Modelling SystemC scheduler by refinement

Dominique Cansell, Dominique Méry, Cyril Proch

MODELLING SYSTEMC SCHEDULER BY REFINEMENT

Dominique Cansell
LORIA, Université de Metz

Dominique Méry and Cyril Proch
LORIA,Université Henri Poincaré Nancy 1

ABSTRACT

Systems on Chip, or shortly SoCs, and SoC architectures denote a challenging set of problems of specification, modelling techniques, security issues and structuring questions. Our methodology, for designing models of (SoC) system from requirements, leads to formally justify hints on the future architectural choices of that system; it is based on the B event-based method, which integrates the incremental development of models using a theorem prover to validate each step of development called refinement. The target system is generally expressed using a programming language notation like SystemC; the SystemC language is used by electronic designers to describe different parts of the system (hardware and software); SystemC constitutes a general framework for simulating and validating the design of the system under construction. The semantics of SystemC is based on its scheduling algorithm described in the language reference manual and we develop a B model of the scheduling. The B *scheduling* model left unspecified parameters depending on the simulated SystemC program and those parameters are instantiated from the operational semantics of the developed SystemC program. By instantiation, we obtain a B abstract model of the simulated program and we can study properties of the SystemC program by simulation. B models are completely validated by the proof assistant of the event-B method. Finally, our models provide a sound framework for understanding the scheduling process.

Keywords. Event B method, refinement, scheduler, operational semantics, systemC

INTRODUCTION

Modelling the SystemC Scheduler

The refinement of events-based models provides a general framework for developing systems from requirements and for expressing the semantical relationship between views of a system; the main idea is to begin the development by a very abstract view or model and to state the fundamental properties required by the system. The goal of the refinement-based development is to produce a formal validated model of the system in an incremental way. Benefits of refinement are numerous and first we underline the control of proof complexity by diffusion through the refined models. Second, the refinement process should start from a very abstract view of the system that leads to the possibility to tackle non trivial systems. The main objective is to write a B event-based model [3] of the SystemC [19] scheduler; the modelling is the part of a general refinement-based methodology for developing systems on chip from requirements to SystemC-like systems. First, we develop a B event-based model of the scheduler defined in the reference manual of SystemC and we let informations on the simulated program in parameters; the refinement makes possible the production of a precise model for the scheduler. Second, since the scheduler's model has parameters left unspecified, we can instantiate the scheduler for a specific SystemC program. Subsequently, the resulting B event-based model is a formal model of the global system made up of the scheduler and the particular program; the resulting model can be used in further developments and can be compared to another instantiated model. It means that the generic model provides a framework for defining the operational semantics for the simulation process, as defined in the reference manual. Since the scheduler is modelled as a generic model, it is defined and developed only once and the user should only define the parameters

specific to the give SystemC program. Moreover, the resulting model is completely validated by the proof process. Objectives of the paper can be summarized as follows:

- To provide an (formal) operational semantics for the SystemC scheduler and hence for the simulation of each SystemC program.
- To use the refinement for capturing the semantics of the scheduler.
- To validate the correctness of the translation of B models into SystemC modules.

Proof-based incremental modelling

Proof-based development methods [4, 2] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [4, 2]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination.

A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine [8].

At the most abstract level it is obligatory to describe the static properties of a model's data by means of an "invariant" predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events or operations of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of proof obligations related to the refinement.

The goal of a B development is to obtain a *proved model*. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for traceability of requirements.

B models rarely need to make assumptions about the *size* of a system being modelled, e.g. the number of nodes in a network. This is in contrast to model checking approaches [7]. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity. Where B has been exercised on known difficult problems, the result has often been a simpler proof development than has been achieved by users of other more monolithic techniques.

A short introduction to B event-based notations

The B event language is based on substitutions; a substitution states the transformation of state variables from a possible pre-state to a possible post-state. In our B models, we use specific substitutions; the substitution $x := E(x)$ denotes the transformation leading to the updating of the state variable x according to the value of $E(x)$ and the substitution $x :\in A(x)$ denotes the updating of the state variable x according to a value of $A(x)$ (a set depending on the pre-value of x).The Before-After predicate of a substitution $P(x, x')$ defines the relation between values of variables before substitution (x) and values of variables after substitution (x'). For the substitution $x := E(x)$, the predicate $P(x, x')$ is $x' = E(x)$ whereas for the substitution $x :\in A(x)$, the predicate is $x' \in A(x)$. Each event has a guard controlling the substitution and the occurrence of the event. A Before-After predicate of event is defined from Before-After predicate of substitution and guard of event. We denote by $S(x)$ any substitution form and an event is built with respect to three schemata recalled in the figure 1.

Finally, the B model language provides the way to define a B event-based model. A (abstract) model is made up of a part defining mathematical structures related to the problem to solve and a part containing elements on state variables, transitions and (safety and invariance) properties of the model. Proof obligations

Event : E	Guard	Before-After Predicate
begin $S(x)$ **end**	$true$	$P(x, x')$
select $G(x)$ **then** $S(x)$ **end**	$G(x)$	$G(x) \ \wedge \ P(x, x')$
any t **where** $G(t, x)$ **then** $S(x)$ **end**	$\exists t \cdot (G(t, x))$	$\exists t \cdot (G(t, x) \ \wedge \ P(x, x', t))$

Figure 1: Definition of events and before-after predicates of events

Name	Syntax	Definition
Binary Relation	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Domain	$\mathsf{dom}(r)$	$\{a \mid a \in s \wedge \exists b.(b \in t \wedge a \mapsto b \in r)\}$
Codomain	$\mathbf{ran}(r)$	$\mathsf{dom}(r^{-1})$
Co-restriction	$r \rhd t$	$r;\ \mathbf{id}(s)$
Anti-co-restriction	$r \ \rhd\!\!\!> t$	$r \rhd (ran(r) - t)$
Image	$r[w]$	$\mathbf{ran}(w \lhd r)$
Overwrite	$q \lhd\!\!+ r$	$(\mathsf{dom}(r) \lhd q) \cup r$
Partial Function	$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \ \wedge \ (r^{-1}; r) \subseteq \mathbf{id}(t)\}$
Total Function	$s \rightarrow t$	$\{f \mid f \in s \nrightarrow t \ \wedge \ \mathsf{dom}(f) = s\}$

Figure 2: B set notations

are generated from the model to ensure that properties are effectively holding: it is called *internal consistency* of the model. A model is assumed to be closed and it means that every possible change over state variables is defined by transitions; transitions correspond to events observed by the specifier. A model m is defined as follows. A model has a name m; the clause **sets** contains definitions of sets of the problem; the clause **constants** allows one to introduce information related to the mathematical structure of the problem to solve and the clause **properties** contains the effective definitions of constants: it is very important to list carefully properties of constants in a way that can be easily used by the tool. Another point is the fact that sets and constants can be considered like parameters and extensions of the B method exploit this aspect to introduce parameterization techniques in the development process of B models. The second part of the model defines dynamic aspects of state variables and properties over variables using the invariant called generally inductive invariant and using assertions called generally safety properties. The invariant $I(x)$ types the variable x, which is assumed to be initialized with respect to the initial conditions and which is preserved by events (or transitions) of the list of events. Conditions of verification called proof obligations are generated from the text of the model using the first part for defining the mathematical theory and the second part is used to generate proof obligations for the preservation of the invariant and proof obligations stating the correctness of safety properties with respect to the invariant.

The B event-based method includes the B data modelling language, the B events language and the B models language. The figure 2 gives set-theoretical notations of the B data modelling language and it borrows notations and concepts of Bourbaki's group. If f is a function then the substitution $f(x) := E$ is equivalent to $f := f \lhd\!\!+ \{x \mapsto E\}$.

Due to the lack of space, we do not introduce formally the refinement models and they will be effectively used later. A complete introduction of B can be found in [6].

Applications to SoC development

The scheduler model was developed for validating a system on chip produced for measuring the service performance (TS level) in a DVB environment. Formal modelling techniques [12, 1, 5] provide hints on the architecture of the future system and the formal model has been developed using the B event-based method; the resulting B model provides an invariant, which is incrementally built and validated through the

refinement process and the details are extracted from the documentation [9]. However, the resulting B model should be translated into an equivalent code and the question of the adequacy of the resulting code with the B model should be solved by defining a semantical framework for asserting the semantical relationship. It is why we have developed the B model for the SystemC scheduler. Our case study is a monitoring tool for measurement in Digital Video Broadcasting Television (DVB-T) and problems are related to the number of computations and real-time constraints. The implementation of this tool is driven by the hierarchy derived from invariant of models. The refinement allows us to classify parameters into a consistent hierarchy; the hierarchy has properties for deriving a so-called abstract architecture for the system. The hierarchy of the abstract model is not falsified by the hierarchy of the concrete one, thanks to the refinement. Obviously, events of the model can be used to derive algorithmic methods for computing the value of each parameters. Explanations to non specialists of refinement are given through graphs, which capture the relation between parameters. The project includes colleagues of the electrical engineering department and three industrial partners; the project leads to the effective design of a tool correct with respect to the hierarchy among parameters and the B event-based method helps in validating the final choice of implementation. However, it is out of the scope of the current paper which focuses on the model of the scheduler.

Related works

The definition of an operational semantics is not new [15, 17]; the main fact is that we use the B event-based methodology for writing the abstract scheduler; for instance, the ASM language is used to define the simulation semantics of SystemC [15, 10] as the semantics of SpecC [13], an equivalent language of SystemC, or semantics of VHDL [11]. Unfortunately, these works [15, 10] consider the scheduler of SystemC V1.0 which is really different of the actual version (V2.0). The major goals of these works are the definition of precise specifications for future implementation of a scheduler or to investigate SystemC interoperability with Verilog, SpecC and VHDL. Our goal is to provide a formal semantics to the SystemC scheduler and we use the B framework for expressing the semantics. A second difference is that we write incrementally the operational semantics and the incremental process improve the understanding of the scheduler. Finally, the resulting B event-model for the simulation semantics can be used as a parametric framework for analysing a specific SystemC program and this point is not addressed elsewhere in the literature. Others works [18, 16] aim to develop abstract models of SystemC programs and use model checking techniques; those approaches are verification-oriented and we are dealing mainly with design-oriented ones.

Summary

Section 2 describes the SystemC programming language and its concepts; the principles of simulation are sketched by the simulation algorithm. Section 3 reports the incremental development of the SystemC scheduler using the refinement process; the section is the main technical aspect of the paper. A simple example illustrates the technique of model instantiation in the section 4. Section 5 concludes the work.

SYSTEMATIC B MODELS FOR SYSTEMC SIMULATION

Requirements for the SystemC Simulation

SystemC [19, 14] is a set of C++ class definitions with hints for using these classes. The SystemC library of classes and simulation kernel extend C++ to enable the modelling of systems. Extensions include handling for concurrent behavior, time sequenced operations, data types for describing hardware, structure hierarchy and simulation support. The core language consists of an event-driven simulator as the base (scheduler). The scheduler uses events and processes.

SystemC: Quick Overview

A SystemC system consists of a set of modules. A *module* is a container class and provides the ability to describe structure. Module is a hierarchical entity that can have other modules or processes inside it. Modules typically contain processes, ports, internal data channels and possibly instances of other modules. Ports are used to describe structure, while channels are used to represent communication. Processes are concurrent and are used to model the functionality of the module. Processes are contained inside modules and are particular methods of modules. SystemC provides different process abstractions for hardware and software designers. Channels or signals handle communications between processes but communications

between processes inside different modules is supported by ports, interfaces and channels. The port of a module is the object through which the process accesses a channel. Events are the basic synchronization objects for processes. Processes are triggered with respect to sensitivity on events. Concretely, an event is used to represent a condition that may occur during the simulation and to control the triggering of processes. Static sensitivity is defined before simulation starts but dynamic sensitivity is defined after simulation starts and can be altered during simulation.

SystemC: Execution Semantics

The function `sc_main()` is the entry point from the library to the user's code (as the function `main()` in C++ programs). Elaboration is defined as the execution of the `sc_main()` function from the start of `sc_main` to the first invocation of scheduler. During elaboration, the structural elements of the system are created and connected throughout the system hierarchy. The structure of the system is created during elaboration time and does not change during simulation.

Before first invocation of scheduler, *initialization* is the first step of simulation. Each process is executed (you can turn off initialization for particular processes with calls of method `dont_initialize()`) during initialization. The order of execution of processes is unspecified but two simulations run using the same version by the same simulator must yield identical results. The next figure presents an example of SystemC modules with concurrent processes, channels and events.

```
#include ''systemc.h''

SC_MODULE(my_module) {
  sc_in<bool> port1;
  sc_out<bool> port2;
  event e2,e3; // events declaration
  sc_signal<int> count; // intern channel

  void proc1() {
     if (count.read() < 10) {
       count.write(count.read()+1);
       e2.notify(); // immediate notification
     } else {
       e3.notify(5,SC_NS); // timed notification
     }
  }

  void proc2() {
     if (count.read() < 11) {
       count.write(count.read()+2);
     } else {
       e3.notify(4,SC_NS); // timed notification
     }
  }

  void proc3() {count.write(0);}
}

  SC_CTOR(my_module) {
    count.write(0);
    SC_METHOD(proc1); sensitive << count;
    SC_METHOD(proc2); sensitive << e2;
    dont_initialize();
    SC_METHOD(proc3); sensitive << e3;
    dont_initialize();
 }
};
```

The SystemC scheduler controls the timing and order of process execution, handles event notifications and manages updates to channels. It supports δ-cycles. A δ-cycle consists of the execution of *evaluate* and *update* phases. There may be a variable number of δ-cycles for every simulation time. SystemC processes are non-preemptive. It means that for *thread* processes, code delimited by two `wait` statements will execute without any other process interrupt and a *method* process completes its execution without interrupt by another process. The scheduler may be invoked such that it will run indefinitely. Once started the scheduler continues until either there are no more events, or a process explicitly stops it, or an exception condition occurs.

Event Notification

Events can be notified in three ways: immediate, δ-cycle delayed and timed. Immediate notification means that the event is triggered in the current evaluation phase of the current δ-cycle. A δ-cycle delayed notification means that the event will be triggered during the *evaluate* phase of the next δ-cycle, the event is scheduled for the next δ-cycle. Timed notification means that the event will be triggered at the specified time in the future.

Events can have only one pending notification, and retain no "memory" of past notifications. Multiple notifications to the same event, without an intermediate trigger are resolved according to the following rule:

$$timed \prec \delta \prec immediate$$

An earlier notification will always override a scheduled one to occur later, and an immediate notification is always earlier than any δ-cycle delayed or timed notification, rules imply non determinism.

Complete Algorithm of Scheduler

The semantics of the SystemC simulation scheduler is defined by the following eight steps in [14]. A δ-cycle consists of steps 2 through 4.

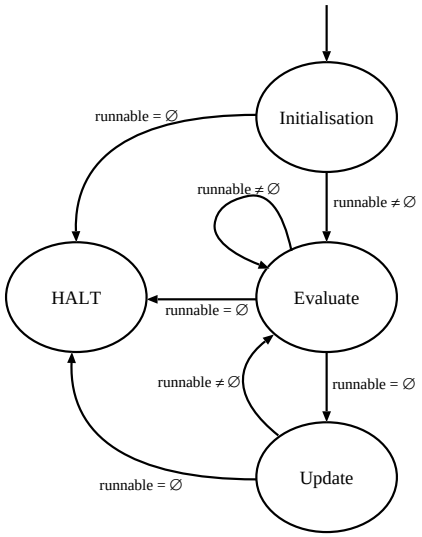
1. *Initialization Phase*:
2. *Evaluate Phase*: From the set of processes that are ready to run, select a process and resume its execution. The order in which processes are selected for execution from the set of processes that are ready to run is unspecified.

   The execution of a process may cause immediate event notifications to occur, possibly resulting in additional processes becoming ready to run in the same *evaluate* phase.
3. Repeat step 2 for any other processes that are ready to run.
4. *Update Phase*: Execute any pending calls to `update()` from calls to the `request_update()` function executed in the *evaluate* phase.
5. If there are pending delta-delay notifications, determine which processes are ready to run and go to step 2.
6. If there are no more timed event notifications, the simulation is finished.
7. Else, advance the current simulation time to the time of the earliest (next) pending timed event notification.
8. Determine which processes become ready to run due to the events that have pending notifications at the current time. Go to the step 2.

We propose to develop the algorithm by refinement from the description of the language reference manual. By this way, we provide an abstract simulation framework which can be instantiated later for a given SystemC program. By instantiation of abstract scheduling model, we define operational semantics of SystemC programs.

INCREMENTAL CONSTRUCTION OF THE SYSTEMC SCHEDULEr

Our B models models the SystemC scheduling and different parts of algorithm are introduced by refinement. Dynamic sensitivity is not considered in our models for simplifications reasons.



model
$Abstract\_Scheduler$
sets
$PROCESSES$;
$STATE = \{init, run, stop\}$
variables
$runnable$
invariant
$runnable \subseteq PROCESSES \wedge$
$phase \in STATE \wedge$
$(phase = stop \Rightarrow runnable = \emptyset)$
initialisation
$runnable :\in \mathbb{P}(PROCESSES) \parallel$
$phase := init$

Figure 3: Automaton and header of abstract model

Abstract Model

The first abstract model describes, in a very abstract way, SystemC scheduler during simulation of program. As shown in previous algorithm, scheduler has two important phases: during the *evaluate* phase, runnable processes are executed and are removed from list of runnable processes. During the *update* phase, a new list of runnable processes is built. In particular cases, processes are adding to the list in *evaluate* phase. The abstract model captures the essence of scheduler and an automaton presented in figure 3 shows the different states of our model. Only processes are considered and there are no clocks, signals and events. The abstraction plays with processes of abstract program. Our abstract model contains three distinct events to animate variables and represent SystemC scheduler reactions. The three events are represented by the three states of figure 3. Because of the abstraction level, the automaton is not deterministic, from particular state (Update for instance), many transitions are possible with the same conditions. As shown figure 3, initialization, δ-cycle and, possibly stop are modelled in the system. Remember that, when refining models, the main idea is to reduce non-determinism but we should start by a very abstract model.

More precisely, abstraction is built very simply: $PROCESSES$ is the set of processes defined in an abstract SystemC program. The abstract model uses a variable $runnable$ which is a sub-set of $PROCESSES$, runnable processes at the current time. Last, a variable $phase$ is introduced. This variable is used to separate different states of the system. Header of model with constants, properties of constants, variables, invariant and initialization of system are presented in figure 3. First, set $STATE$ and value of variable $phase$ model three different states of the system:

- $phase = init$, means than system is in *initialization* phase.
- $phase = run$, means than system is in execution phase i.e. in *evaluate* phase or *update* phase.
- $phase = stop$, means than system is halting and simulation finished.

A first interesting safety property about $runnable$ is $phase = stop \Rightarrow runnable = \emptyset$. It means than simulation is finished only, when there is no more runnable process. This is an important property of simulation presented in the language reference manual. The invariant property is preserved by events of abstract model. The $runnable$ variable is updated during *evaluate* phase, after executions of processes:

- when a process p is executed, it is suppressed from set $runnable$.
- execution of p could add new processes in the same current *evaluate* phase.

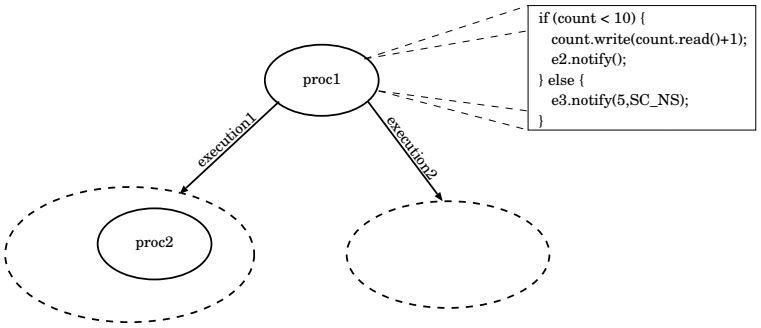


Figure 4: Possible executions of same process

In general case, a same process can have different executions between context of its current execution. For example, figure 4 shows a process with an `if then else` instruction. The process `proc1` is considered as runnable and different executions are produced by different activations of the process. Figure 4 presents the two subsets of processes generated by executions of `proc1`. These two subsets are very simple: only one process for the first and the second is empty.

The two next events model the dynamic of system and scheduling of SystemC design during simulation. Event Evaluate represents a non-deterministic choice of process p in $runnable$ (see guard of event: $runnable \neq \emptyset$) and resulting consequences of its execution. Event Evaluate suppresses processes in $runnable$ and builds a new set of runnable processes. In this abstract level, details of the list construction are not presented but the main information is stated: after each process execution a new list of processes is built.

After one or more activations of event Evaluate, value of variable $runnable$ can be the empty set ($\emptyset$). In the SystemC point of view, it means than all runnable processes have been executed and scheduler must begin its *update* phase. Event Update models the *update* phase of scheduler. Its abstracts level of modelling can not express how the new list is built but our model shows that a new list of runnable processes is built in *update* phase. Details of new list built will be presented in the first refinement.

Evaluate =
select
$phase \neq stop$
then
$runnable :\in \mathbb{P}(PROCESSES) \parallel$
$phase := run$
end

Update =
select
$phase = run \wedge$
$runnable = \emptyset$
then
$runnable :\in \mathbb{P}(PROCESSES)$
end

At last, event HALT models the ending of simulation. In the abstract model, without SystemC event notion, simulation can halt, when variable $runnable$ is empty. The invariant properties are preserved and event is consistent with invariant and requirement of SystemC scheduler. After ending of simulation (modelled by B event HALT), system is deadlocked and no event can be activated.

HALT =
select
$phase \neq stop \wedge$
$runnable = \emptyset$
then
$phase := stop$
end

Finally, our first model sketches the core of SystemC scheduler and simulation principles. Our abstraction presents evolution of processes (runnable thereafter not) during simulation but does not explain scheduling algorithm. Next refinement add details of SystemC simulation principles and the role of scheduler.

First Refinement: SystemC Events

The first refinement introduces SystemC events and notifications of SystemC events. Addition of SystemC events notion implies to split B event Update to specify algorithm of scheduling. Splitting concrete

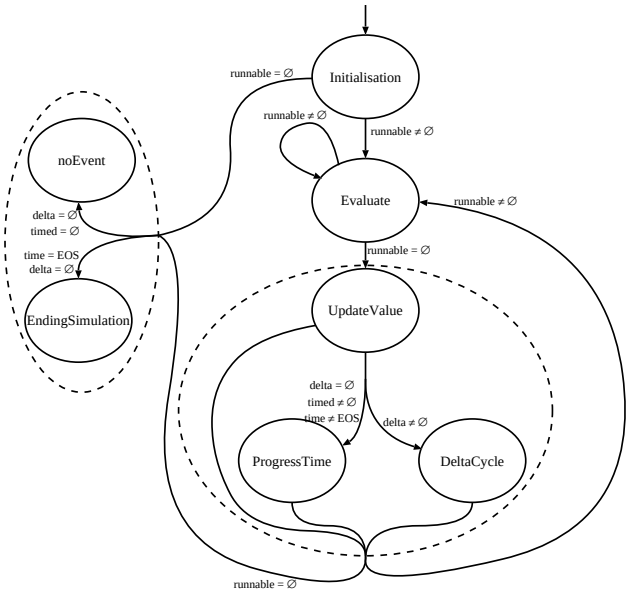


Figure 5: Concrete automaton of the scheduler

events refines abstract event Update. In the same way, abstract B event is refined by two concrete events to model different terminations. The figure 5 shows the new concrete automaton produced from the refined model. The non-deterministic transitions of abstract model (see figure 3) are now deterministic because the new refined model introduces more details. New set and constants are introduced:

- SC_EVENTS is the set of events used during execution of abstract SystemC program.
- $sensitivity$ is a relation from $PROCESSES$ to SC_EVENTS which models the static sensitivity list of each processes defined in program. Because a process can be sensitive on many events, $sensitivity$ is a relation. The relation $sensitivity$ is constant because our models do not consider dynamic sensitivity.
- EOS is a number which represents the ending-of-simulation time. Scheduler can be invoked with a integer parameter which represents the total time of simulation.
- $trigger$ is a relation from $PROCESSES$ to $\mathbb{P}(SC\_EVENTS)$. The relation represents events produce by execution of processes. Because of the code structure, $trigger$ is a relation; a conditional instruction can produce two different executions as presented in figure 4.

SystemC event and sensitivity of processes notions are introduced, we must model different kinds of event notification. To represent notifications, time must be considered in the refined model. Header of refined model is presented below:

refinement
$Event\_Scheduler$
refines
$Abstract\_Scheduler$
sets
SC_EVENTS
constants
$sensitivity, trigger, EOS$
properties
$sensitivity \in PROCESSES \leftrightarrow SC\_EVENTS \wedge$
$trigger \in PROCESSES \leftrightarrow \mathbb{P}(SC\_EVENTS) \wedge$
$EOS \in \mathbb{N}$
variables
$runnable, time, phase,$
$timed, \delta$

invariant
$time \in \mathbb{N} \wedge$
$EOS \geq time \wedge$
$timed \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$\forall t.(t \in ran(timed) \Rightarrow t > time) \wedge$
$\delta \subseteq SC\_EVENTS \wedge$
$dom(timed) \cap \delta = \emptyset \wedge$
$(phase = stop \Rightarrow \delta = \emptyset) \wedge$
$(phase = stop \Rightarrow time = EOS \vee timed = \emptyset)$
initialisation
$time := 0 \parallel$
$phase := init \parallel$
$runnable :\in \mathbb{P}(PROCESSES) \parallel$
$timed := \emptyset \parallel$
$\delta := \emptyset$

New concrete variables help to model scheduling algorithm. The two variables $timed$ and δ correspond to different kinds of event notifications. δ is the subset of SystemC events which have a pending delta-delay notification whereas $timed$ is a function from SC_EVENTS to $\mathbb{N}$ which models the subset of pending time notification events. An important invariant predicate is the disjunction of the two subsets: SystemC events can have only one pending notification and multiple notifications of the same event are resolved by priority rule. New natural variable $time$ models the current time of the system. The variable is very important and a new invariant predicate $\forall t.(t \in ran(timed) \Rightarrow t > time)$ translates that timed notifications indicate future occurrences of events.

The new concrete version of B event Evaluate must now precise behaviors of SystemC events triggered by execution of process p selected in $runnable$. First, the set E models the set of events triggered by an execution of process p ($E \in trigger[\{p\}]$). We must partition the set E:

- let i, a subset of E, the set of SystemC events related to immediate notification.
- let d, a subset of E, the set of SystemC events related to δ-delay notification.
- let t, a subset of E, the set of SystemC events related to timed notification.

These subsets are only composed of explicit invoked events. At this abstract level, the model uses only explicit SystemC events and not events produced by channels updates. These three subsets partition E as defined in the guard of B event Evaluate. The next box presents a part of B event Evaluate guard:

$E \in trigger[\{p\}] \wedge$
$t \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$dom(t) \subseteq E \ \wedge \ d \subseteq E \ \wedge i \subseteq E \wedge$
$dom(t) \cap d = \emptyset \ \wedge dom(t) \cap i = \emptyset \wedge$
$d \cap i = \emptyset \ \wedge \ dom(t) \cup d \cup i = E \wedge$

Rules of priority about multiple event notifications imply important properties on function t and on the new subset of events with pending timed notifications represented by domain of the function $newTimed$. The function is built with the function $timed$ (old set of timed notification events) and the function t which represents events with timed notifications triggered by execution of process p. Because of priority rules presented in section* , we establish these properties:

$newTimed \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$dom(newTimed) = dom(timed \ \lhd\!\!\!\!- \ t) - (d \cup i) \wedge$
$\forall e.(e \in (dom(timed) \cap dom(t)) \Rightarrow newTimed(e) = min(\{timed(e), t(e)\})) \wedge$
$\forall e.(e \in dom(newTimed) \wedge e \notin dom(t) \Rightarrow newTimed(e) = timed(e)) \wedge$
$\forall e.(e \in dom(newTimed) \wedge e \notin dom(timed) \Rightarrow newTimed(e) = t(e))$

In the same way, variable δ is updated by rules of scheduler; immediate notifications are more prioritary than δ-notifications ($\delta := \delta \cup d - i$). In another hand, the set $runnable$ is updated by suppression of executed process p and by addition of the set of processes sensitive to immediate notifications of events of i subset. The new concrete version of B event Evaluate is finally:

Evaluate =
any
$p, E, t, d, i, newTimed$
where
$p \in runnable \wedge$
$E \in trigger[\{p\}] \wedge$
$t \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$dom(t) \subseteq E \ \wedge \ d \subseteq E \ \wedge i \subseteq E \wedge$
$dom(t) \cap d = \emptyset \ \wedge dom(t) \cap i = \emptyset \wedge$
$d \cap i = \emptyset \ \wedge \ dom(t) \cup d \cup i = E \wedge$
$newTimed \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$dom(newTimed) = dom(timed \ \lhd\!\!\!\!- \ t) - (d \cup i) \wedge$
$\forall e.(e \in (dom(timed) \cap dom(t)) \Rightarrow newTimed(e) = min(\{timed(e), t(e)\})) \wedge$
$\forall e.(e \in dom(newTimed) \wedge e \notin dom(t) \Rightarrow newTimed(e) = timed(e)) \wedge$
$\forall e.(e \in dom(newTimed) \wedge e \notin dom(timed) \Rightarrow newTimed(e) = t(e)) \wedge$
$dom(t) \cap \delta = \emptyset \ \wedge \ \forall x.(x \in ran(newTimed) \Rightarrow time < x)$
then
$runnable := (runnable - \{p\}) \cup sensitivity^{-1}[i] \ \|$
$timed := newTimed \ \|$
$\delta := \delta \cup d - i \ \|$
$phase := run$
end

Now, we detail the *update* phase of SystemC simulation. Because of different kind of notifications, *update* phase is more complex. The *update* phase begins when the set $runnable$ is empty. It means that all runnable processes have been executed in the previous *evaluate* phase. This important precondition was present in the guard of first abstract B event Update. The splitting of abstract B event Update produces three concrete events, updateValue, DeltaCycle, ProgressTime.

The concrete event updateValue is a non-deterministic event which adds a subset S to the set δ of events with δ-notifications. It means that sometimes, in *update* phase, SystemC scheduler produces new events notifications. Details will be added in the second refinement. Invariant properties are preserved by activation of this event. New concrete event DeltaCycle models the *update* phase due to pending δ-notifications (see guard of event:$\delta \neq \emptyset$). At this abstract level, the model explains the construction of new list of runnable processes: this is the set of processes sensitive to events with pending δ notification.

updateValue =
any
S
where
$S \subseteq SC\_EVENTS \wedge$
$runnable = \emptyset \wedge$
$S \cap dom(timed) = \emptyset \wedge$
$phase = run$
then
$\delta := \delta \cup S$
end

In the other hand, B event ProgressTime represents the *update* phase due to pending timed event notification. The event is activated only when there is no more δ-delay notifications. B event ProgressTime advances the current simulation time to the time of the earliest (next) pending time event notification. New list of runnable processes is built with $sensitivity$ relation and events which occur at new current simulation time.

DeltaCycle =
select
$runnable = \emptyset \wedge$
$\delta \neq \emptyset \wedge$
$phase = run$
then
$runnable := sensitivity^{-1}[\delta] \parallel$
$\delta := \emptyset$
end

ProgressTime =
select
$runnable = \emptyset \wedge$
$\delta = \emptyset \wedge$
$timed \neq \emptyset \wedge$
$EOS \geq min(ran(timed)) \wedge$
$phase = run$
then
$runnable :=$
$sensitivity^{-1}[timed^{-1}[\{min(ran(timed))\}]] \parallel$
$time := min(ran(timed)) \parallel$
$timed := timed \rhd \{min(ran(timed))\}$
end

Finally, the abstract event HALT is refined into two more concrete events. First, the event noEvent models the ending of simulation, because of lack of event notifications. The simulation stops because of lack of activities: no more event notifications implies no more runnable processes. Second, the event EndingSimulation models ending simulation, because of the simulation time is the ending-of-simulation time and no more events notifications can occur.

EndingSimulation =
select
$runnable = \emptyset \wedge$
$\delta = \emptyset \wedge$
$time = EOS \wedge$
$phase \neq stop$
then
$phase := stop$
end

noEvent =
select
$runnable = \emptyset \wedge$
$\delta = \emptyset \wedge$
$timed = \emptyset \wedge$
$phase \neq stop$
then
$phase := stop$
end

Second Refinement: Complete Model

Channels are introduced in this final refinement. New automaton is not presented because only some transitions (ie guards of events) are strengthened to consider adding of channels and values of them.

New constants are introduced in this final refinement. First, a new set $CHANNELS$ models the set of used channels in a SystemC program. Another set is $VALUE$ which represents the set of abstract values of considered channels. Our model introduces a subset C_EVENTS which represents the set of implicit events of the program. The implicit event is an event used by system to indicate a modification of channel's value. SystemC users can not access directly to this kind of events. Our model introduces the next properties which translate previous remarks:

$$C\_EVENTS \subseteq SC\_EVENTS \wedge$$
$$\forall S.(S \in ran(trigger) \Rightarrow S \cap C\_EVENTS = \emptyset)$$

Another constants are introduced in this model. The function $produce$ is a total function from $CHANNELS$ to C_EVENTS and represents implicit events triggered by modifications of channel. The current model deals with abstract channels and does not detail generation of implicit events for positive and negative sensitivity lists (keywords `sensitive_pos` and `sensitive_neg`). These kinds of sensitivity lists are only used with particular (boolean) channels and, in the same way, we could model these lists.

A new variable $value$ is introduced to represent the current values of channels. A second new variable $newValue$ is introduced to construct the new valuation of channels after *update* phase. The two variables $value$ and $newValue$ are total functions from $CHANNELS$ to $VALUE$.

New concrete version of event Evaluate considers channels. A partial function f is introduced to represent modifications of channels made by process p. The variable function $newValue$ is built with the partial function f during the *evaluate* phase. It is easy to prove that the new version of event Evaluate refined the oldest abstract version.

Evaluate =
any
$p, E, t, d, i, newTimed, f$
where
$p \in runnable \wedge$
$t \in EVENTS \nrightarrow NATURAL \wedge$
$E \in trigger[p] \wedge$
$dom(t) \subseteq E \wedge$
$newTimed \in EVENTS \nrightarrow NATURAL \wedge$
$d \subseteq E \ \wedge i \subseteq E \wedge$
...
$f \in CHANNELS \nrightarrow VALUE$
then
$runnable := (runnable - p) \cup sensitivity^{-1}[i] \parallel$
$timed := newTimed \parallel$
$\delta := \delta \cup d - i \parallel$
$newValue := newValue \ \lhd\!\!\!\!- f \parallel$
$phase := run$
end

The new concrete version of event updateValue is deterministic and explain the adding of δ-delay notification during *update* phase. When values of channels are updated, δ-delay events notifications occur, when the new value is different from old value. The set δ is updated with these new δ-delay notifications. The event explains the behavior of scheduler when multiple channels modifications: only the last modification is considered.

updateValue =
select
$runnable = \emptyset \wedge$
$value \neq newValue \wedge$
$phase = run$
then
$value := newValue \parallel$
$\delta := \delta \cup produce\left[\left\{ \begin{array}{l} c \mid c \in CHANNELS \wedge \\ value(c) \neq newValue(c) \end{array} \right\}\right]$
end

The new concrete versions of other events are not too different from abstract versions and for limited size reasons we do not present the concrete versions of B events.

Producing a B model from a SystemC program

From a particular SystemC program or design, we can produce a B event model which represents the simulation of the program by the scheduler. The new B model must be a particular instantiation of abstracts models of scheduler. For each process of SystemC program we produce events (one at least) which represent execution of process. The set of events produced must refine event Evaluate from abstract models, which represent abstract executions of abstract processes.

Abstract sets and constants are concretized with particular values of program. Sets and constants of instantiated model must preserve properties of abstract sets and constants. For example, concrete set $CHANNELS$ is composed by the channels used in Systemc programs modelled. In the same way, abstract set $PROCESSES$ is the set of particular processes of current SystemC program.

EXAMPLE

Our abstract models can be instantiated to simulate execution of particular program by the SystemC scheduler. Instantiation of abstract scheduler for particular program is very easy: sets, constants and variables are specified for particular SystemC program studied. Abstract event Evaluate is split into particular events, which model execution processes of particular SystemC program. As previously announced, other abstract events are unchanged: algorithm of SystemC scheduler did not evolve with programs simulations.

Sets and constants

We use a toy example presented in section page 5. Instantiation is made with the concrete sets:

Abstract set $PROCESSES$ is instantiated by three processes (`proc1, proc2, proc3`) of the toy example. Abstract set SC_EVENTS is instantiated by SystemC events $e2$ and $e3$, defined by user, and by SystemC event e which is an implicit event. In the same way, abstract set $CHANNELS$ is instantiated by only one element, $count$ channel. Event e is gluing to channel $count$ in relation $produce$. Abstract set $VALUE$ is instantiated by the integer set.

$PROCESSES = \{proc1, proc2, proc3\}$
$SC\_EVENTS = \{e, e2, e3\}$
$CHANNELS = \{count\}$
$VALUE == \mathbb{N}$

Concrete constants are defined for the particular SystemC program considered. It is easy to show that concrete constants satisfy abstract properties of abstract constants. The major part of properties is type-checking and concrete constants trivially preserve these properties.

$C\_EVENTS = \{e\}$
$produce = \{count \mapsto e\}$
$trigger = \{proc1 \mapsto \{e2\},$
$\quad proc1 \mapsto \{e3\},$
$\quad proc2 \mapsto \emptyset,$
$\quad proc2 \mapsto \{e3\},$
$\quad proc3 \mapsto \emptyset\}$
$sensitivity = \{proc1 \mapsto e, proc2 \mapsto e2, proc3 \mapsto e3\}$

variables
$runnable, time, phase,$
$\delta, timed, value, newValue$
invariant
$\mathbf{dom}(timed) \subseteq \{e3\} \ \wedge \ e3 \notin \delta \wedge e2 \notin \delta$
initialisation
$time := 0 \parallel$
$runnable := \{\text{proc1}\} \parallel$
$phase := init \parallel$
$timed := \emptyset \parallel \delta := \emptyset \parallel$
$value := \{count \mapsto 0\} \parallel$
$newValue := \{count \mapsto 0\}$

The invariant predicates of model precise relations between concrete instantiated variables. In the current SystemC program, only event $e3$ is concerned by time event notifications. This fact is translated into an invariant predicate $\mathbf{dom}(timed) \subseteq \{e3\}$.

Instantiation: concrete event EvaluateXXX

From structure of the three processes of listing page 5 we derive five B events:

- Process `proc1` contains conditional statement and it implies that two different executions can occur. Two different events model the process. Guards of events are disjunctive.
- As process `proc1`, process `proc2` uses a conditional statement. Equivalently, two B event simulate behaviors of process `proc2`.
- Process `proc3` does not contain conditional statement. Only one event is needed to represent its execution.

EvaluateProc1Then =
select
$proc1 \in runnable \wedge$
$value(count) < 10$
then
$runnable := runnable - \{proc1\} \cup$
$\quad sensitivity^{-1}[\{e2\}] \parallel$
$newValue := newValue \ \lhd\!\!+$
$\quad \{count \mapsto value(count) + 1\} \parallel$
$phase := run$
end

EvaluateProc1Else =
select
$proc1 \in runnable \wedge$
$value(count) \geq 10$
then
$runnable := runnable - \{proc1\} \parallel$
$timed := \{e3 \mapsto$
$\quad \min(\mathbf{ran}(timed) \cup \{time + 5\})\} \parallel$
$phase := run$
end

Finally, the two previous B events simulate executions of process `proc1` during SystemC simulation. All instructions of each block of conditional statement are translated/considered in the corresponding event. Guards of B events represent test of conditional instruction and context of simulation.

As previously, two next events are built and model behavior of process `proc2` during SystemC simulation. This two events give operationnal semantics of process `proc2`.

EvaluateProc2Then =
select
$proc2 \in runnable \wedge$
$value(count) < 15$
then
$runnable := runnable - \{proc2\} \parallel$
$newValue := newValue \lhd\!\!\!\!- \{count \mapsto value(count) + 2\} \parallel$
$phase := run$
end

EvaluateProc2Else =
select
$proc2 \in runnable \wedge$
$value(count) \geq 15$
then
$runnable := runnable - \{proc2\} \parallel$
$timed := \{e3 \mapsto \min(\mathbf{ran}(timed) \cup \{time + 4\})\} \parallel$
$phase := run$
end

The next event represents executions of process `proc3`.

EvaluateProc3 =
select
$proc3 \in runnable$
then
$runnable := runnable - \{proc3\} \parallel$
$newValue := newValue \lhd\!\!\!\!- \{count \mapsto 0\} \parallel$
$phase := run$
end

CONCLUSION AND OPEN ISSUES

The refinement is the key concept for developing complex systems, since it starts by a very abstract model and incrementally adds new details of the set of requirements; the main result of our work is the production of a formal model for the SystemC scheduler with proved invariant properties correct with respect to the properties required by the scheduling process; the incremental proof-based construction of the formal model allows us to produce a understandable and well structured documentation for the SystemC simulation. The complexity of the proof process is indicated by the assessment:

B Models	Automatic Proofs	Interactive Proofs	%automatic/interactive P.
AbstractScheduler	4	0	100/0
Scheduler1	22	5	82/18
Scheduler2	10	2	84/16
Instanciation	20	10	66/34
Total	56	17	77/23

The SystemC scheduler allows us to instantiate parameters according to the current SystemC program to simulate and hence we obtain an instance of the scheduler that can be used for simulation and for further studies of the current instantiated SystemC program. Another result is directly related to the methodology for producing an operational semantics for a given algorithm and the refinement proves that the definition of an operational semantics can be incrementally written and can be proved by checking proof obligations. However, the result is applied to our case study which is an effective tool for measuring the quality of audio/video signals in the Digital Video Broadcasting (DVB) [9]; the tool is built from the B modelling and the SystemC code is certified by the use of a proof assistant. Further works should implement a tool for helping the manipulation of abstract scheduler and for checking conditions over the instantiation for a given SystemC program; the tool should integrate a function for defining the parameters to instantiate in the scheduler model. New case studies should be developed, as well as others properties over SoC should be taken into account like confidentiality, access control, ...

References

[1] Abraham, D., Cansell, D., Ditsch, P., Méry, D., and Proch, C. Synthesis of the QoS for digital TV services. In *IBC'05, The Netherlands* (2005).

[2] Abrial, J. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, 1996. ISBN 0-521-49619-5.

[3] Abrial, J.-R. B# : Toward a synthesis between Z and B. In *ZB'2003 - Formal Specification and Development in Z and B* (Turku, Finland, June 2003), D. Bert, J. P. Bowen, S. King, and M. Waldén, Eds., vol. 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, Springer, pp. 168 – 177.

[4] Back, R. J. R. On correct refinement of programs. *Journal of Computer and System Sciences 23*, 1 (1979), 49–68.

[5] Cansell, D., Culat, J.-F., Méry, D., and Proch, C. Derivation of SystemC code from abstract system models. In *Forum on specification & Design Languages - FDL'04, Lille, France* (Sep 2004).

[6] Cansell, D., and Méry, D. Logical foundations of the B method. *Computers and Informatics 22* (2003).

[7] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*. The MIT Press, 2000.

[8] ClearSy. Web site b4free set of tools for development of b models. `http://www.b4free.com/index.php`, 2004.

[9] European Broadcasting Union. Digital video broadcasting (DVB)- measurement guidelines for DVB systems. Tech. Rep. TR 101 290 v1.2.1., ETSI, 05 2001.

[10] Gawanmeh, A., Habibi, A., and Tahar, S. An executable operational semantics for SystemC using Abstract State Machines. Tech. rep., Concordia University, Department of Electrical and Computer Engineering, mar 2005.

[11] Glässer, U., Börger, E., and Müller, W. Formal definition of an abstract VHDL'93 simulator by EA-machines. In *Formal Semantics for VHDL* (1995), C. Delgado Kloos and P. T. Breuer, Eds., Kluwer Academic Publishers.

[12] Méry, D., Cansell, D., Proch, C., Abraham, D., and Ditsch, P. The challenge of QoS for digital television services. *EBU Technical Review 302* (Apr 2005).

[13] Mueller, W., Dömer, R., and Gerstlauer, A. The formal execution semantics of SpecC. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis* (New York, NY, USA, 2002), ACM Press, pp. 150–155.

[14] Open SystemC Initiative. *SystemC 2.0.1 Language Reference Manual*, 2004.

[15] Ruf, J., Hoffmann, D., Gerlach, J., Kropf, T., Rosenstiehl, W., and Mueller, W. The simulation semantics of SystemC. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe* (Piscataway, NJ, USA, 2001), IEEE Press, pp. 64–70.

[16] Ruf, J., Hoffmann, D., Kropf, T., and Rosenstiel, W. Simulation-guided property checking based on a multi-valued AR-automata. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe* (Piscataway, NJ, USA, 2001), IEEE Press, pp. 742–748.

[17] Salem, A. Formal semantics of synchronous SystemC. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 376–381.

[18] SOCFV Project. System on chip formal verification home page. `http://www.ensta.fr/~hammami/resproj.SOCFV.html`, 2004.

[19] SystemC. Official web site of SystemC community. `http://www.systemc.org/`, 1999.