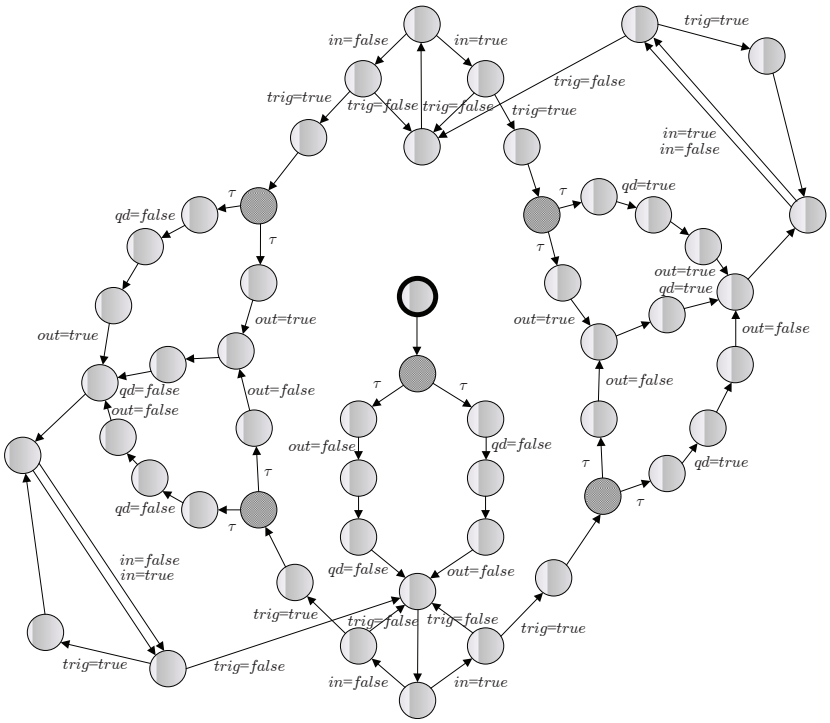

Figure 11. The state space of Figure 10 after branching bisimulation reduction

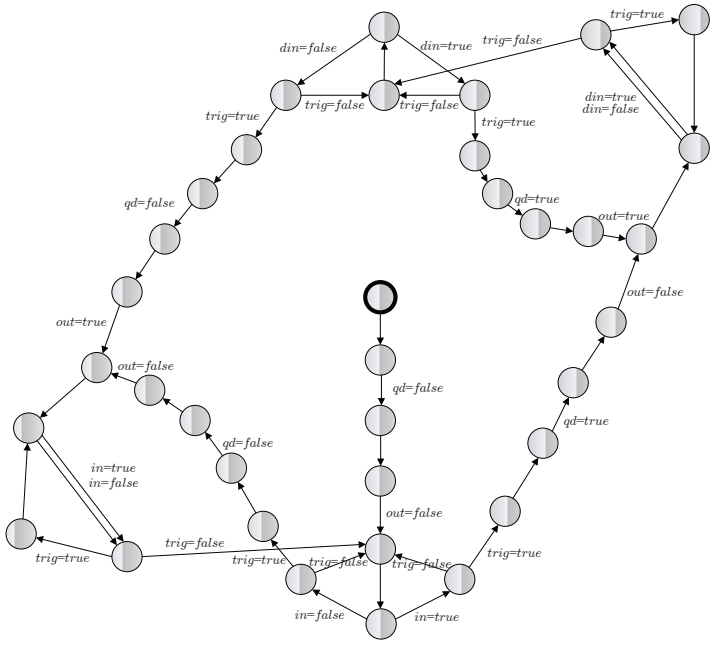

Figure 12. The state space of Figure 10 after tau-confluence reduction

used in our shifter example.

$$
\begin{aligned}
&[true * .str(x)](\\
&\quad [\\
&\qquad (\neg str(x) \wedge \neg fin(x)) * .\ \text{changeSig}(out, boolean(false)).\\
&\qquad (\neg \text{changeSig}(out, boolean(false)) \qquad \wedge\\
&\neg \text{changeSig}(out, boolean(true)) \wedge \neg \text{fin}(x)) * .\\
&\qquad fin(x)\\
&\quad] false\\
&\quad \vee\\
&\quad [\\
&\qquad (\neg str(x) \wedge \neg fin(x)) * .\ \text{changeSig}(out, boolean(true)).\\
&\qquad (\neg \text{changeSig}(out, boolean(false)) \qquad \wedge\\
&\neg \text{changeSig}(out, boolean(true)) \wedge \neg \text{fin}(x)) * .\\
&\qquad fin(x)\\
&\quad] false
\end{aligned}
$$

The above formula states that at any point, if the start of delta cycle x is observed then both of the actions changeSig$(out, boolean(true))$ and changeSig$(out, boolean(false))$ cannot be seen within the delta cycle.

```
proc Evaluate(round:Rnd) = ∑ pq:ProcQueue.
  s_getQueue(pq). (getFirstReady(pq,round)≠NullInf) →
    (∑ x:ProcInf.(x in getReadyList(pq,round) )→
    s_start(getID(x),round).r_finish(getID(x)).Evaluate(round))
 ◇ (∑ sl:SigList.s_getSigList(sl).Update( round, findChangedSigs(sl)));
```

Figure 13. Evaluate for the non-deterministic scheduling

8. Case Study

We considered both single-cycle and pipelined specification of mMIPS (mini MIPS) which is the generic basis for MIPS processors. With some extensions for incorporating network facilities, mMIPS is used in MiniNoC, a Network-On-Chip based multi-processor SoC. We analyzed both simplified single-cycle version of the mMIPS processor (comprising about 1000 lines of SystemC code) as well as its pipelined version (comprising about 2500 lines of SystemC code), which are explained in the coming subsections.

8.1. Single-Cycle mMIPS

The overall design of the single-cycle MIPS processor is depicted in Figure 14. It is a simplified SystemC implementation of a classic single-cycle MIPS processor as described in [16].

To deal with its complexity, we partitioned the system into subsystems, and for each part we separately translated the described components into mCRL2 using our tool and analyzed it for different properties. The subsystems are the data path for controlling the program counter, arithmetic (R-type) instructions, and memory unit. The rationale behind this separation is avoiding state space explosion. In Figure 14, the program counter datapath and memory unit components are shown. To verify arithmetic instructions part, we had to exclude the data memory sub-component (denoted by “dmem” in Figure 14) from our selected components.

We verified the system with a computational program. In order to verify the system, we used processes for monitoring the actions of the system for each temporal property. These processes synchronize with the actions in the system in the order specified by the temporal property. If these synchronizations cannot take place, then a deadlock is reported in the state space generation. In the first subsystem, the monitor synchronizes itself with the changes in the program counter (pc) signal and after each non-jump operation, checks them to be incrementing. In the second subsystem, it checks the inputs and also the output of the ALU. For the memory part, the changes of the memory signals are watched; in the PC subsystem, we were able to find a minor bug (due to an inappropriate type definition in the SystemC code) leading to deadlock. The mistake was happening in the model because of declaring a signed variable for the program counter. When half of the expected interval is enumerated by the counter, because it is signed, its value will wrap around. This will make the simulation erroneously.

The following table shows the size of the state-space generated for each subsystem.

Subsystem	Memory	ALU	PC Datapath
States	130601	230757	6838
Trans.	138726	256009	7836

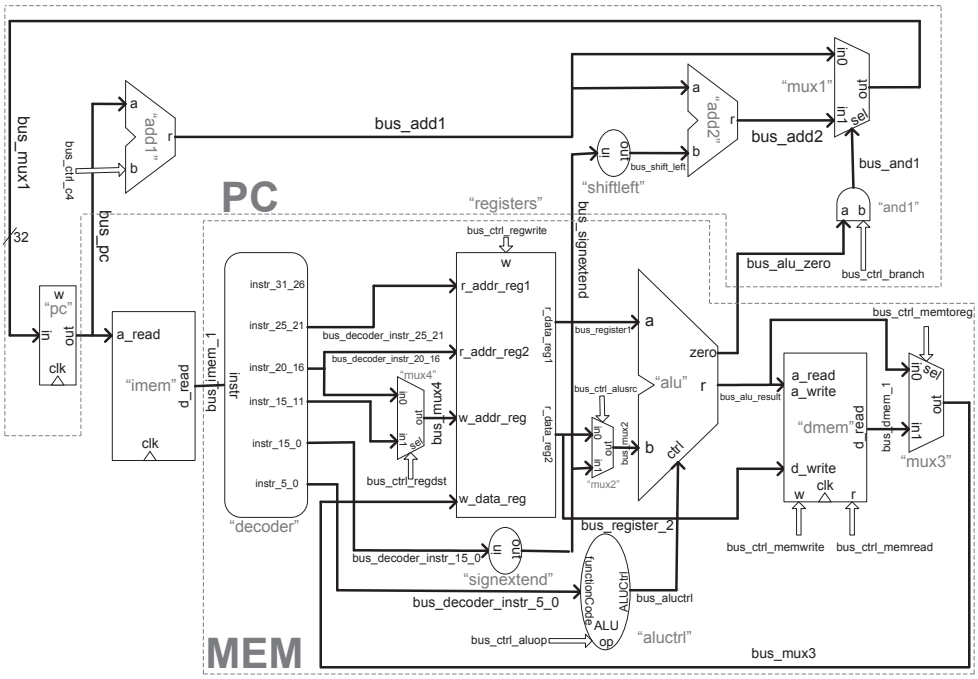

Figure 14. A Schematic View of the Single-Cycle mMIPS Design

8.2. Pipelined mMIPS

The pipelined mMIPS processor substantially extends the instruction sets of the single-cycle processor and implements the classic pipelined architecture of [16]. It supports about thirty instructions from the MIPS instruction set (about 5 times as many as the single-cycle design), and is used in the design of a Network-on-Chip (NoC) multi-processor system (MiniNoC). In this section, we first describe the generic verification technique and afterwards, we explain the application of the technique to our case study.

In their seminal paper [6], Burch and Dill put forward an efficient approach to the verification of the pipelined execution of instructions. This method has been widely considered as a fundamental way to formally prove the correctness of processors, and has been used in many case studies such as [36, 7, 33]. The verification scheme is depicted in the commuting diagram of Figure 15.

The state of the processor is defined at two different abstraction levels: ISA (Instruction Set Architecture) and MA (Micro Architecture). In ISA, only those parts of the system are considered which are visible to the user: register file, data memory and program counter. In MA, the actual state of the system is considered, containing all the intermediate storages which are also important to the actual implementation of the pipeline processor. The functions F_{Impl} and F_{Spec} are the transition functions for implementation and specification. A transition function gets the current state as well as the inputs and gives the next state.

The key idea of Burch and Dill's approach is to use the flushed state of the MA level, in the sense that only those states are investigated in which the processor is stalled for a sufficient number of cycles. In the flushed states, no partially executed instruction remains in the pipeline. The processor is said to be

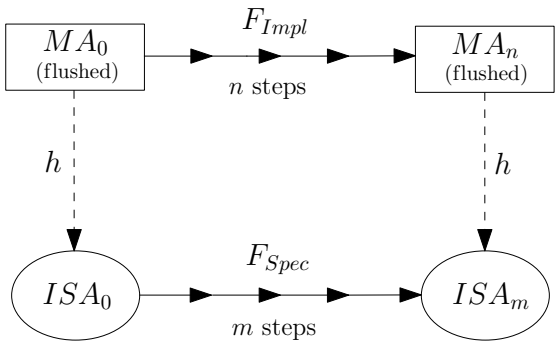

Figure 15. Burch and Dill's Commuting Diagram

correct if any arbitrary sequence of execution at the MA level has the same effect as its ISA counterpart on the observable components, as long as the comparison is done in the flushed states. The function h is an abstraction function that maps an MA state to its corresponding ISA state by stripping off all but the programmer visible parts of the implementation state.

We translated the SystemC specification of the pipelined mMIPS to mCRL2. In order to observe the result of a single instruction, we put it at the beginning of the pipeline, i.e., the output signal from the instructions ROM. As we seek the finite state space representing this instruction, we prevent the program counter adder to increase unboundedly. With each edge of the clock, the instruction advances one stage in the pipe, and since no new instruction is allowed to come into the system, after quite a number of steps the state space generation is stopped. On average, after about 5000 states (in the mCRL2 specification) an instruction traverses all through the pipeline. Notice that the Burch and Dill approach cannot be directly applied to the above state space, since in our case, we have to deal with labeled transition systems in which all the information is reflected on the labels. In the original approach, two paths of specification and implementation were checked to see that the user-visible components of the state are modified similarly or not. Likewise, here we have to check whether the actions that are in charge of changing the contents of the user-visible components are matching. For this purpose, we use the mCRL2 abstraction operator to hide all the uninteresting actions which do not have any effects on these components. By applying tau-confluence reduction to eliminate all the confluent internal actions (i.e., the abstraction function in the Burch and Dill approach), the result can be compared with the specification to see if the actions are issued correctly. As an example, consider the R-type instructions, that is, arithmetic and logic operations with all operands in registers. For this kind of instructions, what is important is the result from the ALU, which is written back to the memory in the write-back stage of the pipeline. Similarly for the store instructions, the main actions are those that write the actual value into the memory in the memory stage of the pipeline. We considered different types of instructions of the system (about 15 different types), and investigated the exposed behavior to be the same as the desired behavior of the processor.

During our analysis, we discovered a flaw in the semantics of the arithmetic shift instruction. In the different flavors of the shift right instructions in the previous implementation, the sign bit of the register was not shifted inside. Instead the processor used to introduce zeros in the register. Although this is reasonable for unsigned number, but when the negative numbers are taken into account, the situation

Case Study	States	Transitions
DFF (Deterministic)	2248	2440
DFF (Nondeterministic)	6670	7423
Memory	130601	138726
ALU	230757	256009
PC Datapath	6838	7836

Table 1. The size of the generated state spaces

differs. For negative numbers the shifted bit should be one. This error was reported to the designers and corrected in the new version of the code.

8.3. Discussion

Verifying the above-mentioned case studies has helped us improve our translation and extend its scope to a larger subset of SystemC. Furthermore, it has revealed certain structures in the state space of SystemC designs, which can be exploited further to reduce their state-spaces while preserving the modal properties of interest. Table 1 gives an overview of the size of the generated state spaces. The following observations can be made:

Firstly, all the state spaces seem to be a sparse graph and almost linear, since a dense graph or a frequently appearing branching would result in a considerable difference between the number of states and transitions. Secondly, the difference between the deterministic and nondeterministic version of scheduling is considerable but not huge (a factor of 3 in the case of the DFF). Both facts can be explained by the nature of hardware, which is supposed to be deterministic. Thus, it seems feasible to detect the (seemingly) few sources of non-determinacy in the system design and prove (e.g., by the static analysis of code) that they deliver the same outcome. Then, a simple linear search of part of the state space would provide an exhaustive proof of correctness. Elsewhere, in [30], we have investigated this path by detecting the source of non-determinism in a subset of the Verilog language.

From our experience with the case studies, it seems unavoidable to use compositional techniques for analyzing larger designs. An obstacle in the practical application of this technique is to re-define the interfaces and the properties on the decomposed components. Mechanizing the decomposition of both the system and the properties is a challenging task, which we would like to take on next in our research.

9. Conclusions

In this paper, we presented a formalization of SystemC code in the process algebra mCRL2. The formalization is implemented in a tool and this enables automatic translation and formal verification of SystemC code in the mCRL2 tool-set. The mCRL2 code can be analyzed in different aspects. A useful application is to prove output determinism, which is ensuring that different orderings in a delta cycle does not affects the final outcome. We showed two systematic ways for this purpose: model-checking and confluence checking.

Using the implemented tool, we analyzed several case studies including a single-cycle and a pipelined MIPS processor. We plan to extend our work by providing support for more advanced features of SystemC, such as quantitative timing and transaction level modeling (TLM). In particular, we believe that our approach is very suitable for the verification of TLM designs due to its algebraic natures. Namely, one can verify the computation components separately and once locally proven correct, they can be replaced by abstract specification processes in order to verify the overall behavior of the TLM design. Furthermore, we think that tailor-made reduction techniques are yet to be developed for the process graphs resulting from hardware designs, in general, and SystemC designs, in particular.

As for the particular case of the mMIPS pipelined design, we did not investigate matters related to control hazards and instruction dependencies. This mainly results from the fact that we only allowed one instruction to go through the pipe that then closed it for the other instructions entrance. There is a possibility that if particular sequences of instructions are put in the pipe some undetected problems occur because of the inconsistencies between them. In order to tackle these issues, we are working to find an upper bound on the length of the sequence of instructions which has to be checked to guarantee that in general, no such problem may take place in the system.

Mechanizing the (de)compositional reasoning about large system-level designs is another research line, which we deem essential for the future application of formal verification techniques at larger scales.

Acknowledgments. Our mMIPS case studies are due to Sander Stuik from the Electronic Systems group at the Electrical Engineering Department of TU/Eindhoven. Useful comments from and stimulating discussions with Sander Stuik and Jan Friso Groote are gratefully acknowledged. We are also appreciative to the members of the formal methods laboratory of the University of Tehran for their useful discussions on the semantics of delta cycles.

References

[1] 1666-2005, IEEE SystemC LRM, available at `www.systemc.org`, 2005.

[2] Baeten J. C. M., Basten T., and Reniers M. A.: "Process Algebra: Equational Theories of Communicating Processes", Cambridge, 2010.

[3] Behjati R., Sabouri H., Razavi N., and Sirjani M.: "An effective approach for model checking SystemC designs.", Proc. of ACSD'08, pp. 56-61, IEEE CS, 2008.

[4] Black D.C., Donovan J.: "SystemC: From the Ground Up", Kluwer Academics, 2005.

[5] Blanc N., Kroening D.: "Speeding Up Simulation of SystemC Using Model Checking", Proc. of SBMF'09, vol. 5902 of LNCS, pp. 1-16, Springer, 2009.

[6] Burch J.R. and Dill D.L.: "Automatic verification of pipelined microprocessor control.", Proc. of CAV'94, vol. 818 of LNCS, pp. 68-80, Springer, 1994.

[7] Brock B., Kaufmann M. and Moore J.S.: "ACL2 theorems about commercial microprocessors." Proc. of FM-CAD'96, pp. 275-293, IEEE CS, 1996.

[8] Campbell C., Grieskamp W., Nachmanson L., Schulte W., Tillmann N., and Veanes M.: "Model-based testing of object-oriented reactive systems with Spec Explorer." *Technical Report MSR-TR-2005-59*, Microsoft Research, 2005.

[9] Fernandez J.-C., Garavel H., Kerbrat A., Mounier L., Mateescu R., and Sighireanu M.: "CADP - a protocol validation and verification toolbox." Proc. of CAV'96, pp. 437-440, 1996.

[10] Gurevich Y., Rossman B., and Schulte W.: "Semantic essence of AsmL." Theoretical Computer Science, 343(3):370–412, 2005.

[11] Ghenassia F.: "Transaction-level modeling with Systemc: TLM concepts and applications for embedded systems." Springer, 2006.

[12] Große D. and Drechsler R.: "Formal verification of LTL formulas for SystemC designs." Proc. of ISCAS'03, pp. V-245–248, IEEE, 2003.

[13] Groote J.F. and Ponse A.: "The syntax and semantics of μCRL, Proc. of ACP'94, pp. 26–62, 1995.

[14] Groote J.F., Mathijssen A., Reniers M., Usenko Y., and van Weerdenburg M.: "The formal specification language mCRL2." Proc. of Dagstuhl'07, 2007. `www.mcrl2.org`.

[15] Habibi A., and Tahar S.: "An approach for the verfication of SystemC designs using AsmL." Proc. of ATVA'05, vol. 3707 of LNCS, pp. 69-83, Springer, 2005.

[16] Hennessy J.L. and Patterson D.A.: "Computer architecture: A quantitative approach." 4th Ed., Morgan Kaufmann, 2006.

[17] Herber P., Fellmuth J. and Glesner S. : "Model checking SystemC designs using timed automata", Proc. of CODES/ISSS'08, pp. 131-136, ACM Press, 2008.

[18] Hojjat H., Sirjani M. and Mousavi, M.R.: Process algebraic verification of SystemC codes. Proc. of ACSD'08, pp. 62-67, IEEE CS, 2008.

[19] Hojjat H., Sirjani M., Mousavi, M.R. and Groote J.F.: Sarir: A Rebeca to mCRL2 Translator (Tool Paper). Proc. of ACSD'07, pp. 216-222, IEEE CS, s2007.

[20] Holzmann G.J.: "The Spin Model Checker - Primer and Reference Manual." Addison-Wesley, 2003.

[21] Karlsson D., Eles O., and Peng Z.: "Formal verification of SystemC designs using a Petri-Net based representation." Proc. of DATE'06, pp. 1228-1233, ACM Press, 2006.

[22] Keutzer K., Malik S., Newton R., Rabaey J. and Sangiovanni-Vincentelli A.: "System level design: orthogonalization of concerns and platform-based design." IEEE TCAD, 19(12):1523-1543, 2000.

[23] Kroening D., and Sharygina N.: "Formal verification of SystemC by automatic hardware/software partitioning." Proc. of MEMOCODE'05, pp. 101-110, IEEE CS, 2003.

[24] Larsen K.G., Pettersson P. and Yi W.: "Uppaal in a nutshell", Journal on Software Tools for Technology Transfer (STTT), 1(1-2): 134-152, 1997.

[25] Man K.L: "SystemC^{FL} : a formalism for hardware/software co-design." Proc. of ECCTD'05, pp. 193-196, IEEE, 2005.

[26] Moy M., Maraninchi F., and Maillet-Contoz L.: "LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level." Proc. of ACSD'05, pp. 26-35, IEEE CS, 2005.

[27] Müller W., Ruf J., and Rosenstiel W.: "An ASM based systemC simulation semantics." Chapter 4 of SystemC: methodologies and applications, pp. 97-126, Kluwer, 2003.

[28] Panda P.R.: "SystemC: a modeling platform supporting multiple design abstractions." Proc. of ISSS'01, pp. 75-80, ACM Press, 2001.

[29] Patel H.D., and Shukla S.K.: "Model-driven validation of SystemC designs." Proc. of DAC'07, pp. 29-34, IEEE, 2007.

[30] Raffelsieper M., Mousavi M.R., Roorda J.-W., Strolenberg C. and Zantema H.: "Formal Analysis of Non-Determinism in Verilog Cell Library Simulation Models." Proc. of FMICS'09, pp. 133-148, 2009

[31] Razavi N., Behjati R., Sabouri H. , Khamespanah E., Shali A., and Sirjani M.: "Sysfier: Actor-based Formal Verification of SystemC", ACM Transactions on Embedded Computing Systems, 2010. To appear.

[32] Sirjani M., Movaghar A., Shali A., and de Boer F.S.: "Modeling and verification of reactive systems using Rebeca." Fundamenta Informaticae, 63(4):385–410, 2004.

[33] Srinivasan S.K., and Velev M.N.: "Formal verification of an Intel XScale processor model with scoreboarding, specialized execution pipelines, and impress data-memory exceptions." Proc. of MEMOCODE'03, pp. 65-74, IEEE CS, 2003.

[34] SystemC to mCRL2 toolkit, `http://www.win.tue.nl/~mousavi/sysc08/`.

[35] Traulsen C., Cornet J., Moy M. and Maraninchi F.: "A SystemC/TLM semantics in Promela and its possible applications." Proc. of SPIN'07, vol. 4595 of LNCS, pp. 204-222, Springer, 2007.

[36] Velev M.N. and Bryant R.E.: "Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors." J. Symb. Comput. 35(2):73-106, 2003.
[37] Vardi M.Y.: "Formal techniques for SystemC verification.", Proc. of DAC'07, pp. 188-192, IEEE CS, 2007.