



## **SystemC: Key Modeling Concepts Besides TLM to Boost your Simulation Performance**

*Martin Schnieringer, VaST Systems Technology, Munich Germany*

*Kevin Brand, VaST Systems Technology, Sydney, Australia*

## Abstract

The virtualization of System on Chip (SoC), Electronic Control Units (ECUs) and other electronic systems is used to explore (micro-) architectures early in design cycles to develop and verify software. Key to those Virtual System Prototypes (VSP) is the simulation speed and accuracy dependent on the use case. The latest SystemC standards, such as OSCI TLM 2.0, OCP IP TL2, etc., focus on the standardization of the bus interfaces for peripheral devices connected in a virtual system simulation. Recent proposals have extended these standards in regards to the runtime configuration of the hardware simulation models. However, on different levels of abstraction, such as programmer's view (PV) and programmer's view with timing (PV-T), these standards do not ensure fast simulation speeds for processor centric systems. They also do not promote efficient modeling techniques for the behavior of the simulation internet protocol (IP) block.

This is important as inefficient modeling of the behavior can easily bring down your simulation speed into the KIPS range (kilo instructions per second) instead of the MIPS range (million instructions per second) requested by the software engineers who develop applications on virtual platforms.

This paper will demonstrate a methodology for modeling your virtual hardware efficiently by minimizing events yet still being accurate on cycle boundaries. As an example we will use a simple watchdog timer to illustrate the basic concepts of a SC\_METHOD based modeling approach that will allow us to maintain simulation speeds in the MIPS range. In the end we will also show that complex systems with a decent number of complex devices can still maintain the promised simulation speeds in the MIPS range.

## 1. Motivation

Virtual System Prototypes are mainly used by two groups of engineering communities: system architects and software developers. While speed proves to be more important for software developers, accuracy seems to be more important for system architects. Yet, the software developer also often also requires a high degree of accuracy, for example in real-time critical reactive systems, or multimedia applications with stringent throughput requirements. In addition, the system architect is moving more and more towards software-driven architecture analysis and optimization strategies, in which real software loads (OS, Multimedia, etc.) are used for stimuli. For both communities, the typical opinion is that two different sets of models are required, each differentiating by either speed of the model or accuracy. The industry has been moving strongly towards providing speed and accuracy in integrated solutions, with only the best being able to provide speed and accuracy in one model, known as 1-source models. Still, the question remains, how can Virtual System Prototypes be completed by the individual modeling teams and remain fast and accurate? This paper discusses how peripheral models of IP blocks should be modeled so that they enable full Virtual System Prototypes to execute embedded target software at two digit MIPS speeds yet simulate cycle-accurate at a pin and bus boundary level.



## 2. Minimizing the Number of Events

The simulation speed depends on many factors such as

- Size of the system to model
- Level of detail of the system model
- Performance of the host machine that is executing the simulation

The simulation speed for your simulator is affected by the number of events that have to be processed by your simulation kernel for a given simulation time. In order to maximize simulation performance, it is necessary to minimize the number of events. Fewer events lead to higher MIPS rates hence shorter development time for the application software developer. It is also important to note that fewer events do not necessarily lead to less granularity for the system architect. There are many different types of events and in the following we discuss these and how to minimize their occurrence whilst maintaining cycle accurate boundaries. This paper will discuss Protocol and Interface related events, Behavioral events and Clock events.

### Protocol and Interface Events

When a real device communicates with other real devices in a system, these communication transitions translate to simulation events in the virtual world. Using transaction level interfaces to represent these is the obvious way to remove unnecessary events. Function calls, also called Interface Method Calls (IMCs), are used to communicate over a virtual bus infrastructure. The events that could be saved and hence do not need to be modeled are the ones typically seen in a pin-level accurate interface such as handshaking signals, read and write strobes etc. Of course minimizing the number of events is possible for all kind of protocols. Consider a serial interface and the number of events caused by shifting a 32-bit data word into the receiver resulting in each bit shift causing an event at the serial clock edge. The number of events could be significantly reduced by transmitting the serial data at once, or in parallel, and keep track of the start time of the transfer and end time. From a software developers perspective we are just interested in how long it takes to receive that serial data word and visibility into the partly shifted data is not required. Using this modeling style we reduced the number of events – namely to two, one for the start and one for the end time – but kept the accuracy for application software developer and system architect.

Minimizing the number of interface related events is a good starting point but does not ensure great simulation speeds. When a Transaction-Level Modeling (TLM) interface is used but the behavior of your device is modeled in a register transfer level (RTL) fashion (all internal states are updated each clock cycle) it becomes obvious that modeling edge clocked states that are not visible from the outside results in simulation performance loss. These internal state behavior events are required to be abstracted.



## Behavioral Events

To better illustrate efficient modeling techniques a simple watchdog timer is used. The bus slave device has two memory mapped registers: one to control the watchdog timer and a single count register.

The Boolean output port should go high for one clock period when the timer register counted down to zero. The inefficient RTL style approach would be to decrement the counter register each clock period and it would be even worse doing that in a clocked SC\_THREAD causing a context switch each time.

The most important event for the simulation in terms of the watchdog timer is the Boolean output port change. Therefore the port change is only interested in one event at a precise time and that is the counter reaching zero at the clock edge that caused it.

To model the behavior of driving the output port at the precise clock edge, a SC\_METHOD can be implemented that is sensitive to an sc\_event. That event is notified in the simulation future using the .notify() method for SystemC events. This models a counter match event that will happen at a known time. It is quite simple to calculate the time in the future and hence when to trigger the SC\_METHOD. The time in the future is determined by the following equation:

$$[\text{watchdog counter value}] \times [\text{clock period}]$$

The clock period could be communicated by using an input port of type unsigned int or sc\_time in the peripheral device. For the later of the two, one the sc\_trace() must be overloaded to support ports of type sc\_in<sc\_time>. This approach has the benefit that the time unit of the clock period could be passed as well to avoid confusion when IP blocks of different sources are mixed. In the following example we use an input port of type unsigned int and the time unit is 1 picosecond, which is the default resolution of the SystemC simulator in use.

```
sc_event MatchEvent;
SC_METHOD(MatchCallback);
sensitive << MatchEvent;
dont_initialize();

MatchEvent.notify(TimerClkPeriod * Value,
SC_PS);

...

void watch_dog::MatchCallback() {
TimerInterrupt.write(0x1);
```

```
}
```

To drive the output port low again we could schedule another SC\_METHOD one clock period of time in the future using the same approach. Then another match event could be scheduled in the future for the next count down of the timer. This SC\_METHOD based approach is sometimes also called a “Callback” based approach. These are simple techniques to accurately model this interrupt event efficiently, however, there are other external actions that have an impact on the behavior of the watchdog model.

The following actions all impact the design:

- Reset of the device
- Access of the memory mapped register by the software target code
- Change of the clock period

The reset of the device is the simplest scenario. Pending sc\_events are canceled and registers are loaded with their reset values.

```
sc_in<bool> Reset;  
  
SC_METHOD(ResetChanged);  
sensitive_pos << Reset;  
dont_initialize();  
  
void watch_dog::ResetChanged() {  
    //Reset member vars, registers, ports  
    TimerControlReg = 0x00;  
    //Cancel scheduled events  
    MatchEvent.cancel();  
}
```

Depending on the design of model, other actions may need to be taken on this reset event (e.g. rereading



the clock period of a clock input port). From the perspective of the software developer, the next action that the model may experience is reading and writing to the watchdog's memory mapped registers, especially after the watchdog timer has been enabled. Since this action can happen at any time, the correct count value must be returned to the target software. The challenge is to reflect the accurate state of the counter even though the current state is not known, as we have not modeled the decrementing counter.

In order to return the correct counter value a new member variable is introduced. This variable stores the simulation time (`sc_time`) of the last "update" to the model. It is updated whenever an access to a memory mapped register occurs that has an impact on the counter state (e.g. a write access to the control register or a read/write access to the counter register). The member variable is named `mLastUpdate`. At any point in time during the simulation, the correct return value of the counter register can be calculated using the following equation:

$$\text{value} = (\text{NowTime} - \text{mLastUpdate}) / \text{clock\_period}$$

`NowTime` is the simulation time when the read access to the counter register occurs. To ease modeling calculations later, clock ticks can be used instead of absolute times. A write access to the watchdog control register is still simple. Cancel all pending events and only re-schedule the "Match" event when the enable bit `0x80` is set. The code fragment below shows a possible implementation of the address decode function:

```
watch_dog::write(int *data, unsigned int
address) {
double NowTime = sc_simulation_time();
MatchEvent.cancel();
//Calculate address offset
int Offset = (address - m_start_address)
& ~(sizeof(unsigned int) - 1);
switch (Offset) {
case WatchDogCtrlOfst:
WatchDogCtrl = *data;
if(WatchDogCtrl & 0x80) { //Schedule
```



```
// an MatchEvent
MatchEvent.notify((mWatchDogClkPeriod
* WatchDogValue), SC_PS);
.. ..
}
.. ..
```

A write access to the watchdog counter register needs the Match event to be re-scheduled.

```
switch (Offset) {
.. ..
case WatchDogValueOfst:
WatchDogValue = *data;
if(WatchDogCtrl & 0x80) { //Schedule
//MatchEvent based on new value
MatchEvent.notify((mWatchDogClkPeriod
* WatchDogValue), SC_PS);
}
}
```

In this simple example above the bus interface and the watchdog timer rely on the same clock period. When different clock domains are used things become more complex. For example, the counter register is manipulated over the bus interface and the re-scheduling of the Match event must take into account the delta between the two clock domains. The helper function below does the job and returns that delta. As an argument we pass the clock period of the other clock domain.

```
double
watch_dog::ScClockSyncTime(const double&
Period) {
double NowTime =
sc_simulation_time();
```

```
double IntegerPart;  
if(modf(NowTime/Period, &IntegerPart)) {  
return(IntegerPart+1)*Period - NowTime;  
} else { //Clock domains are aligned  
return 0;  
}  
}
```

The example above shows that the number of events can be reduced dramatically by using a different modeling approach. To get an idea how many behavioral events we saved we assume a simple use case and do not consider events created by accesses to the bus interface. The clock period does not change and the system is operating correctly. It reloads the watchdog counter register in time so that the output port is not driven high. In that case not a single behavioral event has been fired, driving the output port high. The RTL style would have fired many events, one for each decrement of the watchdog counter register. This will happen continuously after the watchdog counter register has been reloaded again. This modeling approach can be summarized by the following statement, “If there are no interactions with the model interface, there is no need to execute any model code”. The model is still timing accurately at the interface boundaries. The drawback is that a separate model for simulation and implementation is needed. This approach can also be used in the modeling of the internal state machine of any device model because the “state” only needs to be updated when the device “interacts” with other elements in the simulation.

## Clock Events

In general a pulsing clock causes many events and easily reduces the simulation speed to the KIPS range. In the previous section the clock period has been used to model timing accurate behavior. One additional advantage of this style is that support for a dynamic clock support. Whenever a clock generation unit communicates a new clock period, the watchdog model connected to that clock signal can change its behavior accordingly. Again, changes of behavior need only be modeled if actions occur on the interface. SystemC does not support dynamic clocking because the period, duty cycles, etc., are set during construction of the `sc_clock` object. It is obvious that we need dynamic clock support for today’s SoCs especially in the handheld market where battery life does matter.

The function below demonstrates that when the value of the clock period input port changes the device state can be updated. As the clock period has an impact on the Match event all future events are canceled and rescheduled. Consequently the current value of the watchdog counter register, based on the previous clock period, has to be calculated. Finally the Match event is rescheduled based on the



new watchdog timer period stored in mWatchDogClkPeriod.

```
watch_dog::WatchDogClkPeriodChanged() {
    MatchEvent.cancel();
    //Nothing to do when Watch Dog disabled
    if(WatchDogCtrl & 0x80) { //Enabled
        double NowTime = sc_simulation_time();
        //Calculate WatchDogValue
        WatchDogValue = WatchDogValue -
            (NowTime - mLastUpdate) /
            mWatchDogClkPeriod;

        mWatchDogClkPeriod=WatchDogClkPeriod-
>read()); //Read new period from port
        if(mWatchDogClkPeriod) {
            //When WatchDogClkPeriod is zero power down
            MatchEvent.notify(WatchDogValue *
            mWatchDogClkPeriod, SC_PS);

        }
    } else { //Only update mWatchDogClkPeriod
        mWatchDogClkPeriod = WatchDogClkPeriod-
>read();
    }
}
```

The callback function above to handle dynamic clocks also accepts a period of zero what means we completely switch off the device to save power (and conveniently simulation events). That's all it takes to write an accurate and fast SystemC model that has little impact on simulation speed. Of course the coding methodology demonstrated here needs a different way of thinking compared to the RTL style often adopted by the hardware engineer.



### 3. SystemC Deficiencies

SystemC `sc_clock` has no support for dynamic clocking. While the clock period via an input port overcomes that shortcoming, there is no kernel support to automatically re-schedule events when their time base changes. Providing that dynamic clock support is a convenient solution, this behavior has to be implemented in each SystemC device and in a base class. It also can be tricky when the model has several clock domains that have to be taken into account. Imagine a bridge between two buses clocked at different speeds. An incoming bus transaction at time  $n * t_{IN}$  has to be re-transmitted at time  $m * t_{OUT}$

Where:

$t_{IN}$  is the clock period of the incoming bus

$t_{OUT}$  is the clock period of the outgoing bus

$n$  and  $m$  are the number of clock cycles

The model developer then manually has to calculate the timing delta to the next edge of the clock on the outgoing side. Again helper classes are extremely useful in that case in order to convert the clock domains.

Another deficiency is that there is no means to pass arguments to the `SC_METHOD`. This is helpful when you want to use a callback function to handle multiple instantiations of a sub-functionality. An example is when you have an array of  $N$  timers in your model and you use a single callback function to handle all  $N$  timers. It is possible to pass the number of the timer array you want to manipulate through the argument however the SystemC syntax does not support that. Another minor issue is that SystemC does not warn you when you schedule an event that has already been scheduled. It will just pick the next one on the simulator event wheel. For debugging your model it would be helpful to get a warning.

### 4. Outcome

The discussed modeling methodology has been used to model a SystemC subsystem containing an interrupt controller, a UART and a timer model.

That subsystem has been instantiated in a VaST Virtual System Platform with an ARM926e processor model and a fully arbitrating Advanced High performance Bus (AHB) bus (both models using VaST proprietary modeling methodology). The target software executing on the ARM model was embedded Linux and the simulation still ran at 50 MIPS. Not only is the modeling style of the peripheral models important with regards to simulation speed but also with the other key components of a VSP such as the simulator, the buses, bridges, memory and the Virtual Processor Models (VPMs). The VaST simulator

can switch tasks efficiently within a few host machine cycles and the bus/bridge models typically ensure 1.5 million transactions per second. They are also cycle accurate on a transaction-level and fully arbitrating on a cycle basis. The VaST VPMs use well-known techniques to run in the double digit MIPS range and ensure timing accuracy. So although the SystemC peripherals are on a low bandwidth path in terms of events/ access on the Linux embedded target code, it is seen that they contribute very little in terms of simulation performance. This would not be the case if they were modeled in a clocking RTL style.

## 5. Summary

The SystemC approach has proven to support accuracy and speed in a single model by reducing the number of events. Further recommendations are to avoid SC\_THREADS when possible and use non-blocking interface method calls to prevent the user from using SC\_THREADS. Those IMCs should support timing annotation as well as the passing of an event that could be fired by the slave when the bus transaction has been processed. On the bus interface PV modeling could be used in the first step to keep the modeling effort low but add timing details later on. With the use of annotated time the notion of scheduling future events can be followed. These guidelines enable the simulation of complex systems at reasonable speeds. SystemC does not have support for that methodology built in the simulation kernel, which is why cumbersome calculations of event firing in the future are necessary.

## 6. References

- [1] Groetker Thorsten, Liao Stan, Martin Grant, Swan Stuart: System Design with SystemC. 2002
- [2] IEEE Standard SystemC® Language Reference Manual
- [3] OSCI TLM 2.0 Draft 1