M.Sc. Thesis
Master of Science in Engineering

SystemC Tracing for Observability and Profiling

Author: Roman Birca

Supervisors:

Martin Schoeberl, Department of Applied Mathematics and Computer Science, DTU, Denmark

Ola Dahl, Ericsson AB, Stockholm

Abstract

The increasing complexity of modern systems on chip requires the employment of system-level simulations to enable the co-development of hardware and software, in order to meet time-to-market windows. SystemC is a C++ library that facilitates electronic system-level design (ESL) to address this issue. The difficulty of debugging such systems remains present. Using system-level simulations, in the form of virtual platforms, makes it possible to monitor and trace the execution of a system to a level that otherwise requires the designed hardware to be present. This types of monitoring and tracing can serve as complementary to hardware-based tracing, where the hardware needs to contain subsystems incorporated specifically for tracing and debugging purposes.

The purpose of this thesis was to instrument the Accellera reference SystemC simulation kernel to provide simulation traces, with application to Ericsson's SystemC-based virtual platforms. Along with the instrumented kernel, a set of tools for analyzing and visualizing the trace data was developed. The instrumented kernel was used to run simulations of a virtual platform containing thousands of SystemC processes across hundreds of SystemC modules. Traces from these simulations were analyzed using the set of tools on a series of case studies, to illustrate the possibilities for exploring virtual platforms. The case studies were selected with the purpose of illustrating increased understanding of platform complexity, run-time profiling, and collection of data for debugging and trouble-shooting support.

Table of Contents

List of Figures and Listings

1. Introduction

The design complexity of large scale hardware such as SoCs (system on chip) is increasing. There are requirements on functionality and performance, and there are constraints from a time-to-market perspective. In addition, the amount and complexity of the software running on an SoC is increasing. This, in combination with requirements on the resulting product, such as a Radio Base Station, or a complete network, puts high demands on a coordinated development of hardware and software.

It is usual for hardware design to take extensive periods of time, depending on design complexity, and whether the design is full-custom or uses a standard cell library. After rigorous searching for errors in the design, it is sent to a silicon foundry for manufacturing. This was the case also in the early days of ASIC design, where no simulation of the ASIC was used when developing the software that should run on the developed ASIC. Software development could only start once the hardware was delivered, otherwise there was a risk of incompatibility between software and hardware. In the worst case, design errors that rendered the entire chip useless could be discovered, further delaying software development.

The advent of HDLs and simulators allowed design abstractions to be raised from gate-level to register-transfer-level (RTL). Advances in verification methodology enabled hardware to be verified in a more complete and rigorous manner. This significantly reduced the risk of manufacturing faulty hardware. While an improvement for design and verification, the impact for accelerating software development was still minor, as running large amounts of software on RTL simulations was infeasible. Since hardware and software still had to be tested separately, a high risk of errors when combining the two into the final product remained.

To further optimize the development of hardware and software systems, transaction-level modeling (TLM) can be used. This allows abstraction to be raised to a functional and bit-accurate level, and simulation speed can be increased significantly. In this way, complete systems can be modeled and simulated with sufficient accuracy for software development and functional verification. In addition, these system models can be used for software development before the target hardware is available. This technology allows for a feedback loop between the two domains (HW/SW), potentially optimizing each other, allowing design adjustments of hardware, and software, in a coordinated manner.

1.1 Virtual Platforms

At Ericsson, SystemC [1] and TLM [2] are used to model digital hardware used in radio base stations. The models are integrated into a simulator, referred to as a virtual platform. The virtual platform is used to bridge hardware and software development teams.

The virtual platform executes concurrent target software, running on several processors and interacting with a variety of hardware blocks. In addition, there is often interaction with the outside world, using test tools for logging and input stimuli handling, and with debuggers for setting breakpoints and watchpoints.

The virtual platform contains models for hardware components, implemented as SystemC modules, with SystemC processes representing the simulated behavior of the different hardware components, and with TLM sockets connecting the different models.

The virtual platform models the functional behavior of the underlying hardware. It also models timing, by annotating functional transactions. This approach is referred to as a loosely timed modeling style [2]. The goal is to achieve a timing accuracy that is good enough for timing-dependent software, such as time triggered interrupts, software timers, and other time-dependent functionality in an operating system, e.g. time-out functionality in a message passing mechanism or suspending a thread for a certain amount of time.

The virtual platform is developed by a distributed team, with sub-teams located at different sites. The overall platform, with its different variants representing different ASICs and boards, contains several hundred SystemC modules and several thousand SystemC processes.

1.2 Model Observability and Profiling

When working with the virtual platform, e.g. when solving a bug report indicating a fault in software or in the modelled hardware, or in both, software debuggers as well as debuggers attached to the virtual platform itself are used. In addition, different types of tracing, such as tracing of messages sent between hardware modules, can be used. This type of tracing gives increased observability, and can be used as a complement to a traditional, breakpoint oriented, debugging strategy.

It is possible, when using a commercial SystemC implementation, to use dedicated tools for debugging and tracing, provided by the vendor [3] [4]. These tools are not available when using the open source reference SystemC implementation from Accellera [5].

It is the purpose of this thesis to investigate how model observability can be increased by modifying the reference SystemC implementation from Accellera to provide tracing and recording capabilities. The modifications are done with the goal of complementing the debugging strategies currently used at Ericsson.

1.3 Contributions

An instrumented SystemC kernel that provides simulation traces has been developed. The implemented traces include SystemC process creation, suspension, and resumption, calls to the different SystemC wait functions, and SystemC event notifications.

Along the tracing capabilities, appropriate tools for analyzing and visualizing trace data are provided. Some tools are used to measure the difference between two traces. Others are used to visually represent simulation behavior. Behavior in this case can constitute how SystemC threads suspend, thus providing a means for investigating effects related to timing, e.g. measuring how often SystemC threads execute. Such measurements can serve as input data when determining the best trade-off between frequency of SystemC-kernel synchronizations and simulation speed.

Another behavioral aspect of a platform is how threads interact with each other during simulation. By tracing the event-driven interactions between threads, it is possible to obtain information that can be used in debugging scenarios, where for example the order of events is of importance, or when determining if all notifications of events are caught, or if some notifications are lost, due to simulator bugs.

General process activity can be used to generate a heatmap of platform usage, which can be used to determine test coverage, showing how test cases stimulate different parts of a virtual platform.

1.4 Outline

In chapter 2 related works are introduced and reviewed. Chapter 3 describes the component parts and the available ways for modelling timing in SystemC. Chapter 4 introduces the notion of virtual platforms, how they are used at Ericsson, and how they can be traced.

The following chapters describe the work done during this thesis. Chapters 5 and 6 present the integration of tracing capability into the SystemC kernel and how traces obtained from running simulations using this kernel can be analyzed and visualized. The report ends with some case studies and conclusions presented in chapters 7 and 8 respectively.

2. Related Work

In [6] the authors stress the importance of visualization to aid the exploration of complex designs. Considering the lack of built-in functionality for aiding design exploration in SystemC, the authors propose an implementation for automatic design visualization for SystemC models.

They achieve this by extending and modifying the SystemC library so that the full hierarchy of a design is extracted before running a simulation. The data is extracted using functions for hierarchy traversal already present in the SystemC library. As such, the kernel modifications are minimal. Finally, the authors use a commercial tool to visualize the data, which is obtained from models for a bus arbiter and a RISC processor.

In [7] Hartmann et al. propose the implementation of tracing SystemC models to extract platform activity attributed to active software. Their solution introduces extra-functional quantities to support power consumption and performance estimation. These quantities include state quantities for describing the state of a system at a given time (e.g. ambient temperature), and process quantities for describing state changes that have a duration (e.g. number of bytes in a transaction).

The other aspect of their implementation is the usage of so-called timed value streams. The timed values represent state and process quantities. The timed value streams can be consumed by different stream processors for on-line filtering of data, i.e. preprocessing trace data before saving it for offline processing. The authors conclude that using this framework enables the creation of different stream processors to generate data in different formats, depending on the use case. As an example, the authors suggest saving data as value chain dump files which can then be visualized by many established commercial tools.

In [8], Rogin et al. present an alternative approach to aiding design exploration. Instead of modifying parts of the SystemC library or providing tools for writing models such that they can be traced, the authors implement a SystemC-aware debugger.

To achieve this, the authors used the freely available and open source debugger GDB as a baseline. This way, the authors take advantage of the support for function level debugging already present in GDB, on top of which they built system level debugging capabilities. In the context of SystemC, such capabilities include setting high-level breakpoints, e.g. on a specific event notification, or retrieval of simulation information, e.g. signal paths.

In [9], Lagraa et.al present a methodology to detect memory contention in concurrent software using a simulated multi-processor platform. The authors begin by narrowing the scope of the interested traces to load and store instructions These traces are obtained from running video decoding software on a virtual, SystemC-based, multi-processor

platform containing different number of MIPS32 processors that were instrumented for tracing memory accesses.

The traces are processed using a moving window algorithm (introduced in the same paper) to identify high latency contention windows. The authors identify the functions responsible for the high latencies by looking at function frequencies inside the contention windows. Finally, the authors use the LCM [10] frequent itemset mining algorithm to determine if there is any relation between the frequently occurring functions.

In [11] the authors describe the implementation of a tool for capturing and visualizing SystemC simulation traces. It is pointed out that the use of common profilers such as Valgrind [12] is possible but impractical as it will obscure SystemC-specific features. The authors then show that a variety of visualizations in the form of graphs and tables can be produced from simple data, such as when processes yielded control to the kernel and for how long. These visualizations can provide an overview of simulations as well as aid fault discovery in the simulated models. The paper [11] does not provide details on how the SystemC kernel was instrumented.

3. SystemC & TLM

SystemC is a system-level design library written in C++. The library comes with its own run-time environment, commonly referred to as the SystemC kernel. The SystemC kernel is responsible for scheduling SystemC processes, in an event-driven way, while moving the simulated time forward, as dictated by the processes and their interactions.

SystemC can be used to model a variety of use cases, however, it is most useful in what is called electronic system-level design (ESL). Here, SystemC enables development of hardware and software in parallel through models which can be reasonably accurate and fast. A typical goal is to strive for a more cost-optimized partitioning of hardware and software functions and a reduction of faults in the final product.

3.1 The SystemC Module

Common HDLs such as VHDL and Verilog aid hardware design by offering mechanisms for arranging different components of a system into a hierarchy. The top-most module, analogous to a real-life packaged chip, simply "holds" the interconnected system components, e.g. the CPU, RAM, ROM, and I/O of a microcontroller. The modules lower in the hierarchy can then hold sub-modules of their own, e.g. the ALU inside the CPU.

SystemC modules can be used to represent individual blocks or a collection of blocks. A module's counterpart is the VHDL entity and Verilog module. A SystemC module is simply a class that extends the sc_module class provided by the SystemC library.

Listing 1 and Listing 2 below show two syntax styles for writing SystemC modules. For more information about SystemC modules see e.g. Section 5.2 in [1] and Sections 4.2 and 4.3 in [13].

```
class module_name : public sc_module
{
public:
    SC_CTOR(module_name);
};
```

Listing 1 Module definition using standard C++ syntax

```
SC_MODULE(module_name)
{
  SC_CTOR(module_name)
  {
    // process registration here
  }
};
```

Listing 2 Module definition using built-in SystemC macro

3.2 SystemC Processes

Modules can have member functions as any C++ class. Module member functions can be used as traditional C++ functions, with parameters and return values, and desired (or non-desired) side effects. Module member functions can also be used as SystemC processes.

A function intended for usage as a SystemC process should have no return type and take no arguments. It must also be registered with the SystemC kernel using an appropriate macro.

A process function can be registered as a SystemC thread, or as a SystemC method. Registration is done using the SC_THREAD macro for SystemC threads, or using the SC_METHOD macro for SystemC methods, as seen in Listing 3. The registration can be done within the module constructor using the SC_CTOR macro. Registration of a function as a thread or a method determines how it will be scheduled by the kernel and limits what the function can do.

```
SC_MODULE(module_name)
{
  SC_CTOR(module_name)
  {
    SC_THREAD(a_thread);
    SC_METHOD(a_method);
  }
  void a_thread();
  void a_method();
  void just_a_function();
};
```

Listing 3 Registration of SystemC threads and methods

A SystemC method has a sensitivity list and is invoked by the kernel each time there is a change in the list. In this way, methods are like the VHDL `process` and Verilog `always@`. Once invoked, methods will run to completion and cannot be interrupted as they do not support context switching. Executing code that either directly or indirectly attempts to suspend a method will result in the simulation hanging or crashing.

Listing 4 shows the implementation of a NAND2 module, simulating a two-input NAND gate, using a SystemC method. The module contains a single function for implementing the NAND operation which is registered as a method, sensitive to the inputs A and B.

```
SC_MODULE(nand2)
{
  sc_in<bool> A, B;
  sc_out<bool> F;

  SC_CTOR(nand2)
  {
    SC_METHOD(do_nand2_method);
    sensitive << A << B;
  }

  void do_nand2_method()
  {
    F.write( !(A.read() && B.read()) );
    // F = !(A && B); //also possible
  }
};
```

Listing 4 A NAND2 module implemented with a SystemC Method

A SystemC thread is also similar to the VHDL process and Verilog always@ as it can have a sensitivity list. However, threads can be suspended as they execute. A thread is suspended by calling the SystemC function wait. Moreover, if a thread returns during simulation it will not run again. When a thread runs it continues without interruption until it suspends itself by calling wait (or returns). This is referred to as cooperative scheduling, and it means that a thread cannot be pre-empted, in contrast to threads in an operating system, such as Linux, or threads in a real-time operating system (RTOS), where preemptive scheduling is often used.

Listing 5 shows a simple stimulus generator for the NAND2 module. The stimulus generator thread will write values to ports A and B, then stop the simulation at the next positive clock edge.

```
SC_MODULE(stim)
{
  sc_out<bool> A, B;
  sc_in<bool> Clk;

  SC_CTOR(stim)
  {
    SC_THREAD(stim_gen);
    sensitive << Clk.pos();
  }

  void stim_gen()
  {
    A.write(false);
    B.write(false);
    wait();
    sc_stop();
  }
};
```

Listing 5 A stimulus generator module implemented with a thread

The main considerations when choosing between threads and processes are performance and difficulty of implementation. Because methods cannot be suspended they do not include the context switching overhead. However, as designs get bigger, communication between methods is harder to implement than it is for threads. Most models use a combination of methods and threads, employing methods when simulation performance of threads is not sufficient.

For more information about SystemC processes see Sections 5.2.10 and 5.2.11 in [1] and Sections 4.4 and 4.5 in [13]

3.3 SystemC Events

SystemC uses the notion of events to enable inter-process coordination. Events have no duration or value, so processes must wait for an event before it occurs, otherwise the event will go unnoticed.

To cause an event, processes should call the `notify` member function of the `sc_event` class. The notification can either be immediate or timed, and it affects how processes waiting for that event are scheduled. This subject is covered more in depth in section 3.6.

Processes can either be sensitive to a predetermined set of events, in which case they will be triggered every time the events are notified. This is referred to as static sensitivity. Alternatively, and seen more often in threads than in methods, processes can dynamically become sensitive to events. This enables them to be sensitive to an event outside their static sensitivity list. It is also useful to note that when a thread waits for a timeout it is implicitly listening to a kernel event that will be notified when the timeout is reached.

Listing 6 shows how events are used to coordinate a handshake protocol between two threads The `notifier_thread` notifies `rec_event` and waits for a notification on `ack_event`, upon which it ends the simulation. Meanwhile, the `waiter_thread` waits for notificiations on `rec_event`, notifying `ack_event` when they happen. For more information about SystemC events see Section 5.10 in [1] and Section 6.5 in [13].

```
class WaiterNotifier : public sc_module {
public:
  sc_event rec_event;
  sc_event ack_event;

  SC_HAS_PROCESS(WaiterNotifier);
  WaiterNotifier(sc_module_name nm) :
    sc_module(nm),
    rec_event("req_event"),
    ack_event("ack_event")
  {
    SC_THREAD(notifier_thread);
    SC_THREAD(waiter_thread);
  }

  void notifier_thread() {
    // if picked first, allow the other thread to enter the waiting state
    wait(SC_ZERO_TIME);
    rec_event.notify();
    wait(ack_event);
    sc_stop();
  }

  void waiter_thread() {
    while (true) {
      wait(rec_event);
      ack_event.notify();
    }
  }
};
```

Listing 6 Thread coordination using events

3.4 SystemC Channels & Ports

SystemC modules communicate with each other using channels and ports. Channels and ports simplify communication between modules by hiding implementation details of the communication protocols, and only providing a set of operations for that port-channel combination.

The common point of the port-channel combination is the interfaces. These are abstract C++ classes that extend the `sc_interface` class. A channel implements interface classes so that ports for that channel can be used. In other words, channels implement communication interfaces while ports expose these interfaces to modules. A port is an instance of the `sc_port<some_in_if>` class, where `some_in_if` is the input interface of the channel. Figure 1 illustrates this relationship between ports and a channel.

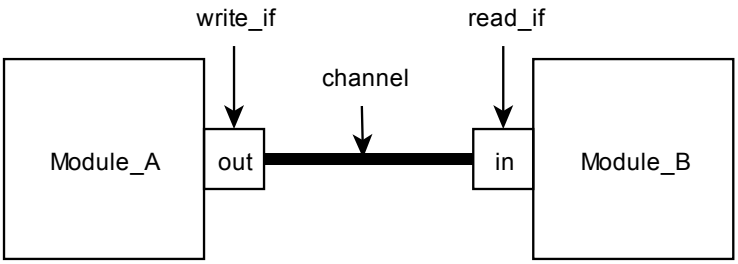


Figure 1 modules connected via ports and a channel

There are two types of channels: primitive and hierarchical. Primitive channels don't have a simulation process or hierarchy, making them fast. Hierarchical channels can have internal structure, i.e. a channel can have internal modules that implement communication protocols.

Listing 7 and Listing 8 show a simple write interface and its implementation as a FIFO primitive channel. For simplicity, the read interface and its implementation are left out in this example. Listing 9 shows how a module uses `write_if` in a port declaration. This enables the module to use the communication mechanisms provided by the write interface without knowing its implementation details. For more information about SystemC channels see Sections 5.2.23 and 5.15 in [1] and Sections 8 and 13 in [13].

```
class write_if : virtual public sc_interface
{
public:
  virtual void write(char) = 0;
  virtual void reset() = 0;
};
```

Listing 7 A write interface

```
class fifo : public sc_channel, public write_if, public read_if
{
public:
  fifo(sc_module_name name)
  :sc_channel(name),
  ,num_elements(0)
  ,first(0) {}

  void write(char c) {
    if (num_elements == max)
      wait(read_event);

    data[(first + num_elements) % max] = c;
    ++ num_elements;
    write_event.notify();
  }

  void reset() { num_elements = first = 0; }

private:
  enum e { max = 10 };
  char data[max];
  int num_elements, first;
  sc_event write_event, read_event;
};
```

Listing 8 A FIFO implementation for write_if

```
class producer : public sc_module
{
public:
  sc_port<write_if> out;

  SC_HAS_PROCESS(producer);
  producer(sc_module_name name) : sc_module(name)
  {
    SC_THREAD(main);
  }

  void main()
  {
    const char *str = "Hello World ";
    while (*str)
      out->write(*str++);
  }
};
```

Listing 9 A producer module using a write_if port

3.5 TLM 2.0

TLM stands for transactional level modeling, which replaces events/pin wiggles with function calls to accelerate simulation. The main characteristic of TLM is that it provides a functional representation of communication, with a focus on memory-mapped buses. TLM can be used to model systems in a bit-accurate and register-accurate manner, with clock cycles and detailed bus protocol behaviors being abstracted away. TLM models are fast enough for running operating systems and other software, and should generally be available before RTL models, but accurate enough to be used post-RTL, for example when running software regression tests on a TLM-based virtual platform.

TLM 2.0 is an implementation of TLM, based on SystemC. TLM 2.0 provides a few interfaces that define APIs for blocking and non-blocking interfaces. This allows for simplified development and integration of high-level system components.

For more information about TLM 2.0 see [2] and Chapter 16 in [13].

3.6 The SystemC Simulation Method

The previous sections introduced the main SystemC constructs for building and interconnecting structures. These constructs are a part of the public API of a SystemC implementation. Along the public API, a SystemC implementation must also provide a private kernel. Together these parts are used to run a SystemC application.

Executing a SystemC application consists of running an *elaboration* phase followed by a *simulation* phase. During elaboration, the *module hierarchy* is created. This is done by executing the public API code used by the application with assistance from the kernel. The following simulation phase involves executing the *scheduler*, the part of the kernel responsible for executing SystemC processes present in the application.

The standard defines the following steps for running a SystemC application:

- Elaboration—Construction of the module hierarchy
- Elaboration—Callbacks to function before_end_of_elaboration
- Elaboration—Callbacks to function end_of_elaboration
- Simulation—Callbacks to function start_of_simulation
- Simulation—Initialization phase
- Simulation—Evaluation, update, delta notification, and timed notification phases (repeated)
- Simulation—Callbacks to function end_of_simulation
- Simulation—Destruction of the module hierarchy

3.6.1 Elaboration phase

The elaboration phase is where the module hierarchy is constructed and stored as data structures that support the simulation semantics defined by the standard. The phase lasts during the call to `sc_main` and until the first call to `sc_start`. The `sc_main` function is the only entry point into a SystemC application and is called by the kernel. The first call to the `sc_start` function starts the scheduler, thus marking the end of the elaboration phase and start of the simulation phase, covered in the next section.

The first stage of the elaboration phase instantiates all occurrences of the `sc_module, sc_port, sc_export, and sc_prim_channel` classes. These instantiations may only occur during the elaboration phase. Further, instances of the classes `sc_module` and `sc_prim_channel` may only be created within a module or within the `sc_main` function.

During the second stage *unspawned* processes are created by invoking one of the process macros. It is possible to bypass the macros although the procedure is quite convoluted and thus error prone.

The third elaboration stage is port and export binding. Here port instances are bound to channel instances, or other port or export instances. Export instances are bound to either channel or other export instances. Port and export binding may only occur during the elaboration phase.

The final stage in the elaboration phase is setting the time resolution by calling the `sc_set_time_resolution` function.

3.6.2 Simulation phase

The simulation phase starts with the first call to `sc_start`, which starts the scheduler. The scheduler is event driven, i.e. process execution depends on events. Events are represented by objects of the class `sc_event`. Events can be notified, by calls to function `notify` of the class `sc_event`.

The scheduling algorithm relies on four sets:

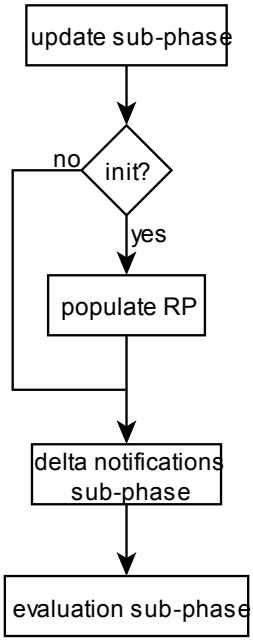
- The set of runnable processes (RP-set)
- The set of update requests (UP-set)
- The set of delta notifications and time-outs (D-set)
- The set of timed notifications and time-outs (T-set)

RP-set is first populated during the elaboration phase and contains at most one instance of each process. UP-set is populated through calls to member function `request_update` or `async_request_update` of class `sc_prim_channel`. D-set and T-set are populated through calls to member function `notify` of class `sc_event` with zero-valued and non-zero-valued time arguments, respectively. Time-outs result from

calls to `wait` and `next_trigger` member functions of `sc_module`. The set to which the time-out is added depends on whether it resulted from a call with zero-valued or non-zero-valued time argument. The kernel updates these sets through five sub-phases, described as follows.

Initialization
The initialization sub-phase runs only once per simulation. It starts by running an update phase once without continuing to the delta notification phase. After the initial update phase, all process instances in the module hierarchy are added to RP-set, except for those that were indicated not to be initialized. Finally, the delta notification phase runs followed by the evaluation phase. The figure below illustrates the initialization sub-phase.

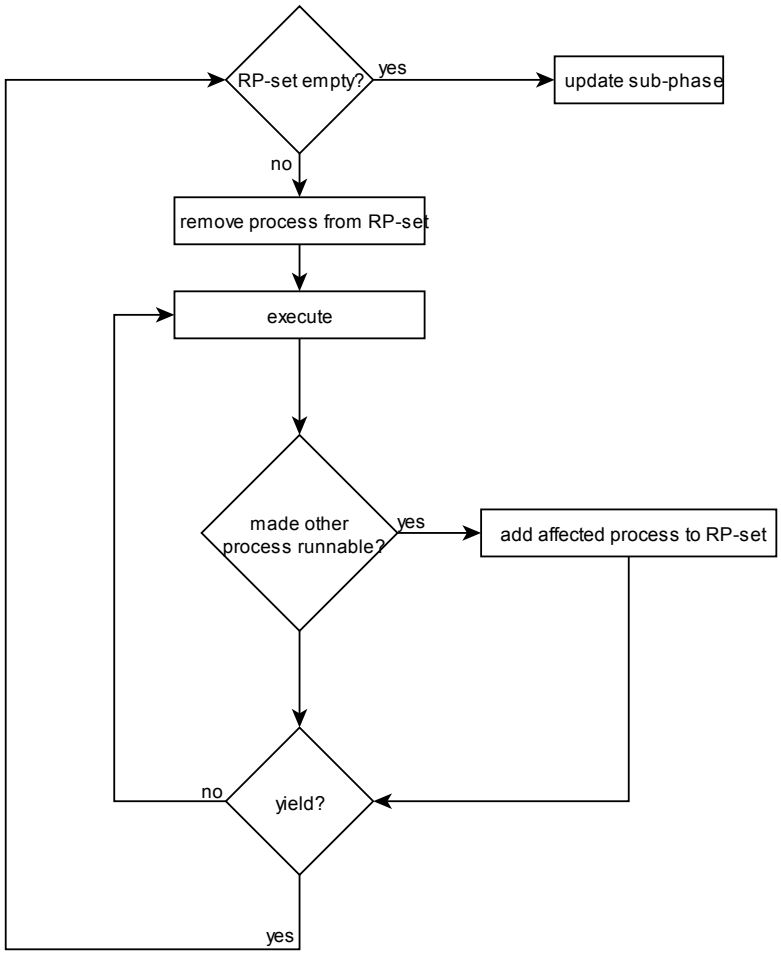
It is necessary to run the initial update phase and a delta notification phase because there may have been update requests during the elaboration phase. This can be the case when it is desired to set initial values for primitive channels.



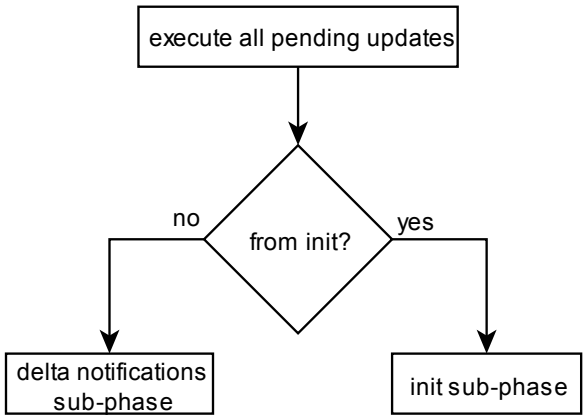
Evaluation
This phase is responsible for executing SystemC processes. Each process instance in RP-set, in no particular order, is allowed to run without interruption until the process either suspends itself or returns. This kind of scheduling is known as co-operative multitasking. In this case, the designer of the SystemC application is responsible for making sure that each process eventually yields control back to the kernel.

While a process runs it can cause other processes to become runnable by spawning one dynamically, issuing an immediate notification to which there are currently sensitive processes, calling `request_update` or `request_update` on a primitive channel, or directly calling a process control function of another process. In all but the request_update cases, the affected processes are moved to RP-set and will be run in the current evaluation phase.

This sub-phase continues until there are no more processes in RP-set, at which point the simulation continues to the *update* sub-phase. The figure below illustrates the evaluation sub-phase.



Update
In this sub-phase, all pending calls to the update function of primitive channels are executed at most once. If the phase was not entered from the initialization phase, the simulation continues to the delta notification phase, otherwise it returns to the initialization sub-phase. The image below illustrates this sub-phase.



Delta notification
If there were calls to `notify` or `wait` during the last evaluation or update sub-phases, all processes sensitive to the events or time-outs are added to the RP-set and the notifications and timeouts are removed from the D-set. If at the end of this sub-phase RP-set is empty, the simulation continues to the timed notification sub-phase. Otherwise, the evaluation sub-phase is repeated.

Timed notification
This sub-phase is similar to the delta-notifications phase in that it finds processes sensitive to notifications or time-outs and moves them to RP-set. The difference is that simulation time is advanced to the earliest timed notification. If processes were added to RP-set, the simulation continues with the evaluation phase, otherwise the sub-phase is repeated, advancing time again. If there are no timed notifications or timeouts the simulation ends.

3.7 Modelling of Timing in SystemC & TLM

SystemC and TLM allow modelling at different levels of time granularity. This allows models to be created with the level of detail required at the current design stage or by their use case. The TLM-2.0 standard defines the following coding styles appropriate for each design stage.

3.7.1 Loosely-timed Coding Style

The loosely-timed style uses the blocking transport interface which has two timing points marking only the start and end of a transaction. Its most important feature is that it allows temporal decoupling, where SystemC processes can run ahead of simulated time by a certain amount of time, referred to as the global quantum in the standard [2].

By tuning the quantum, it is possible to trade off simulation speed (higher quantum means less thread synchronization and thus lower kernel overhead) against simulation accuracy (lower quantum means more thread synchronization and thus higher kernel overhead). In the ideal case, threads should call wait only when the global quantum is reached. This coding style supports the modelling of timers and interrupts, and as such is suitable for developing virtual platforms on which to run operating systems and applications.

3.7.2 Approximately-timed Coding Style

The approximately-timed style uses the non-blocking transport interface and each transaction can have multiple timing points. The TLM-2.0 base protocol has four points, marking the beginning and end of the request and response.

This style cannot use temporal decoupling and is best used for hardware architecture analysis.

3.7.3 Cycle-accurate Coding Style

Cycle accurate models provide an accurate description of the state of the model for each clock cycle. As such they are useful for performance verification. They could also be useful as reference models for RTL designs, e.g. for processors.

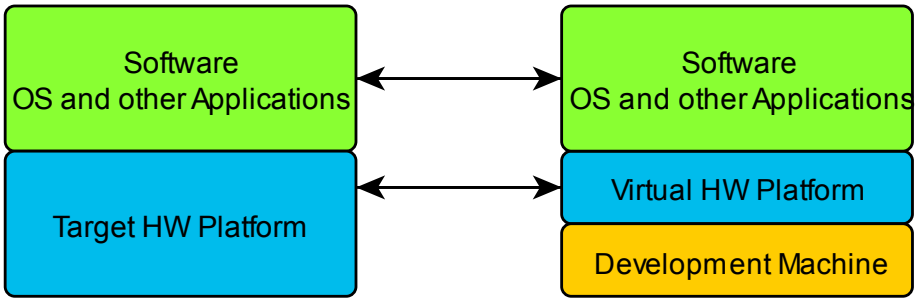
Cycle-accurate modeling can be done in SystemC, however, it is not directly supported within TLM-2.0.

3.7.4 RTL Models

RTL modelling provides the most accurate results from the described modelling styles. It achieves this by providing both the state of the model at each clock cycle as well as the state of individual signals involved in computation and communication. However, this accuracy comes with a significant performance overhead, making RTL models unsuitable for software verification. Although SystemC can be used also for RTL modelling, VHDL and Verilog are usually used for this purpose given the availability of synthesis tools.

4. Virtual Platforms

A virtual platform is a simulation of certain hardware. For this purpose, models of the target hardware are needed. When developing the virtual platform, the goal is to make it behaviorally identical to the target hardware. Represented in the figure below is this requirement, as well as the fact that the same software binaries should run on both the platform and the actual hardware.



A virtual platform can be implemented using SystemC and TLM. to allow bit-accurate simulations of target platforms, which is important if the virtual platform is to be used for hardware and software co-development with any guarantee of reliability. While SystemC can be used to model simple hardware at the RTL level, it is often used for ESL design, allowing the modelling and fast simulation of full SoCs. These models can then be used to run software intended for the target platform.

4.1 Ericsson System Virtualization Platform(SVP)

Ericsson uses SystemC and TLM to model hardware used in radio base stations. SVP is a virtual platform, used at Ericsson for testing software developed for the Ericsson Many Core Architecture (EMCA).

EMCA is a real time multi-processor system, on which an operating system that manages task scheduling and resource allocation is running. The same operating system can run on EMCA platforms simulated with SVP. In turn, baseband processing software and other software systems can run on SVP, using the operating system, executing on the simulated processors within SVP.

Most of the processor models in SVP are instruction-accurate with some models having a cycle-accurate variant too. The instruction-accurate models are used for most software development tasks as they are sufficiently accurate.

SVP's main goal is to facilitate software development and integration, without being dependent on actual hardware. Therefore, the loosely-timed coding style is used to develop the platform, as it provides good simulation performance and maintains the same behavior as the actual hardware.

4.2 Tracing from a Virtual Platform

Dynamic analysis tools are used to inspect programs during run-time. As such, they can perform a variety of functions like detecting access to restricted memory, memory leaks, cache inspection, call graph generation etc.

This thesis draws inspiration from these tools to provide an instrumented SystemC kernel. This modified kernel profiles how SystemC threads and methods execute within a given simulation. This data can then be used to draw conclusions about the system model.

4.2.1 Non-SystemC-aware Methods

Valgrind [12] offers a set of popular dynamic analysis tools. Originally, it was a memory debugging tool for Linux, but has since its creation evolved into a general framework for building dynamic analysis tools. It comes with a plethora of tools for different kinds of profiling, most notably Memcheck for memory profiling, Cachegrind for cache profiling, and Callgrind for generating call graphs.

While very versatile, Valgrind is not very useful for this thesis. Since SystemC models are themselves C++ programs, profiling them with Valgrind is possible but the result traces would be cluttered with operations not directly related to the simulated models.

4.2.2 SystemC-aware Tracing

As mentioned previously, there are a variety of dynamic analysis tools, most of which focus on a specific piece of software or specific languages. It is possible to use Valgrind but a more efficient approach is to directly modify the SystemC kernel. This approach gives direct access to the parts of interest within SystemC, such as threads and methods, and generating traces can be as simple as running a simulation with different trace parameters. We refer to such profiling as SystemC-aware, as it explicitly takes into account the properties of the SystemC implementation, for example when tracing thread executions or event notifications.

A consideration before implementing SystemC-aware tracing is considering whether there are already tools that accomplish the desired end-goal. The tool set offered by Cadence [4] for example, can generate wave-form and TLM transaction views, and it also supports software and hardware model co-debugging. These features are useful when developing virtual platforms, however, they are too specific for this work, where

the interest lies in analyzing general platform usage. Considering this, the approach chosen for this thesis was to modify the freely available SystemC implementation from Accellera [5].

5. A SystemC Kernel Instrumented for Tracing

As stated by the SystemC standard, the kernel can be extended to provide additional functionality not included in the standard scope. The result of this thesis accomplished exactly this, resulting in a modified SystemC kernel that supports detailed and selective simulation tracing.

We generate data for certain events in the kernel. We refer to these as *simulation events* (not to be confused with the events used in SystemC, i.e. the variables of the sc_event class). To trace a simulation event the SystemC code implementing that event must be determined first. This is mainly done by inspecting the Accellera reference kernel code directly in conjunction with the IEEE language reference manual.

5.1 Process Creation, Suspension, and Resumption

Some generally useful simulation events to trace are process creation, suspension, and resumption. The information from these events forms the baseline for implementing observability into the SystemC kernel. One function was introduced to trace general process activity with the signature seen below. The first argument to the function is a process pointer so data about the process can be accessed, the second argument indicates the event type so that the trace is formed correctly.

```
static void trace_process_event(sc_process_b*, const EVENT_TYPE)
```

Process creation tracing is useful since processes in general correspond to hardware functionality. Therefore, missing processes could indicate a faulty model (the model has not been implemented correctly, or not initialized correctly). As mentioned in section 3.6.1, processes are created in the second stage of the elaboration phase using predefined macros. These macros perform the necessary setup before calling the process creation functions. The signatures for these functions are shown below and their instrumentation is shown in Listing 10 and Listing 11

```
sc_simcontext::create_thread_process(const char*, bool, SC_ENTRY_FUNC,
    sc_process_host*, const sc_spawn_options*)

sc_simcontext::create_method_process(const char*, bool, SC_ENTRY_FUNC,
    sc_process_host*, const sc_spawn_options*)
```

```
sc_process_handle sc_simcontext::create_thread_process(
  const char* name_p, bool free_host,
  SC_ENTRY_FUNC method_p,
  sc_process_host* host_p, const sc_spawn_options* opt_p )
{
  sc_thread_handle handle =
      new sc_thread_process(name_p, free_host, method_p, host_p, opt_p);
  tracing::TraceLib::trace_process_event(handle,
                        tracing::TraceLib::CREATE_T);
  ...
}
```

Listing 10 Thread creation tracing

```
sc_process_handle sc_simcontext::create_method_process(
    const char* name_p, bool free_host,
    SC_ENTRY_FUNC method_p,
    sc_process_host* host_p, const sc_spawn_options* opt_p )
{
  sc_method_handle handle =
      new sc_method_process(name_p, free_host, method_p, host_p, opt_p);
  tracing::TraceLib::trace_process_event(handle,
                        tracing::TraceLib::CREATE_M);
  ...
}
```

Listing 11 Method creation tracing

Once processes are created their execution is handled by the simulation phase, as described in section 3.6.2. Process suspension and resumption provide a coarse-grained view of how processes execute. This information can reveal processes that never executed (created but never resumed), or were never resumed once suspended (suspended but never resumed). Both scenarios might indicate faulty models or faults in software. Alternatively, the information can be used to acquire an overview of module utilization for a given simulation configuration.

Since methods and threads have different behavior, their suspension and resumption is handled differently. In the case of threads, whose execution can be interrupted, there is an explicit function which is called when the thread suspends. In the case of methods, that run uninterrupted, the simulation function directly handles their suspension. These functions have the signatures seen below and their instrumentations are shown in Listing 12 and Listing 13.

```
sc_thread_process::suspend_me()

sc_simcontext::crunch( bool once )
```

```
inline void sc_thread_process::suspend_me()
{
  ...
  if( m_cor_p != cor_p )
  {
    tracing::TraceLib::trace_process_event(this,
                            tracing::TraceLib::SUSPEND_T);
    DEBUG_MSG( DEBUG_NAME , this, "suspending thread");
    simc_p->cor_pkg()->yield( cor_p );
      tracing::TraceLib::trace_process_event(this,
                               tracing::TraceLib::RESUME_T);
    DEBUG_MSG( DEBUG_NAME , this, "resuming thread");
  }
  ...
}
```

Listing 12 Thread suspension and resumption tracing

```
inline void sc_simcontext::crunch( bool once )
{
  ...
  sc_method_handle method_h = pop_runnable_method();
  while( method_h != 0 ) {
    tracing::TraceLib::trace_process_event(method_h,
                            tracing::TraceLib::RESUME_M);
    empty_eval_phase = false;
    if ( !method_h->run_process() )
    {
      goto out;
    }
  tracing::TraceLib::trace_process_event(method_h,
                          tracing::TraceLib::SUSPEND_M);
  ...
}
```

Listing 13 Method suspension and resumption tracing

The output format for general process event traces is shown in the table below. From left to write, the columns indicate the exact event that was traced (i.e. creation, suspension, or resumption), the simulation time at which the event was traced, the name of the involved process, and the wall time in seconds and nanoseconds. In the second row the trace example indicates that at simulation time 0 s, i.e. simulation start, a thread called th0 was created.

event	sc_time_stamp	name	tv_sec	tv_nsec
create_thread	0 s	th0	1	1000000000

5.2 Calls to wait

As described in section 3.2, threads can suspend themselves by calling the wait function. Each call to wait will immediately suspend a thread, making wait tracing equivalent to thread suspension tracing. However, there are different signatures of the wait function, each of which will suspend the calling thread in a different manner. This additional information shows the exact reason for a thread suspending, making it useful to trace calls to wait.

Below are all the signatures for the wait function. Generally, they can be divided in two groups: untimed and timed. The first three signatures are for untimed suspensions. Threads that called wait this way will stay suspended until the event or event list they are waiting for is notified. Note that there are two types of event lists. A thread waiting for an `event_or_list` will trigger when any event in the list is notified, while threads waiting for an `event_and_list` will only trigger when all the events in the list are notified. The following four signatures are for timed suspensions. Threads suspended in this way will wait for event notifications only for the specified amount of time before resuming. There is also the special case of calling the single time argument wait function with a zero-time value, which suspends threads for a delta notification phase. Finally, there is the last signature, which is used to suspend threads until changes in their sensitivity lists occur.

In the case of untimed waits, the calling thread will suspend itself until one or more events are notified. In the case of timed waits, the suspended thread will wait for event notifications only for the specified amount of time before resuming. There is also the case of the empty wait call, which suspends the thread until there are changes in its sensitivity list. Another special case is calling wait with a zero-time value for t, this suspends the thread for the current delta cycle so that other threads get a chance to run.

```
wait(const sc_event&)
wait(const sc_event_or_list&)
wait(const sc_event_and_list&)
wait(const sc_time&)
wait(const sc_time&, const sc_event&)
wait(const sc_time&, const sc_event_or_list&)
wait(const sc_time&, const sc_event_and_list&)
wait()
```

The signatures for the wait tracing functions are seen below. Each function takes a pointer to the thread that called wait along with either the event or event list that the thread is waiting for. In the case of timed suspensions, the timeout is also passed. Listing 14 shows how single event wait tracing is integrated into the kernel, the other functions are integrated similarly in their respective wait functions.

```
trace_wait(sc_process_b*, const sc_event&)
trace_wait(sc_process_b*, const sc_event_list&)
trace_wait(sc_process_b*, const sc_time&)
```

```
trace_wait(sc_process_b*, const sc_time&, const sc_event&)
trace_wait(sc_process_b*, const sc_time&, const sc_event_list&)
trace_wait(sc_process_b*)
```

```
inline void sc_thread_process::wait( const sc_event& e )
{
  if( m_unwinding )
    SC_REPORT_ERROR( SC_ID_WAIT_DURING_UNWINDING_, name() );
  tracing::TraceLib::trace_wait(this, e);
}
```

Listing 14 Single event wait tracing

The output format for the wait tracing is presented in the table below. The columns from left to right indicate the type of simulation event, the name of the event for which the thread waited, the simulation time at which the trace was taken, the name of the waiting thread, and the wall time in seconds and nanoseconds. In this example wait was called for event `e0` at simulation time 0s by thread `th0`.

event	for	sc_time_stamp	name	tv_sec	tv_nsec
wait	e0	0 s	th0	1	1000000000

5.3 Calls to notify

Calls to notify are used to schedule processes, that are sensitive to the notified event, for execution. Notifications can either be immediate or timed. In the case of an immediate notification, sensitive processes will be scheduled for execution in the same delta cycle. Timed notifications will schedule processes after simulation time has advanced to the given time. The signatures for these functions are shown below.

```
sc_event::notify()
sc_event::notify(const sc_time&)
```

When a process waits for an event, it registers itself as sensitive to that event. This means that it will be scheduled for execution, also referred to as triggered, when the event is notified. This mechanism can be used to synchronize processes, by letting a process wait for an event until a specified condition is fulfilled, which is then signaled by a notification on the event. The process is then scheduled for execution. If the process does not call wait before `notify` is called, it will not be sensitive to the event when the notify call is done, and it will therefore not be triggered.

When tracing a notification, it is possible to obtain the list of triggered processes. In the case of an empty trigger list, this indicates that no processes were sensitive to the event, which might not be the intended behavior.
The `sc_event::notify` functions are part of the SystemC API, and they are typically used by developers of SystemC programs. It can be seen, in the SystemC kernel source code, that the functions are wrappers, containing calls to other, SystemC internal, functions.

In the case of immediate notifications, tracing can be done within the `sc_event::notify` functions, since all useful information, such as the identity of the notifying process, the notification time, and what processes were triggered by the notification, is available in the simulation context.

In the case of timed notifications, the function only stores the fact that a given event is to be notified once simulation time has advanced to the desired point in a subsequent timed notification sub-phase. Because of this, the simulation context does not, at the moment of calling notify, contain the processes that are triggered by the notification. Therefore, tracing of timed notifications is done inside the SystemC internal `sc_event::trigger` function, which actually schedules the processes that are sensitive to the event. However, this approach loses the information on the identity of the notifying process, and when the notification was issued. This information can be kept, however, by adding two more data members to the `sc_event` class.

To keep the implementation consistent, immediate notifications are also traced within the `sc_event::trigger` function. The signature for the tracing function is seen below, and its integration into the SystemC kernel is shown in Listing 15. The argument is a pointer to the event object for accessing the trace data mentioned above.

```
trace_notify(sc_event*)
```

```
void sc_event::trigger()
{
  int       last_i; // index of last element in vector now accessing.
  int       size;   // size of vector now accessing.
  tracing::TraceLib::trace_notify(this);
  ...
}
```

Listing 15 Notify tracing

The output format for notify traces is presented in the table below. The columns from left to right indicate the type of the simulation event, the name of the event that was notified, the name of threads that were triggered by the notification, the time at which the event was notified, the simulation time at which the trace was taken, the name of the notifying process, the wall time of the simulation in seconds and nanoseconds. In this example event `e0` was notified by `th0` at simulation time `0s` and the trace was taken at the same time, indicating an immediate notification. The last two columns simply show the wall time of the simulation.

event	on	triggered	call_t	sc_time_stamp	Name	tv_sec	tv_nsec
notify	e0	th0	0 s	0 s	th1	1	1000000000

By tracing when events suspend by calling wait, and resume because of notifications on events, it is possible to construct cause and effect relationships between simulation events and process executions. In other words, the information exposes the causes for

simulation behavior. The issue here is that models can be quite large, and therefore generate a large amount of traces. This renders visual inspection of traces unfeasible and a post processing of trace data is required.

5.4 Storage of Traces

All traces are written in plain text over standard output and saved to files. While not the most efficient approach, its simplicity allows focusing on instrumentation implementation details. After a simulation run, its output log is stripped of non-trace data, and the resulting file is processed using Python scripts which in turn generate four files. Three of the files are in binary format and are used to store all the traces for thread suspension, all thread-specific traces, and a sorted aggregate of all trace data. The aggregated data is also stored in a text file for visual inspection.

6. Trace Use Cases

Trace files by themselves do not provide much insight into program execution. To make any sort of useful conclusions, traces must first undergo some form of data processing. This processing could be something as simple as counting occurrences of a specific instruction/event, all the way to complex statistical analysis ranging over multiple trace files.

The most accessible thing to extract from the trace data is general statistics such as which processes ran during simulation and how often, how many times events were notified and what processes they triggered, and threads that called wait and on what.

This information along with other data is organized by the processing tools in a table as seen in Table 1.

Rank	Thread Name	Runtime [ms[	N_sus	A@0	d_ns avg	Min	Max	Std
1	thread_0	4278.341	19437		47.75	0	447.14	97.45
2	thread_1	2431.791	17749		52.05	0	78430	596.85
3	thread_2	321.682	19259		47.99	0	121169.4	1444.72
4	thread_3	273.624	17167		53.82	0	122216.6	1243.98
5	thread_4	5.596	107		8735.93	0	472305.7	57626.63
6	thread_5	5.442	108		8636.15	4	777069.7	75160.88
7	thread_6	2.857	103		9056.9	4	777273	76973.87
8	thread_7	1.567	65		13158.83	4	837542.9	103864
9	thread_8	1.499	66		12887.18	0	837542.9	103082
10	thread_9	1.481	66		12956.39	0	842041.4	103635.6

Table 1 Runtime statistics for threads

6.1 SystemC Performance Analysis

The data presented in Table 1 can be used to obtain a view of simulation performance. The column “Runtime” shows the wall time for which each thread ran, in the column “d_ns avg” is the average simulated time between executions of a particular thread. Columns “Min” and “Max” show the minimum and maximum simulated time between executions, and the “Std” column shows the standard deviation for the simulated time between executions. The usefulness of this data is presented in section 7.1, where it is used to determine the effective quantum of a simulation.

6.2 SystemC Coverage

Also from Table 1, the number of times threads are suspended can be seen from column “N_sus”, and whether threads only ran at the beginning of the simulation from column “A@0”. The information in the latter column can be used to identify the threads that are not used at all during a simulation, while the information in the former can be used to create a heatmap of thread usage by particular tests. This use case is demonstrated in section 7.3.

Another table is constructed from event notification tracing, as seen in Table 2. From this data notify-trigger causality graphs can be generated, i.e. graphs that show for each simulation time which threads ran and what events caused them. In general, this can be used to confirm that processes were executed in the intended order. Alternatively, the metric can be used on different SVP versions to find mismatches in process ordering. A mismatch is likely to cause sensitive software to fail.

Notifier	Sim. Time	Delay	Wall Time[s]	Wall Time[ns]	Triggered
thread_0	383679142ps	26	266131	95666418	thread_1
	831068285ps	26	266132	4.42E+08	thread_1
	831265428ps	26	266132	4.47E+08	thread_1
	831588285ps	26	266132	4.49E+08	thread_1
	831945428ps	26	266132	4.5E+08	thread_1
	832346857ps	26	266132	4.54E+08	thread_1
	832664ns	26	266132	4.56E+08	thread_1
	833021142ps	26	266132	4.57E+08	thread_1
	833378285ps	26	266132	4.59E+08	thread_1
	924573142ps	20	266133	5.79E+08	thread_1

Table 2 Event notification statistic

6.3 Metrics targeting software sensitivity

We also collect data that allows inferring software sensitivity. This is done by running two simulations using different SVP versions. The trace logs can be compared with respect to simulation time and the processes that were resumed at each time. For this, we aggregate this data into a file that can be used with the UNIX diff tool. From here, it is possible to find the number of differences in timing, i.e. determine the number of times processes were scheduled at different simulation times. This metric is however, obscure, as the change in timing occurs when updating the model or changing the quantum. Therefore, looking at the number of differences alone is not very informative. It might be worth gathering this data for different SVP versions running the test suites and perform statistical analysis on the results to determine the acceptable number of differences when it comes to timing.

Another approach to using the "diff friendly" files is to look at the changes in process scheduling, i.e. the occurrences of tasks being scheduled in different groups.

Ideally, a tool that is capable of looking from both perspective (timing/scheduling) to provide a result would be developed.

The end goal of these metrics is to enable investigations into software sensitivity. The goal is to find metrics, and a set of test cases, so that given a change in SVP, it shall be

possible to evaluate if this change results in high or low software sensitivity, i.e. if this change is likely to break software tests that are sensitive to changes in SVP.

6.4 Processing and Visualization of Traces

A convenient way to relay coverage and performance is through data visualization. The system now supports generation of 4 kinds of plots.

- *sc_activities* - shows the activity of threads during a simulation
- *sc_activities_bundled* - a condensed view based on a regex indicating which processes that are of interest
- *sc_time_ns_diffs* - shows when threads called wait
- *causality_chain* - shows the notify-trigger graph at a given simulation time. If the time instant is not found in the trace the value closest to the provided one is taken.

The scripts allow the use of regular expressions as filters to what should be plotted. This is useful for debugging as it allows isolation of features, such as a particular thread.

The sc_activites plots may reveal threads that are overly active. This could prove useful for identifying simulation bottlenecks, potentially improving simulation performance.

7. Case Studies

7.1 Outliers in effective quantum

This case study illustrates the use of general SystemC coverage to identify outliers in in the simulation quantum. The purpose of the case study is to investigate how often the quantum was utilized, i.e. how often the SystemC threads executed without suspending before the quantum boundary. The resulting average time of execution before synchronization, is referred to as *effective quantum.*

As a starting point, a software test is run on SVP. This produces a trace log from which data on when threads suspended and resumed is extracted. The results are then plotted against the global quantum as seen in Figure X below. The red dots indicate how much of the global quantum was used by threads before suspending. In this example, the virtual platform is allowed to run for 357ns ahead of simulation time. Therefore, dots that fall under or above this time are suspicious. At first, only regions which are significantly far from the quantum are investigated. In this way, we focus on detecting outliers in the effective quantum. Of course, it is up to the developer to decide what is significant.

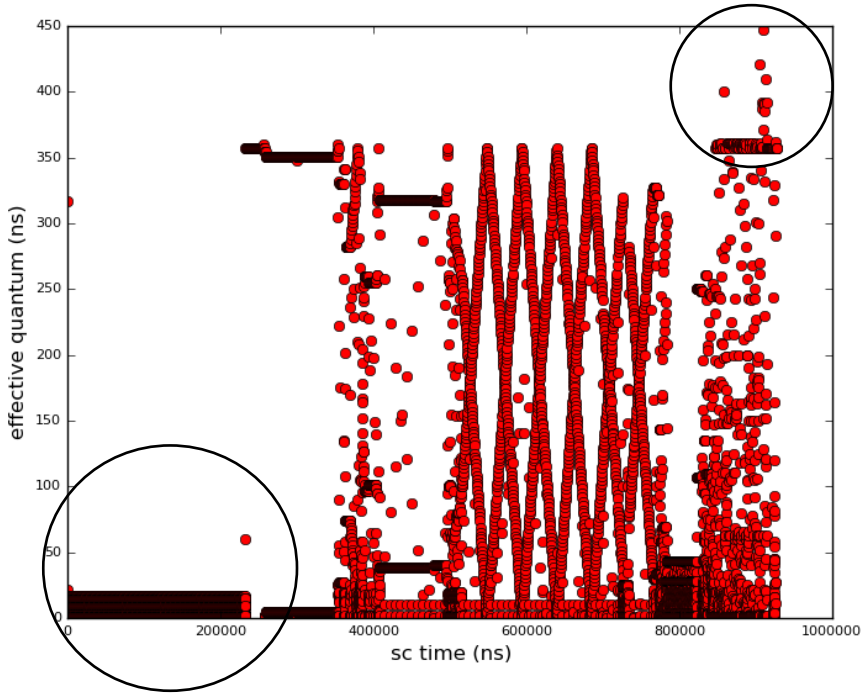


Figure 2 Effective quantum plot

Outliers may be indicative of bottlenecks or anomalies in a simulation. In Figure X, two regions of significant outliers can be seen, marked with circles. The first one is between the simulation start at 0ns and 200000ns. Most threads that were active in this period yielded control to the kernel roughly every 25ns. This over-synchronization slows down the simulation and is worth investigating. Thus, the test program is run again to generate a new log. This time the tracing library is set to only trace calls to wait and only in the period of interest. Furthermore, a backtrace, showing the C++ call chain leading up to the call to wait, is generated for each wait call. The new log is processed, a thread that suspended frequently is picked, and the backtrace in which it occurred is investigated. Doing this revealed that the frequent wait calls were due to semaphore initializations in one of the hardware models in SVP.

The second point of interest is roughly between 800000ns and 1000000ns. Here it seems that some threads are suspended after the specified quantum. This under-synchronization shows that the simulation has run ahead for longer simulated time than expected (before calling wait), and could indicate an unintended behavior. The cause for this behavior could be identified using the same approach as described in the previous paragraph, using backtraces for localization of the points in the source code where the calls to wait were eventually done.

7.2 Notify-Trigger Causality Graphs

Causality graph generation is a tool that can be used to detect flaws in the order of simulation execution. These flaws appear for example when a virtual platform is extended with new processes which are supposed to be sensitive to certain events. In this case, if the order of event notifications is not well understood or known, the newly added processes could miss the notifications entirely.

The code listing below presents a simple example of this case, involving only two threads. The `notifier_thread` function notifies the `missed_e` event. However, if this thread runs before the `waiter_thread`, the notification will go unnoticed. Processing the trace log for running this application produces the graph seen in Figure 3. The figure shows that while there was a notification on `missed_e` it did not trigger any process. To fix this issue, a zero-time wait is added to the notifier (commented in the listing below). Processing the log of the fixed application produce the graph in Figure 4. It can be seen here that notifying `missed_e` actually triggered the `waiter_thread`. The other event seen in the figure is notified by the scheduler from the delta notifications simulation subphase to indicate the notifier_thread functions that a delta cycle has passed.

This tool becomes more useful as the simulated models get larger. Figure 5 shows the causality graph obtained from the trace of a test run on SVP. In this simulation, multiple event notifications did not trigger any processes, as can be seen from the distinct arrows pointing into an empty box. The figure only shows a small part of the entire causality graph. This is done for illustration purposes. It remains a part of future investigations to

find out how larger amounts of data, obtained when creating larger causality graphs, shall be analyzed and visualized. In Figure 5, event and process names have been changed to obfuscate confidential information.

```
#include "systemc.h"

class WaiterNotifier : public sc_module {
public:
  sc_event missed_e;

  SC_HAS_PROCESS(WaiterNotifier);
  WaiterNotifier(sc_module_name nm)
    : sc_module(nm)
    , missed_e("missed_event")
  {
    SC_THREAD(notifier_thread);
    SC_THREAD(waiter_thread);
  }

  void notifier_thread() {
    // allow waiter_thread to wait
    // wait(SC_ZERO_TIME);
    missed_e.notify();
  }

  void waiter_thread() {
    wait(missed_e);
  }
};

int sc_main(int argc, char** argv) {
  WaiterNotifier dut("WaiterNotifier0");
  sc_start();
  return 0;
}
```

Listing 16 Missed notification example

Figure 3 Causality graph of an event that didn't trigger any processes

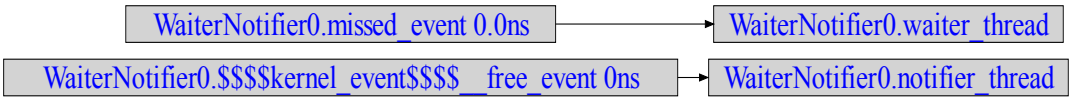


Figure 4 Causality graph showing intended execution

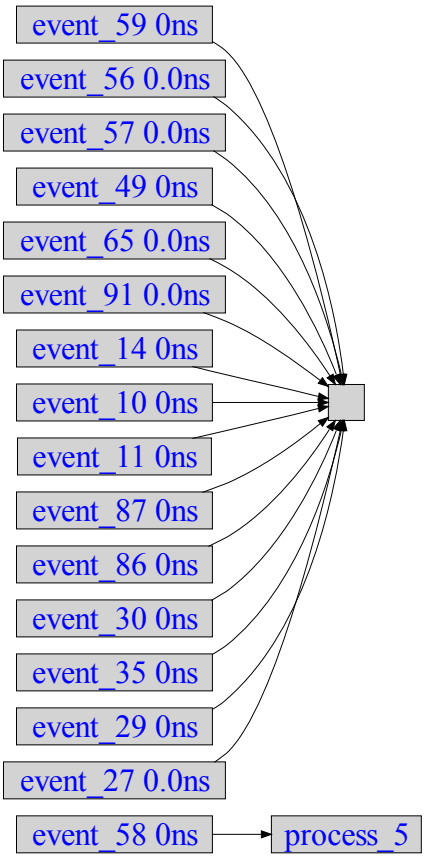


Figure 5 Multiple missed event notifications

7.3 Model Coverage

Another tool for using simulation traces is to graphically illustrate model usage by a particular test, or set of tests. As an example, Figure 6 shows a heat map for the 10 most active processes during 15 different tests that were run on SVP. The data for the figure is obtained by processing the logs of multiple tests and extracting the number of suspensions for each process. The numbers are then scaled logarithmically to account for large differences in process usage between tests, and normalized to the interval [0,1]. In Figure 6, the cooler the color of a box, the less the corresponding process was activated by a test. For example, it can be seen that test_7 frequently used process_9 but rarely used process 8. Further, it can be seen that process_8 is rarely used by any of the tests, as such this might be an incentive to write a test that makes more use of this process.

The overview, as illustrated in Figure 6, can be useful to identify how different tests execute different parts of the simulator. This information gives a direct measure of code coverage with a SystemC perspective, where it can be determined which threads that were executed, for each test case. A secondary usage of the information could be as a base for developing new test cases, with the goal of executing threads that have a poor coverage. It could also be used when determining which tests to run, for a given change of the platform. As an example, if code changes have been done in a certain set of threads, it could suffice (at least as a first test scenario) to run only the test cases that execute the threads where changes have been done. If these tests pass, the test set can be extended, so that also secondary effects of the code changes are tested. As a limiting case, if the coverage data shows that a certain thread is not executed for any of the test cases, this indicates dead code. A decision should then be taken, either to remove the code, or to add test cases that execute it.

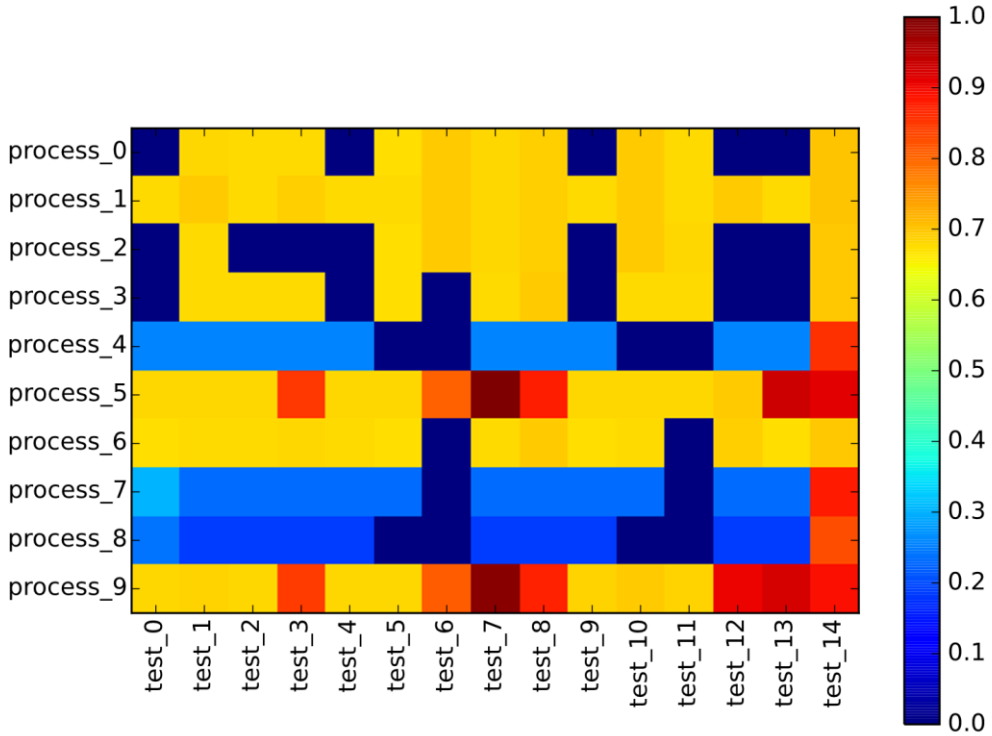


Figure 6 Process usage by different tests

8. Conclusion

The two contributions of this thesis are an instrumented SystemC kernel and a suite of tools for offline processing and visualization of simulation traces. These tools can be used to observe various aspects of virtual platforms to aid design exploration, fault discovery, and performance analysis. Some of these aspects were covered in chapter 6 and can be forwarded to developers to aid platform development.

One of the most interesting aspects remains the way in which tuning the simulation quantum affects its execution. While some tools have already been introduced and used to identify outliers for performance analysis, there is still the issue of determining how exactly software tests depend on changes in the quantum. As a starting point, tools for measuring the difference in processes execution and timing have been introduced. However, the results obtained from the tools are difficult to interpret, and as such an empirical approach with statistical analysis, like the one proposed in [14] for example, seem to be the logical next step. In other words, determining the safe range of change in timing can be determined by running a set of tests on different versions of a virtual platform and determining the safe range based on the timing from tests that failed and passed.

Another next step in the development of observability is to trace TLM-2.0 transport mechanisms such as the b_transport functions. The support for this higher abstraction level would remove noise from a simulation trace, allowing developers to concentrate purely on investigating the communication between modules of a virtual platform, employing lower level traces as points of interest are determined.

There are also more immediate next steps to take. One of them is expanding current tracing capability to include the delta cycles of an operation. While delta cycles are not particularly useful for determining faults, they could be used to compare two executions of different platform versions. Another feature would be storing the contents of the simulation sets introduced in section 3.6.2 in the event of a simulation crash. This would allow to quickly determine the last executing process, which is where debugging would start.

Finally, there is the issue of optimizing the storage of trace data. In the current state traces are in the 10s of megabytes, so storing traces from multiple tests and versions of a virtual platform can get unfeasible. One approach is to employ data compression techniques such as described in [15]. Alternatively, a graph database could be employed to store trace data. This approach would require a redesign of the visualization and processing tools. However, it would enable the usage of semantic queries, potentially simplifying the model exploration process.

References

[1] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005),* pp. 1-412, 1 2012.

[2] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005),* pp. 415-545, 1 2012.

[3] "Synopsys Virtual Prototyping Solutions," [Online]. Available: https://www.synopsys.com/verification/virtual-prototyping.html.

[4] "Cadence Virtual System Platform," [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/software-driven-verification/virtual-system-platform.html.

[5] "Accellera reference SystemC implementation," [Online]. Available: http://accellera.org/downloads/standards/systemc.

[6] D. Große, R. Drechsler, L. Linhard and G. Angst, "Efficient Automatic Visualization of SystemC Designs," in *Forum on specification and Design Languages (FDL)*, 2003.

[7] P. A. Hartmann, K. Güttner and W. Nebel, "Advanced SystemC Tracing and Analysis Framework for Extra-Functional Properties," in *Applied Reconfigurable Computing: 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings*, K. Sano, D. Soudris, M. Hübner and P. C. Diniz, Eds., Cham, : Springer International Publishing, 2015, pp. 141-152.

[8] F. Rogin, C. Genz, R. Drechsler and S. Rülke, "An Integrated SystemC Debugging Environment," in *Embedded Systems Specification and Design Languages: Selected contributions from FDL'07*, E. Villar, Ed., Dordrecht, Springer Netherlands, 2008, pp. 59-71.

[9] S. Lagraa, A. Termier and F. Pétrot, "Data Mining MPSoC simulation traces to Identify Concurrent Memory Aaccess Patterns," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013.

[10] T. Uno, M. Kiyomi and H. Arimura, "LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets," in *FIMI '04, Workshop on Frequent Itemset Mining Implementations*, 2004.

[11] D. Becker, M. Moy and J. Cornet, "SycView: Visualize and Profile SystemC Simulations," in *{3rd Workshop on Design Automation for Understanding Hardware Designs, DUHDe 2016 }*, Dresden, 2016.

[12] "Valgrind," [Online]. Available: http://valgrind.org/.

[13] D. C. Black, J. Donovan, B. Bunton and A. Keist, SystemC From the Ground Up, 2nd ed., Springer, 2010.

[14] E. Anceaume and Y. Busnel, "Sketch *-Metric: Comparing Data Streams via Sketching," in *2013 IEEE 12th International Symposium on Network Computing and Applications*, 2013.

[15] K. Mohror and K. L. Karavanic, "Evaluating Similarity-based Trace Reduction Techniques for Scalable Performance Analysis," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, 2009.

[16] "SoCLib," [Online]. Available: http://www.soclib.fr/trac/dev.