Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semester Thesis

Scalable Distributing Mutli-Linux System for DOL Application

Fabian Hugelshofer
fabianhu@ee.ethz.ch

Prof. Dr. Lothar Thiele
Computer Engineering Group

Advisors: Dr. Iuliana Bacivarov, Kai Huang

Department of Information Technology and Electrical Engineering
Swiss Federal Institute of Technology Zurich

Autumn 2007

Abstract

To master the complexity of highly parallel applications for embedded systems, the SHAPES project is developping a scalable multiprocessor hardware architecture. The project also develops a toolchain which supports the hardware/software codesign of such scalable systems.

An important aspect of the design process is to find an optimal mapping of the application onto the hardware. The Distributed Operation Layer (DOL) helps with this task. To get a base for mapping decisions, DOL analyzes the application by functional simulation. Until now this functional simulation was running on a single system and could not use multiple CPU cores. The task of this thesis is to distribute the functional simulation among multiple Linux machines in the network to keep DOL itself scalable.

The results are promising. Not only the functional simulation has been accelerated by its distribution, but also a library has been developed which supports the distribution of arbitrary simulations based on SystemC. The library is particularly suited to distribute functional and untimed or approximate-timed TLM simulations. It is the first which aims at distributing simulations on such higher levels of abstraction. No modifications have been made to the simulation kernel.

Contents

List of Figures iv

List of Tables v

List of Listings vi

1 Introduction 1
1.1 Distributed Operation Layer . . . 1
1.2 Task . . . 2
1.3 Related Work . . . 3
1.4 Thesis Layout . . . 3

2 SystemC Distribution Library 4
2.1 Introduction to SystemC . . . 4
2.1.1 Scheduler . . . 5
2.2 Design Aspects . . . 7
2.2.1 Communication using Sockets . . . 7
2.2.2 Simulating Step-Wise . . . 8
2.3 Implementation . . . 10
2.3.1 SCD Simulator . . . 10
2.3.2 Remote Channels . . . 10
2.3.3 Controlling the Simulation . . . 13
2.3.4 Socket Handling . . . 18
2.3.5 Simulation Loop . . . 22
2.3.6 Usage . . . 23

3 Integration into DOL 26
3.1 Architecture Specification . . . 26
3.2 Mapping Specification . . . 28
3.3 Validation and Visualization . . . 28
3.4 Code Generation . . . 29
3.5 Running the Simulation . . . 30
3.6 Profiling . . . 31

4 Results 32
4.1 Methodology . . . 32

4.2 Measurements and Discussion 32

5 Conclusions and Future Work **38**
5.1 Conclusions 38
5.2 Future Work 39

Bibliography **41**

A CD-ROM Content **43**

B Step-by-Step Instructions **44**
B.1 Preparing DOL 44
B.2 Running the MPEG-Example 44

List of Figures

2.1 SystemC Scheduler 6
2.2 Simulator Overview 10
2.3 Control Modules 14
2.4 Synchronization 15
2.5 Control State Machines 17
2.6 Socket Handling 19

3.1 Mapping Visualization 28

4.1 Application Example 34

List of Tables

4.1 Simulation Runtime 33
4.2 Runtime Simulators 35
4.3 Runtime Processes 36

List of Listings

2.1 Remote Channel Interfaces 11
2.2 Remote Output FIFO Channel 12
2.3 Socket Event Handler Interface 20
2.4 Socket Poller 20
2.5 Simulation Main Loop 21
2.6 SCD Usage 23
3.1 Architecture Specification 27

1 Introduction

The complexity of applications for embedded systems is steadily increasing. Single processor architectures start to fail to satisfy the demand of modern multimedia, telecommunication and other signal processing applications. More flexible and scalable computing platforms are needed. Heterogeneous multiprocessor system-on-chip (MpSoC) architectures fulfill these requirements but impose new challenges on the design process.

With MpSoC platforms a various number of different hardware configurations can be allocated. It has not only to be decided on the kind of processing elements but also on their quantity and their interconnections. Mapping the application onto the allocated hardware architecture again is possible in many different ways. The overall design space for MpSoC systems is therefore massively bigger than for single processor systems. To be able to handle the design complexity while still being flexible to easily adopt the architecture or the application at any time, new design methodologies are needed.

1.1 Distributed Operation Layer

The European research project Scalable Software Hardware Architecture Platform for Embedded Systems (SHAPES) [1] developed a scalable MpSoC hardware architecture and deals with the challenges of developping software for such systems [2]. One major software aspect is to find an optimal mapping for an application onto an allocated hardware architecture. SHAPES' Distributed Operation Layer (DOL) [3] covers this problem.

To use DOL to find a mapping, the designer must specify the application to be mapped. The specification exposes the parallelism of the application and is completely separated from any architectural aspects. DOL uses the class of Kahn process networks [4] as its application abstraction model. A process network is composed by processes which are connected by first-in first-out (FIFO) channels. Processes can only perform local computations, read data from input channels and write data to output channels. The structure of the process network, a directed graph whose nodes represent processes and whose directed edges represent communication channels, has to be specified in XML. The functionality of the application is defined by the behavior of the processes. Each process has to be specified in plain C/C++ whereby a set of DOL specific coding rules have to be respected.

A specification has to be given for the target architecture, too. It is also defined in XML and contains structural, performance and parametric data. The structure specifies the platform's

resources such as processors, memories, hardware channels and their interconnections. Performance data gives information about clock frequencies, communication delays, throughputs and therelike. Parametric data can for example define memory sizes or operating system parameters. As the mapping optimization is performed at the system level, such an abstract representation of the architecture is sufficient. Not modelling at a lower level of abstraction allows for a much faster design space exploration.

The application and architecture specifications are the input of DOL. To have a base for optimization decisions, profiling data of the application and the hardware is collected first. By platform benchmarks more performance information can be gained. Functional simulation of the application reveals the number of process invocations and the amount of data transmitted over each channel.

An iterative design space exploration and estimation cycle then tries to find optimal mappings of the application onto the architecture. Mapping includes binding of processes to processors and communication channels to hardware channels as well as scheduling of shared resources. Exploration of mappings is based on evolutionary algorithms. Candidate solutions are then analytically analyzed and simulated. Performance estimation results are fed back into optimization for further improvements. DOL deals with multiple conflicting objectives. The current implementation tries to minimize computation and communication time. The result is therefore not a single optimal mapping but a set of Pareto optimal solutions. Mappings are also represented in XML.

DOL targets an efficient execution of parallel applications on a heterogeneous MpSoC. It makes the design process scalable and keeps it flexible. The resulting mapping can be used by other tools to generate the program code for the different processors.

1.2 Task

As described in the previous section, DOL performs a functional simulation of the application to estimate its characteristics on a high level of abstraction. Main figures gained during this simulation are the invocations of each process and the amount of data transferred over each channel. In an additional architecture depending performance estimation the runtime of each process on different processors is obtained. This information is a valuable base for mapping decisions. The functional simulation further allows to validate the correctness the application.

Currently the functional simulation runs on a single computer which limits the maximum computation power. A very complex application could not be simulated in a reasonable time. This conflicts with SHAPES' core objective of scalability. The task of this thesis is to distribute the functional simulation among an arbitrary number of Linux systems which are connected by a network. The target is to be able to speedup the simulation in a scalable way. The scalable distribution backend has to be integrated into the automated DOL toolchain.

The current functional simulation is performed using SystemC as simulation kernel. SystemC is described in more detail in section 2.1. It does not support the distribution of a simulation by itself and neither does it provide a network communication API. Instead of developing a new solution which is completely different, SystemC is extended by a library which allows to distribute the simulation. The advantage is that SystemC is fast and the existing toolchain can be used. As SystemC is widely used for system modeling such a distribution library can also be used by other projects.

1.3 Related Work

Distributed simulation is a need in modern design processes. Several works have addressed the problem to distribute a simulation based on SystemC [5, 6, 7, 8]. Fin et al. [5] are enabling vendors to provide simulation models of their intellectual property cores without disclosing their internal structure. They do not give information about implementation specific details. The other works have all been motivated by the complexity of simulation on register transfer level (RTL). The work of Meftali et al. [7] is only applicable for clocked simulations. The solutions of Trams [6] could be used for an untimed functional simulation but would have an unacceptable performance handicap due to synchronous communication. Cox [8] uses asynchronous communication which would be well suited but he modifies the SystemC kernel which is not desired for portability reasons.

This thesis is the first work which aims at maximizing performance of a distributed functional simulation with SystemC. Further it is the first which uses asynchronous communication without modifying the SystemC kernel.

1.4 Thesis Layout

First, chapter 2 describes the development of the SystemC distribution library. The library is then integrated into the DOL framework in chapter 3. Chapter 4 analyzes the performance achievements of this work based on a concrete application example, draws conclusions and outlines possible future work.

2 SystemC Distribution Library

This chapter provides an overview about SystemC and describes the design and implementation of a SystemC Distribution Library (SCD). This library allows to distribute arbitrary SystemC simulations. Distribution means that an arbitrary number of Linux systems connected by a network can share the workload of the simulation. The objective is to obtain a maximum reduction of simulation time by incorporating all computational resources available. The development of such a library is necessary as SystemC has been designed for single host simulation only and does by itself not provide means to distribute. The library extends SystemC and does not require modifications to the simulation kernel.

Each Linux process running a SystemC simulation is denoted as a simulator. Several simulators can run on a single system. Channels interconnecting simulators are called remote channels.

2.1 Introduction to SystemC

SystemC is a C++ based library to model systems composed of both hardware and software components. It allows to design and verify systems on different levels of abstraction. This supports the development of complex systems. Components can be specified and their interactions analyzed in an early design phase and independent of the target system. SystemC is specified in IEEE standard 1666 [9]. An implementation is provided by the Open SystemC Initiative (OSCI) [10]. It is provided under SystemC Open Source License which grants the royalty-free right to use, modify and distribute its code or derivative work. For details refer to the license agreement. The OSCI implementation is used in this project.

The main components to model a system are processes and channels. Processes describe functionality and allow parallel activities. They are executed without interruption until they yield control to the simulation kernel by returning or waiting for an event to occur. This is known as co-operative or non-preemptive multitasking. OSCI SystemC executes only one process at any time even if the hardware supports execution of concurrent processes. The consequence is that not all computational resources of modern computer systems can be used.

Channels can connect two ore more processes and allow them to exchange data. Primitive channels do not contain any processes. Hierarchical channels can be composed by processes and primitive channels and can therefore be more complex. SystemC provides a rich set of predefined channels and custom channels can be defined.

2.1.1 Scheduler

To be able to distribute a simulation based on SystemC, it is important to understand in detail how scheduling works.

The simulation is event based. Execution of processes depends on the occurrence of events they are sensitive to. The kernel keeps different event queues and a set of runnable processes. Events are generated by calling function `notify` with or without arguments:

- `notify()` – Creates an immediate event.
- `notify(SC_ZERO_TIME)` – Creates a delta event.
- `notify(timeval)` – Creates a timed event at time `timeval` relative to the current simulation time.

The meaning of these different kind of events will become clear soon.

Figure 2.1 shows a flowchart of the scheduling behavior. Initially all processes are made runnable by putting them into the set of runnable processes. As long as runnable processes exist, one of them is selected and executed until it returns or waits for an event to occur. If the process has generated immediate events, all processes sensitive to these events are made runnable immediately. Checking for runnable processes and executing them constitutes the evaluation phase.

If the evaluation phase has been left, the scheduler calls the `update()` member function of channels which generated update requests while being accessed by executed processes. This allows channels for example not to take on new values immediately but before the next delta cycle. Only delta events and timed events can be generated by `update` functions.

After all update requests have been processed, the scheduler makes all processes which are sensitive to pending delta notifications runnable. If such processes exist, the simulation enters the evaluation phase again. Simulation is said to be advanced by one delta cycle. Simulation time remains unchanged by this step.

When the evaluation phase is left and no delta events exist, the scheduler checks for timed events. If such events exist, simulation time is advanced to the time of the earliest timed event. All processes being sensitive to timed events at this simulation time are made runnable and the evaluation phase is entered. If no pending timed events exist the simulation terminates.

The simulation can be started from the main function `sc_main`. SystemC knows about different ways to do so:

- `sc_start()` – Runs the simulation until no more events exist as described above. After returning, simulation time is infinity and the simulation shall not be started again.
- `sc_start(timeval)` – Runs the simulation until simulation time has been advanced by `timeval`. Processes sensitive to timed events at this simulation time are

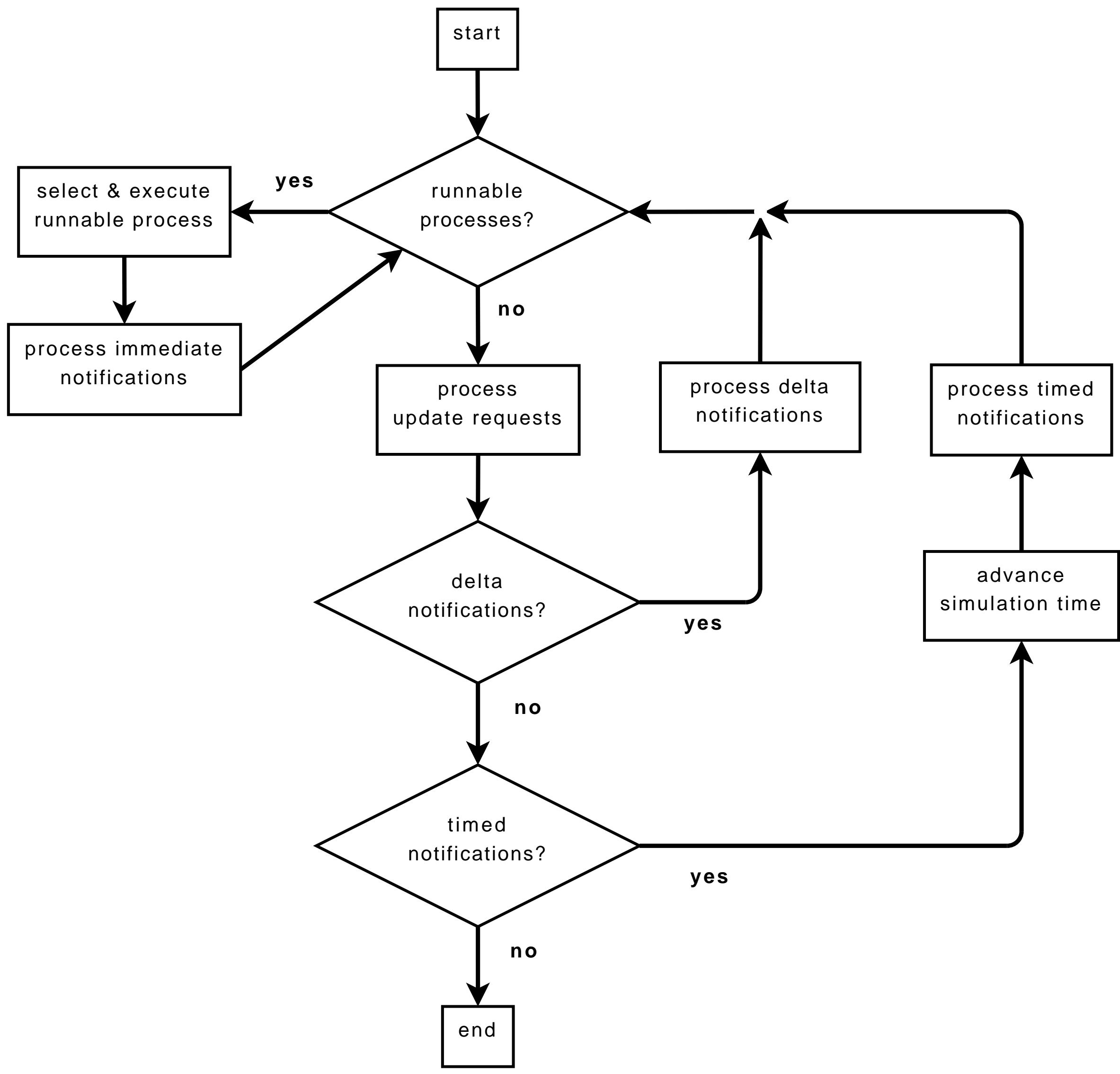


Figure 2.1: SystemC Scheduler

made runnable but but the evaluation phase is not entered yet. After returning, simulation time is always `timeval` later than before the function call. Continuing the simulation by another call to `sc_start` is possible.

- `sc_start(SC_ZERO_TIME)` – Runs the simulation for one delta cycle only. Processes sensitive to delta events of the next delta cycle are made runnable before the function returns. Continuing the simulation by another call to `sc_start` is possible.

2.2 Design Aspects

2.2.1 Communication using Sockets

Communication channels in Kahn Process Networks [4] have exactly two endpoints. A natural way to implement them as remote channels is by the use of Unix Sockets. Sockets are also suitable to implement any, possibly bidirectional, SystemC channel. If a channel has more than two endpoints it can be modeled as a hierarchical channel. As robustness to packet loss and packet reordering is required for correct operation, TCP is chosen as transport protocol. The overhead for packet acknowledgement should be negligible in fast local networks. However UDP might be considered for simulation over slow networks such as the internet. In this case additional mechanisms would be necessary to get the same quality of service.

As no truly parallel execution is possible with a co-operative scheduling, blocking sockets can only be used for synchronous communication where communication takes place at predefined synchronization points. Further it is necessary that at this points data is exchanged in any case. This approach is being taken by Meftali et al. [7]. In their work remote channels exist only between a master simulator and its slaves. After every simulated clock cycle the master gets the values of all its remote input channels from the clients. Then it sends the current value of all its remote output channels to the clients and another clock cycle is simulated. This is suitable for clocked simulations and channels of type `sc_signal` which model hardware wires. Channels which do not allow to set new values every clock cycle such as FIFOs, would require a dummy message to indicate that nothing changed. If no message was transmitted at all, reading simulators would block. Synchronous communication at the end of a clock cycle is natural, as synchronization to advance simulation time is necessary anyway. The functional simulation in DOL does not have a notion of time. Using synchronous communication would therefore require the introduction of artificial synchronization points. This approach is taken by Trams [6]. Assume all simulators but one have reached the synchronization point. As long as the last one did not reach this point, too, all other simulators would have to wait. This waste of resources is not acceptable and makes asynchronous communication necessary.

Let's still study to the case of blocking sockets. Without synchronous communication data can asynchronically be sent at anytime. Assume a case with one process $\mathscr{A}$ on one simulator and two processes $\mathscr{B}$ and $\mathscr{C}$ on another simulator. If process $\mathscr{A}$ wants to send to $\mathscr{B}$ and $\mathscr{B}$ can not accept data at the moment, process $\mathscr{A}$ will block. As scheduling is non-preemptive not only

the process but the whole simulator will do so. If now process $\mathscr{C}$ wants to send to process $\mathscr{A}$ it will block too as $\mathscr{A}$ is not reading. As now also the second simulator is blocked, process $\mathscr{B}$ will never be executed again. But this would be necessary to unblock process $\mathscr{A}$. A deadlock occurred and it becomes clear that non-blocking sockets have to be used for asynchronous communication.

With non-blocking sockets the introduction of a new SystemC process handling all socket communication makes sense. A generic process wanting to read or write from or to a socket delegates this operation to the communication process and waits for an event which indicates completion of the operation. As the `send` or `recv` function return immediately, if the operation is currently not possible, the communication process can retry later. To yield execution to other processes, which might be runnable at that time, the communication process generates an event and waits for this event to occur, in order to be reactivated again. Some aspects have to be considered:

- As soon as no events exist on a single simulator this simulator terminates. The continuous generation of events by the communication process is therefore essential. Else a condition where all other processes are waiting for data to come in, would lead to an unintended termination.
- Continuously generating events, while actually nothing is simulated, constitutes a busy loop. The communication process should in this case be able to notice that no events exist and pause the simulator. How this can be done is described in the following section. Pausing could be achieved by using `sleep` or by waiting for socket activity using `poll` or `select`.
- The communication process must be able to detect global completion of the simulation. In this case it stops generating reactivation events and the simulation terminates.
- In a distributed simulation it is necessary that advancing simulation time is synchronized. Simulation time has to be advanced, if no simulator has events for the current simulation time. Assume a situation where simulation time has to be advanced to t_1 and one simulator's earliest event is at $t_2 > t_1$. In this case it still has to pause the local simulation, until time is advanced to t_2 or new events are generated for t_1 by incoming data. It can not just let the simulation run as the events at t_2 would be processed. Therefore it must be able to check, whether the next event is at t_1 or later.

The approach with the communication process presented here is certainly possible. In the next section another approach is presented which is similar in many points but is better structured. This other approach is used for the implementation.

2.2.2 Simulating Step-Wise

As shown in section 2.1.1 it is possible to run the simulation for only one delta cycle using `sc_start(SC_ZERO_TIME)`. This is exploited for the SystemC distribution library. The

key principle is that after every delta cycle the simulation returns to sc_main where communication and synchronization takes place. Communication means transmitting data over remote channels. Synchronization is also referenced to as controlling simulation activity. It globally synchronizes to advance simulation time or to terminate the simulation. With non-blocking sockets it is possible to check, if data can be received or sent on channels. This does not enforce a fixed point of synchronization solely for the purpose of communication and the simulation can be continued immediately, if no data has arrived or another remote simulator is not able to accept data a channel tries to send. The advantage over the communication process presented in the previous section is that communication and synchronization is completely decoupled from the SystemC simulation. Further complete control over simulation time is provided. Simulation time is only advanced if sc_start(timeval) is called and this is possible independently of the existence of events at the new or later simulation time.

As with the communication process approach it is still necessary to determine when a local simulation is ready to advance simulation time or is ready to terminate. In this case the simulator can pause. Formulated differently, it is necessary to know if further events exist at the current simulation time and if events exist at a later simulation time. IEEE 1666 does not specify functions which provide this information. The OSCI implementation introduced a function sc_pending_activity_at_current_time() in the newest version 2.2.0. The documentation is not up-to-date yet, but this function is described in the release notes. The function returns **true**, if there exist events at the current simulation time. As long as this is the case a simulator has to run the simulation for another step. Therafter it can wait for new events to be generated by remote channels or for synchronization to advance simulation time or to terminate. To distinguish between the two later cases sc_get_curr_simcontext()->next_time() can be used. This function returns the time of the next timed event or SC_ZERO_TIME if no such event exists. If there exists no event at the current simulation time and this function returns SC_ZERO_TIME the simulator is ready to terminate, else it has events in the future and is ready to advance simulation time. However next_time() is a member function of class sc_simcontext which is declared deprecated in IEEE 1666. This means that no implementation is required to provide it and it could be removed in future from OSCI SystemC. Using both these implementation specific and unstandardized functions is not a perfect solution, but it circumvents the need of modifications to the SystemC kernel. If they were not present anymore in future releases of OSCI SystemC, the open source license would still allow to reintroduce them. Therefore the usage of these functions is considered acceptable.

As communication takes place after each delta cycle, the designer using the SystemC distribution library must pay attention not to use immediate events exclusively. In this case the simulation would always run until no more events existed, before data is transmitted to other simulators. This might prevent other simulators from simulating during this time. Partly generating delta events with notify(SC_ZERO_TIME) instead of immediate events gives the developer control over when communication takes place.

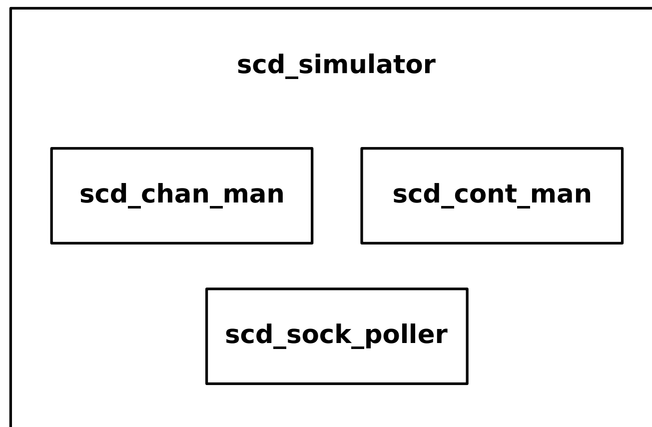


Figure 2.2: Simulator Overview

2.3 Implementation

2.3.1 SCD Simulator

Class `scd_simulator` is the main module of the distribution library. It provides the interface to the designer using the library and runs the simulation. It is composed by three other modules as shown in figure 2.2. The channel manager class `scd_chan_man` handles the remote channels. This means that it feeds data from remote simulators into the channel endpoints of the local simulation and the other way round. How to specify remote channels and how this works in detail is described in section 2.3.2. The control manager class `scd_cont_man` synchronizes the remote simulators regarding when to advance simulation time and when to terminate. See section 2.3.3 for details. These two modules are related to the simulation and provide an abstraction of the main tasks: synchronization and communication. The last module the socket poller class `scd_sock_poller` initiates transmission on sockets in both directions. It interacts with every object which has to communicate over the network and simplyfies the handling of the non-blocking sockets. The socket poller is presented in section 2.3.4.

2.3.2 Remote Channels

Remote channels have been defined to have exactly two endpoints which reside on different simulators. The endpoints are accessed by SystemC and should not behave differently than normal channels. Therefore it was chosen to implement them as primitive channels. Sending data from one such channel stub to another is the task of the channel manager and is transparent to SystemC. For transmission every endpoint pair has a TCP connection associated. Remote channels can be bidirectional. A channel stub accepting data from its remote endpoint is

```
class scd_rem_chan_in_if {
public:
    virtual size_t free() const = 0;
    virtual void receive(const void* buf, size_t len) = 0;
};

class scd_rem_chan_out_if {
public:
    virtual size_t available() const = 0;
    virtual const void* send() const = 0;
    virtual void remove(size_t len) = 0;
};
```

Listing 2.1: Remote Channel Interfaces

called a remote input channel and a channel stub sending data to its remote endpoint is called a remote output channel.

Implementing remote channels is up to the designer using the distribution library. Given an arbitrary two-sided primitive channel it must first be separated into two endpoints of type `sc_prim_channel`. This especially includes the separation of the interface used by the simulation. Assume a unidirectional channel providing a read and a write function. Implementing this channel as a remote channel would result in a primitive channel providing only the read function and a primitive channel providing only the write function. In a second step this channel stubs must provide an interface to the channel manager. Listing 2.1 shows the interfaces which have been defined for remote input and output channels. A remote channel can implement either one or both of them. The channel manager will use these interfaces after every simulated delta cycle to check for every remote channel if data has to be transmitted.

Remote output channels have to implement interface `scd_rem_chan_out_if`. Function `available()` shall return the number of bytes which are available to be sent to the other endpoint. If this function returns a value greater than zero, the channel manager will try to send up to the returned amount of bytes. It will call `send()` once to get the memory location where it starts reading from. From this location at most `available()` bytes are read. As the socket might not be ready to send any amount of bytes, there is no guarantee that the data will actually be sent. Therefore the channel shall not discard anything. If it was possible to send everything or part of the available data, the channel manager will inform the channel about the number of bytes successfully sent by calling `remove(len)`. `len` corresponds to the number of bytes which have been sent and is greater than zero and less or equal to the number of bytes which have been returned by `available()`. Function `remove` can update the channel's state and can generate events to wake up processes if necessary. The channel manager will repeat this procedure until either it is not possible to send all data available or the channel has nothing to send anymore.

Listing 2.2 shows the implementation of the remote output channel for a FIFO channel based

```
1  #include <systemc>
2  #include "scd_rem_chan_if.h"
3
4  class scd_rem_fifo_out : public sc_prim_channel,
5          public scd_rem_chan_out_if {
6  private:
7      int _size, _num_elements, _first;
8      char* _data;
9      sc_event _read_event;
10
11 public:
12     scd_rem_fifo_out(sc_module_name name, int size):
13             sc_prim_channel(name), _size(size),
14             _num_elements(0), _first(0) {
15         _data = new char[size];
16     }
17
18     ~scd_rem_fifo_out() { delete _data; }
19
20     write(char c) {
21         while (_num_elements == _size)
22             wait(_read_event);
23         _data[ (_first + _num_elements) % _size ] = c;
24         _num_elements++;
25     }
26
27     size_t available() const {
28         if (_num_elements <= _size - _first)
29             return _num_elements;
30         else
31             return _size - _first;
32     }
33
34     const void* send() const { return _data + _first; }
35
36     void remove(size_t len) {
37         _first = (_first + len) % _size;
38         _num_elements -= len;
39         _read_event.notify();
40     }
41 };
```

Listing 2.2: Remote Output FIFO Channel

on the `simple_fifo` example of OSCI SystemC. At this endpoint single bytes can be stored in a buffer. The content of this buffer is then transmitted to the other endpoint where it can be read in first-in/first-out order. The buffer is implemented as a ring buffer. Function `write` is used to store a single byte. If the buffer is full, the function will wait and the calling process will be suspended. After one delta cycle the channel manager calls `available()` which returns the number of bytes that can be linearly read from the ring buffer without having to wrap around. If this number is greater than zero, the channel manager will get the pointer to the first of these bytes by calling `send()`. If some bytes could be sent the channel manager will call function `remove` and these bytes will be removed from the buffer. An immediate event will be generated which will wake up suspended processes waiting for the buffer to become ready to accept more data. These processes are made runnable immediately and will be executed during the next simulation cycle. A delta event would take effect one cycle later as processes sensitive to it would be made runnable at the end of the next delta cycle.

Remote input channels have to implement interface `scd_rem_chan_in_if`. If the channel manager can read data from an input channel's socket, it will call `free()`. This function shall return the maximum number of bytes the channel can accept and store in a continuous memory region. If this number is greater than zero the channel manager reads at most this amount of bytes from the socket and stores it in a buffer. Then it calls `receive(buf,len)` where `buf` is the buffer containing the received data and `len` is the number of bytes actually received. `len` is greater than zero and smaller or equal the number of bytes which has been returned by `free()`. The channel then shall process the data received and can generate events to wake up processes if necessary. The channel manager repeats this procedure until either no more data can be read from the socket or the channel can not accept more data. An example implementation of a remote input channel is omitted, as it would be similar to the implementation of the remote output channel.

Transmission between the two channel endpoints takes place in a stream-oriented fashion as it is provided by the underlying TCP sockets. The designer can implement arbitrary communication protocols. Implementing a FIFO channel is one of the easiest ones. Special care has to be taken if other datatypes than bytes have to be transmitted. Else the stream-like communication could lead to corruption, if computers with different endianness are participating in the same simulation. This is especially the case for Intel x68 and SPARC machines. Using a single byte order for network transmission evades this problem.

2.3.3 Controlling the Simulation

In a distributed simulation it is possible that some simulators have no events and therefore are pausing while others are simulating. Paused simulators can be reactivated at any time. A single simulator has only local knowledge. It can check, if it has events to be processed or if it should pause. To know when the simulation completed globally, synchronization is necessary. For this purpose the simulators exchange messages indicating their simulation state. Three simulation states are defined:

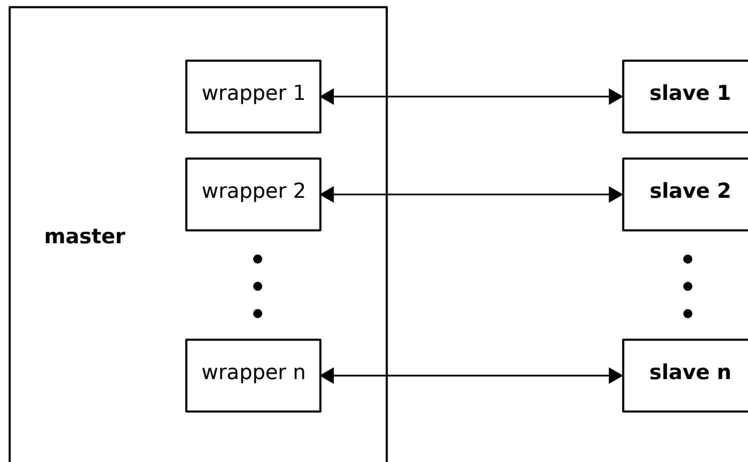


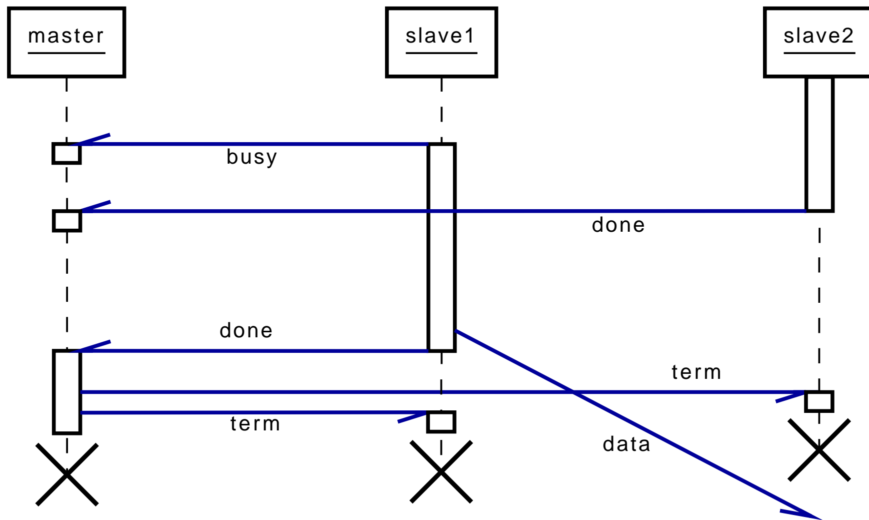
Figure 2.3: Control Modules

- busy – The simulator has events at the current simulation time and is simulating.
- idle – The simulator has no events at the current but at a later simulation time and is waiting for the simulation time to be advanced.
- done – The simulator has no events at all and is waiting for the simulation to be terminated.

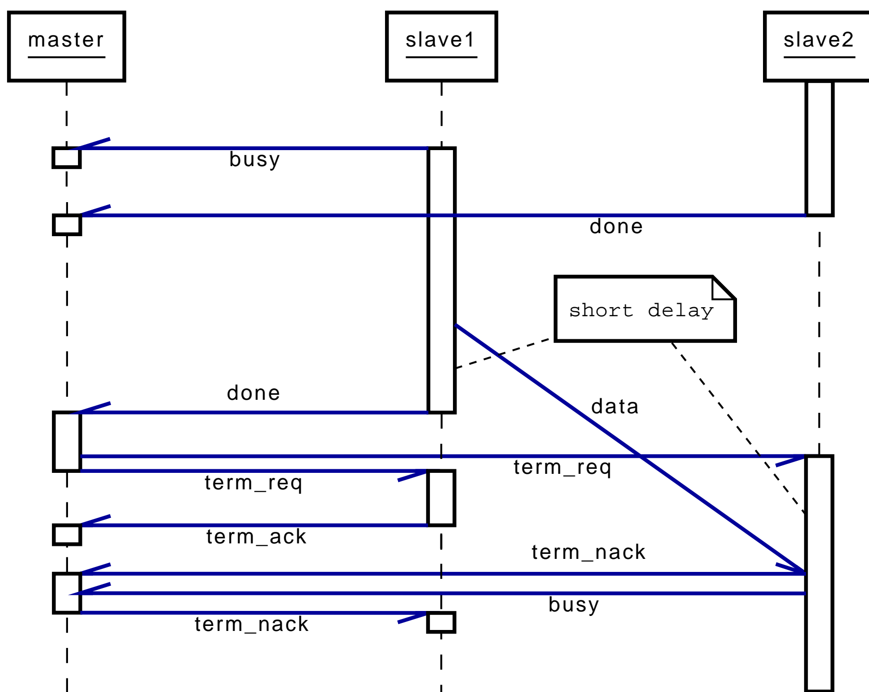
The idea is that the simulation terminates as soon as all simulators are done. If there exist idle or possibly done simulators but no busy ones, simulation time is advanced. To aggregate the global simulation state a centralized approach is taken. A master simulator is connected with every other simulator participating in the simulation. These slave simulators inform the master about their local simulation state. The master aggregates this information and induces simulation time to be advanced or simulation to be terminated, if the global state is idle or done respectively. As the communication of the local state takes some time, the master has not an accurate view of the slaves' local state at every time. Three finite state machines have been implemented. The master controller state machine, the slave state machine and the slave wrapper state machine which represents the master's current view of a slave's state. This is shown in figure 2.3.

If the master sends a termination message to the slaves as soon as all simulators are done, a race condition might occur which leads to unexpected termination of the simulation. A sequence diagram of such a race condition is shown in figure 2.4(a). Three simulators are shown: one master and two slaves. At the beginning only *slave2* is busy. The other simulators are done. After a while *slave1* receives data and gets reactivated. The arrival of this data is not indicated, but it might come from *slave2*. As *slave1* can now contintue the simulation it sends a busy message to the master to reflect the new state. This is not actually part of the race condition, but illustrates how controlling works.

In a next step *slave2* completes its local simulation and sends a done message to the master controller. *slave1* continues running the simulation. At the end it sends some result to *slave2*,



(a) Race Condition



(b) Termination Request

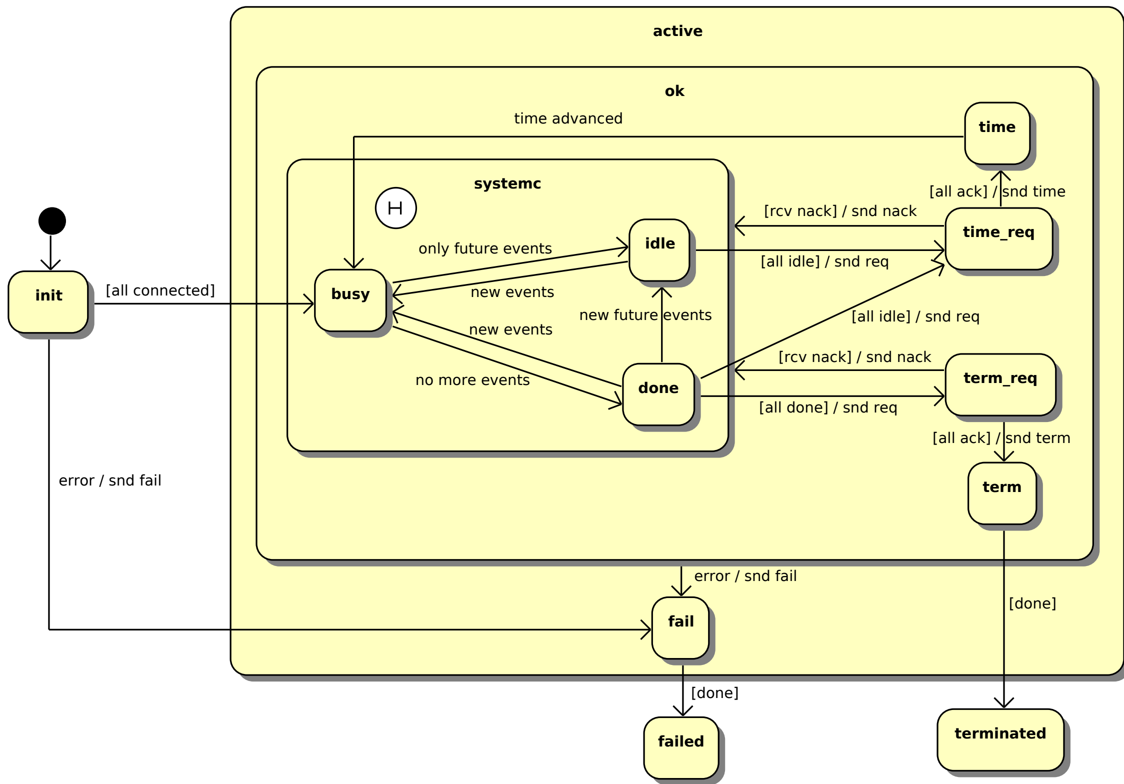
Figure 2.4: Sequence diagrams of synchronization for termination.

but the transmission gets delayed by the host system or the network. As *slave1* has nothing to simulate anymore, it also sends a done message to the master. Because all slaves are done, the master has wrongly the impression that the simulation completed. It sends a termination message to all slaves which then will terminate. The data, which did not arrive fast enough, can not be processed anymore. This problem exists for the synchronization for termination and for the synchronization for advancing simulation time.

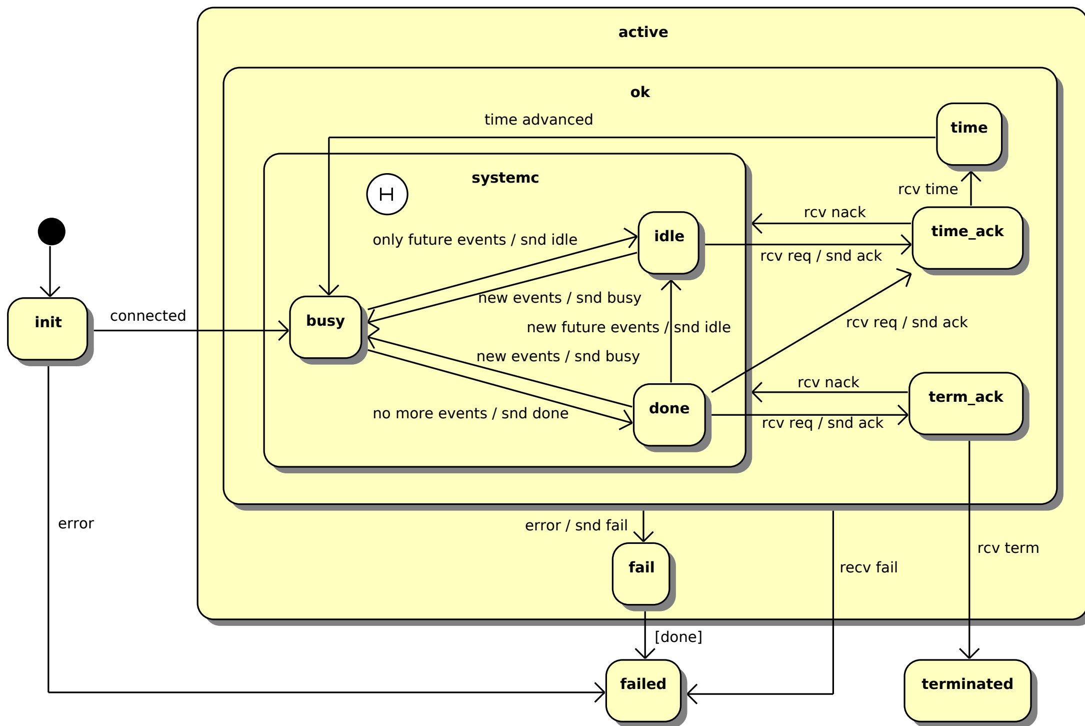
To prevent this race condition, the master does not immediately send a termination message to the slaves, as soon as all of them are done. Instead it makes sure to be in a situation of global completion, by first sending a termination request. The slaves reply to this request with an acknowledgment or a non-acknowledgment. If all slaves reply with an acknowledgment, the master sends the termination message. Else it aborts the termination attempt. Using the triangle inequality, it is expected that the delay of the done message coming from *slave1* plus the delay of the termination request going to *slave2* is bigger than the delay of the data transmission from *slave1* to *slave2*. Therefore the termination request will not be acknowledged by *slave2* and the simulation continues. The dependency on the network delay lets the synchronization adapt to new networking environments.

However it turned out, that the termination request alone did not solve the problem of the race condition. In some cases the simulation still terminated unexpectedly. This can be explained as follows: *slave1* finishes its local simulation and writes the resulting data to the channel. Even if the channel manager is still transmitting this data, *slave1* will send a done message, as it has no more events to process. On the other side it is possible that *slave2*'s channel manager is receiving data, but no events have yet been generated to reactivate the simulation. In this situation a termination request would be acknowledged. The overall solution is now presented in figure 2.4(b). Additionally to the termination request a delay based approach is taken. As long as a busy simulator is sending, it is not allowed to change its state to done. On the other side will a termination request not be acknowledged as long as the simulator is receiving. To detect whether a simulator is sending, it is not possible just to check if the remote channels have data to be sent. It might be the case that some channels have data available, but this will never be read by the other endpoint. The solution is to check whether there is activity on the sockets during a specified time interval. This time interval is specified with constant `SCD_CONT_DELAY` and is set to 20 ms. There must be a period of 20 ms with no activity on the sockets before changing state from busy to done. There must also be such a period of no socket activity before the master sends a termination request or a slave answers with an acknowledgment. In the sequence diagram one can observe that *slave2* does not immediately respond to the termination request. But as soon as it observes socket activity due to the incoming data, it sends a non-acknowledgment followed by a busy message indicating its new state. With this solution and a delay of 20 ms very stable interaction has been observed. The cost of this stability is 60 ms synchronization overhead.

Figure 2.5 shows state charts of the master and slave control state machines. The slave wrapper state machine is omitted as it becomes clear implicitly. To remember, the wrapper state machine reflects the master's view on a slave's state. State transitions in the wrapper state machine are triggered only by messages received from the slave or messages sent by the master.



(a) Master



(b) Slave

Figure 2.5: Control State Machines

All controllers start in the initialization state. As soon as the connection between a slave and the master controller has been established, the slave moves to the busy state. The master controller does so, if all slaves are connected. If an error occurs a slave terminates immediately, if it does not already have a connection to the master controller. Else it informs the master by sending a fail message and moves from the fail state to the failed state soon as this message has been transmitted. In case the master observes an error or receives an error message from a slave it sends an error message to all slaves which are connected. This fail message will bring the slaves to the failed state immediately. The master moves to the failed state as soon as all fail messages have been sent. Once the simulation has been started a simulator will terminate, if its control state machine leaves the active super state. This can either be because the simulation failed or because it terminated.

The SystemC super state reflects the state of the local simulation. It consists of states busy, idle and done. Transitions between these states are triggered by class `scd_simulator` which checks, if events exist at the current or later simulation time. As described before, moving from busy to idle or done is only allowed, if there is no socket activity. Master and slave controllers behave the same except that the slave controller will always inform the master about state transitions. As soon as all slaves seem to be done and the master is also in this state, it will send a termination request to all slaves. If some slaves respond with a non-acknowledgement, all slaves will be informed about the abortion of the termination attempt and the master moves back to the done state. If all slaves acknowledge the termination request, the master sends them a termination message which leads the slave to terminate. The master terminates as soon as all such messages have been sent. A slave receiving a termination request while not being done replies with a non-acknowledgement without state transition.

The synchronization for advancing simulation time behaves similar. It is triggered if some slaves seem to be idle and possibly done but none seem to be busy. If all slaves acknowledge their time request, the master determines the time of the globally next event from the idle messages it has received previously. It then sends the time step by which the simulation time should be advanced to the slaves in a time message. All simulators are moved to the time state. As soon as the simulation time has been advanced by class `scd_simulator` they become busy again.

2.3.4 Socket Handling

Network connections are being established between the master controller and each slave controller and between the two endpoints of every remote channel. Further a listening socket is used for every simulator during initialization and different objects are involved until a connection has been established completely. If an operation such as reading or writing has to be carried out on a non-blocking socket, it is possible that it can not be performed immediately. In this case the operation will have to be retried later. Instead of retrying pending operations repeatedly it would be nicer to be informed when the operation will succeed and only retry then. This can be achieved with the Reactor design pattern [11]. In this pattern objects announce their interest in specific events. A central dispatcher monitors all events and calls back an

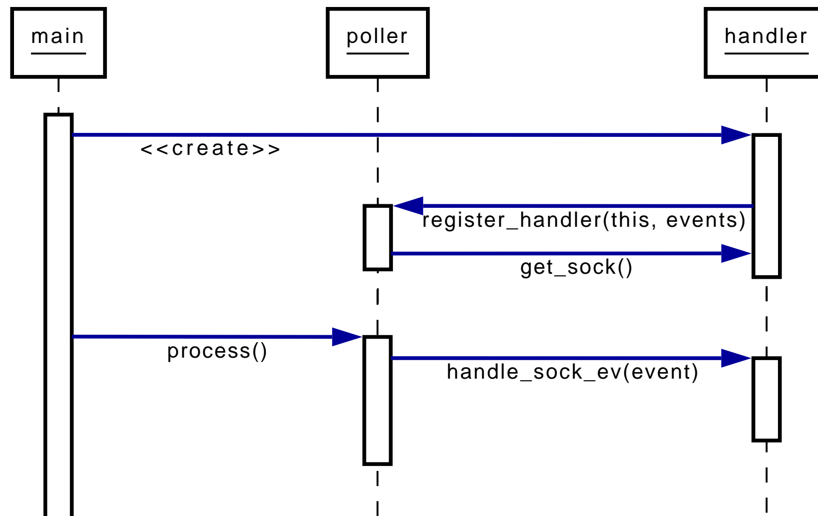


Figure 2.6: Sequence diagram of socket handling

object, if one of its awaited events occurred. The pattern is especially usefull, if many objects are interested in events of the same type. This is the case with our heterogeneous landscape of communicating objects being interested in socket events.

The sequence diagram in figure 2.6 illustrates this concept. Each object using a socket to communicate becomes an event handler by implementing the socket event handler interface. Event handlers register themselves at the central dispatcher called socket poller. While registering they set the events they are interested in. The following socket event flags have been defined:

- `SOCK_EV_READ` – Data has been received and can be read from the socket.
- `SOCK_EV_WRITE` – Data can be written to the socket.
- `SOCK_EV_CLOSE` – The connection has been closed or an error has occurred.

These flags can be concatenated with a bitwise OR. During registration the socket poller calls member function `get_sock()` of the event handler to get the socket which should be monitored. The simulation is being driven by the main simulation loop of class `scd_simulator`. From there the socket poller is called. It uses `poll` to check if registered events have occurred on the sockets. If this is the case, it calls back the corresponding event handler which processes the event. Assume an object wanting to read from a socket. It first registers itself with a read event. As soon reading is possible the poller will call its event handler. There the object can read as much data as possible. Maybe another callback is necessary to complete the read operation, as only part of the expected data has arrived until now. After having finished, the object can deregister itself from the socket poller and will not be called back again.

Listing 2.3 shows the interface which has to be implemented by a socket event handler. Function `handle_sock_ev` is the callback function. It is called by the socket poller, if at least one of the events registered by the handler has occurred. The argument `events` is the bitwise

```
1 class scd_sock_ev_handler_if {
2 public:
3     virtual void handle_sock_ev(sock_ev events) = 0;
4     virtual scd_socket &get_sock() = 0;
5 };
```

Listing 2.3: Socket Event Handler Interface

```
1 class scd_sock_poller {
2 public:
3     bool register_handler(scd_sock_ev_handler_if &handler,
4             sock_ev events);
5     bool remove_handler(scd_sock_ev_handler_if &handler);
6     bool process();
7     void wait_process();
8 };
```

Listing 2.4: Socket Poller

OR of the socket events which have actually occurred. Sending, receiving and reacting to errors takes place within the callback function. Function `get_sock()` shall return a reference to the socket which should be monitored for the handler. It is called by the socket poller during registration. Only one socket can be used per object. If an object needs to access multiple sockets, it needs a wrapper object for every socket. Also the socket shall not change as long as the event handler is registered. Closing and reopening a socket for example requires deregistration and reregistration.

Listing 2.4 shows the public interface of the socket poller. Function `register_handler` registers an event handler. It takes as arguments a reference to the handler to be registered and the bitwise OR of the events to be monitored. A boolean is returned which indicates, if the registration succeeded. The registration fails if the handler is already registered. The function will call the handler to get the socket to watch. Member function `remove_handler` deregisters an event handler. It returns **`true`**, if the handler has been found and was deregistered successfully. The other functions described here are used by the main simulation loop to drive the communication. Function `process()` is called to check all sockets for monitored events. The handlers of events which have occurred are executed. This function returns **`true`**, if some events have been processed. It immediately returns **`false`**, if no events have occurred at all. In contrast to this `wait_process()` waits until an event occurs. It also executes the corresponding event handlers.

```
 1 void scd_simulator::start() {
 2     while ( _cont_man->active() ) {
 3         if ( _pending() ) {
 4             if ( _pending_now() )
 5                 _cont_man->set_busy();
 6             else {
 7                 sc_time step;
 8                 step = _next_time() - sc_time_stamp();
 9                 _cont_man->set_idle(step);
10             }
11         }
12         else
13             _cont_man->set_done();
14 
15         _cont_man->process();
16 
17         if ( _cont_man->busy() ) {
18             sc_start(SC_ZERO_TIME);
19             _chan_man->process();
20             _poller->process();
21         }
22         else {
23             if ( _cont_man->advance_time() )
24                 sc_start( _cont_man->get_time_step() );
25             else if ( _cont_man->active() )
26                 _poller->wait_process();
27         }
28     } // end while loop
29 }
30 
31 bool scd_simulator::_pending_now() {
32     return sc_pending_activity_at_current_time();
33 }
34 
35 bool scd_simulator::_pending() {
36     return ( _next_time() != SC_ZERO_TIME || _pending_now() );
37 }
38 
39 const sc_time scd_simulator::_next_time() {
40     return sc_get_curr_simcontext()->next_time();
41 }
```

Listing 2.5: Simulation Main Loop

2.3.5 Simulation Loop

This section describes the main simulation loop running within class `scd_simulator`. It is shown in listing 2.5. The loop is running as long as the control manager indicates to be in the active super state. This means as long as the simulation did not fail or terminate.

From lines 3 to 13 the state of the local simulation is indicated to the control manager. This has to be done every time, as the state might change by receiving data on channels. If there exist pending events and there exist events at the current simulation time, the local simulation is busy. If events at a future simulation time exist but none at the current one, the simulation is idle and currently waiting for time to be advanced. In this case the time of the next event is retrieved with `_next_time()`. From this the current simulation time is subtracted to get the relative time of the next event. This time is indicated to the control manager. The master controller will use it to globally determine the time of the next event. Therefore simulation time is only advanced to a point where events exist. This is much more efficient than always advancing time by small steps. If there exist no pending events at all, the local simulation is done.

After the local simulation state has been indicated to the control manager, the control manager is being driven. The master control manager checks the state of the slave wrapper state machines. This operation is state depending and includes for example checking, if some slaves failed or if all slaves seem to be done such that a termination attempt can be started.

If the control state machine is currently in state busy, the simulation is performed. On line 18 the simulation is run for one delta cycle. After the simulation the channel manager is driven. The simulation might have stored data in remote output channels. These channels have to be registered at the socket poller to be called back as soon as the socket can be written. If a remote input channel had been full before the simulation cycle, its socket would have been removed from being monitored for further read events. If after the simulation cycle such a channel again can accept incoming data, the read event has to be reregistered. The channel manager takes care of these event registrations. On line 20 the socket poller is driven. This induces the execution of pending transmissions in both directions. As the simulator has just been simulating, the socket poller will return immediately, if no transmissions can be made. If events are still pending, the simulation will be continued during the next iteration.

On line 23 it is checked if the control manager indicates to advance simulation time. In this case the step by which time has to be advanced is retrieved. The time step has been announced by the master controller. The call to `sc_start` then advances the simulation time. As no events exist until the new time, nothing is actually simulated here. Retrieving the time step from the control manager causes a state transition to the busy state and the simulation continues with the next iteration. If simulation time does not have to be advanced and the simulation is still active, the simulator waits for socket activity. It was not simulating before so it either has to wait for synchronization, for incoming data or for resumption of an outgoing transmission.

```
 1 #include "scd_simulator.h"
 2 #include "scd_logging.h"
 3
 4 int sc_main (int argc, char *argv[]) {
 5
 6     // ...
 7
 8     scd_set_loglevel(SCD_DEBUG);
 9
10     scd_simulator sim("host1", 5876, SCD_MASTER);
11     sim.get_cont_man().register_slave("host2");
12     sim.get_cont_man().register_slave("host3");
13
14     //scd_simulator sim("host2", 5876, SCD_SLAVE);
15     //sim.get_cont_man().set_master("10.0.0.1", 5876);
16
17     sim.get_chan_man().register_channel("C1", app_mdl.C1_ins,
18             "10.0.0.2", 5876);
19     sim.get_chan_man().register_channel("C2", app_mdl.C2_ins);
20
21     sim.init();
22     sim.start();
23
24     // ...
25 }
```

Listing 2.6: SCD Usage

The functions beginning at line number 31 separate the implementation specific interactions with the SystemC kernel to get the local simulation state. The functions used have been described in section 2.2.2. If they were changed in future releases of OSCI SystemC, changes would have to be made here or to the simulation kernel directly depending on the type of modifications.

2.3.6 Usage

This section presents how to use the SystemC distribution library. Besides the documentation in this work, it is generally recommended to read the documentation in the header files. Most interesting ones are `scd_simulator.h`, `scd_cont_man.h`, `scd_chan_man.h` and `scd_rem_chan_if.h`.

Listing 2.6 shows a typical use case. `sc_main` is the main function from where a SystemC simulation is typically initialized and started. As with a normal simulation, the application

model must first be instantiated. The difference in a distributed simulation is that processes and channels are mapped onto different simulators. Remote channels have to implement a remote channel interface and only one endpoint is instantiated on a single simulator.

Logging information is printed to standard output. If desired, the level of detail can be adjusted by calling `scd_set_loglevel`. The following hierarchical levels have been defined:

- `SCD_DEBUG` – Shows detailed information about program events. This is mostly usefull for debugging. Transitions of every state machine are displayed.
- `SCD_INFO` – This is the default level. Displays information which is usefull for monitoring or tracking configuration problems.
- `SCD_WARN` – Only errors and possible problems are displayed.
- `SCD_ERROR` – Only errors which have a serious impact on the simulation are shown.

On line number 10 the distributed simulator is instantiated. The first argument is the name of the simulator. Simulator names shall be unique. The second argument specifies the TCP port to be used for incoming connections. The third argument declares the simulator to be master or slave. Legal values are `SCD_MASTER` and `SCD_SLAVE` which are self-explaining. Only one master simulator can exist among all simulators. If the simulator is the master, an arbitrary number of slaves can be registered, as shown on line 11 for example. The argument of such a slave registration is the simulator name as it has been defined in the slave's constructor. If the simulator is a slave, the master simulator has to be set as shown on line 15. The first argument of this function is the IP address or the domain name of the master simulator and the second argument specifies the TCP port on which it is listening for incoming connections.

Remote channels must be registered at the channel manager. For every remote channel one channel endpoint has to be declared as connection initiator and the other endpoint as connection acceptor. The connection initiator will establish the TCP connection during initialization. Line 17 shows how to register a connection initiator endpoint of a remote channel. The first argument is the name of the remote channel. This name shall be unique for all remote channels and shall be identical for both channel endpoints. In the second argument `app_mdl` is the top module of the application model which has been instantiated before. `C1_ins` is the remote channel endpoint to be registered. It is of type `sc_prim_channel` and implements one or both of the remote channel interfaces. The third argument is the IP address or domain name of the simulator with the connection accepting endpoint of the remote channel. The fourth argument specifies the TCP port on which this simulator is listening for incoming connections. To register a connection accepting endpoint of a remote channel only its name and the instance are needed.

If all remote channels have been registered and the master or the slave simulators have been set, the simulator can be initialized by calling `init()`. During initialization every slave simulator establishes a connection to the master simulator and the channel manager will establish for every connection initiator endpoint of a remote channel a connection to the simulator with the accepting endpoint. Initialization of a simulator completes, if all its remote channels are

connected and the connection to the master has been established, if it is a slave simulator, or if all slave simulators have connected, if it is the master simulator.

After the simulator initialization, the simulation can be started by calling `start()`. The simulation will run either until it fails or until the simulation completes globally. Afterwards general things can be cleaned up, such as closing file descriptors. To run the simulation the executable of every simulator has to be started on its specific simulation host. No time frame has to be obeyed while running the simulators. Each simulator will retry to connect to its peers until all connections have been established.

3 Integration into DOL

This chapter describes the integration of the SystemC distribution library into DOL for the functional simulation. The existing toolchain and specification formats in XML are used and had been adopted to support the distributed functional simulation. Performing a local simulation or a distributed simulation is quite similar and the designer has only to consider a few points.

3.1 Architecture Specification

Similar to the mapping case the architecture has to be specified for the functional simulation. Here the architecture specification defines the Linux systems which perform the simulation. More specifically it defines the remote simulators. One Linux machine can run multiple simulators.

The architecture specification used for the mapping optimization models a complete hardware system. This includes processors, memories, hardware channels and their connections. For the functional simulation only processors have to be specified. For communication between the simulators the internet protocol (IP) is used. As routing is transparent, the network can be understood as a complete graph where the nodes represent the simulators. Further it is a complete directed graph where every edge is bidirectional. Scheduling is performed by the operating system and the networking devices in the path. Sockets provide a complete abstraction of underlying communication details. Because of this it is not necessary to specify channels explicitly. Specifying the network architecture or memory sizes might be necessary for an automated mapping of the application onto the remote simulators. Here the mapping is manual and such specifications therefore not necessary.

Listing 3.1 shows a sample architecture specification with one remote simulator for illustration of the semantics. The `processor` element defines the simulator. It has two mandatory attributes. Attribute `name` is the unique name of the simulator. It shall only consist of numbers and alphabetic characters. For the functional simulation attribute `type` has to be "`NETSIM`" and declares the processor to be a remote simulator. `configuration` elements are used to declare parametric data of the simulator. They are defined to have two mandatory attributes `name` and `value`.

A remote simulator is defined by its name, IP address or domain name and TCP port. The `address` and `port` configurations are therefore mandatory for this processor type. The

```
1 <architecture>
2   <processor name="sim1" type="NETSIM">
3     <configuration name="address" value="sim1.ethz.ch" />
4     <configuration name="port" value="5876" />
5     <configuration name="master" value="true" />
6     <configuration name="username" value="fabianhu" />
7     <configuration name="basedir" value="/scratch/dol/" />
8   </processor>
9 </architecture>
```

Listing 3.1: Architecture Specification

`address` configuration can contain either the IP address or the domain name of the Linux system which runs this simulator. It must be reachable by every simulator involved. The `port` configuration specifies the TCP port on which the simulator is listening for incoming connections. It is not possible to assign the same `address`/`port` pair to different simulators. However assigning the same `address` with a different `port` is totally legal. This would mean to run several simulators on the same machine. The Linux system itself is not explicitly specified.

Exactly one simulator of a simulation has to be defined as the master simulator. It controls global synchronization but this does not have a big impact for the user. The master is more important if problems occur. As it has state information of every slave standard output of the master is a good point to start debugging. The master simulator is defined with the `master` configuration. Its `value` must be exactly ”`true`”. A missing `master` configuration or other values declare the simulator as slave.

The other configurations are optional. Later a shell script is generated which allows to distribute the binaries and perform the simulation. This is done over SSH. The `username` configuration specifies the username which is used to login into the Linux system. Per default the username of the user running the script is used. The `basedir` configuration declares the base directory on the Linux system. Inside this, another directory with the name of the simulator will be created. The later one will contain the binary. Further it will contain all data generated during the simulation such as profiling data. In the example this directory will be `/scratch/dol/sim1/`. It might be a good idea to choose the base directory on a partition which provides enough free space. Profiling data can easily occupy up to 1 GB.

The DOL XML specification schema uses an `iterator` element to define regular occurrence of resources. It can only influence resource elements. For remote simulators also the `address` configuration would contain some regularity. Imagine an infrastructure of 100 computers whose domain names range from `sim1.ethz.ch` to `sim100.ethz.ch`. The `iterator` element does not support declaring such an architecture and manual work is required. But as it is likely that the simulation infrastructure does not change too often, the architecture specification has seldomly to be changed.

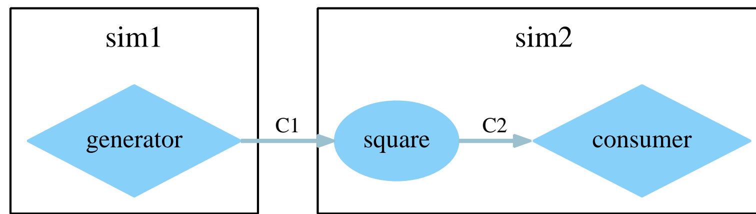


Figure 3.1: Visualization of mapping three processes onto two simulators.

3.2 Mapping Specification

A designer wants to find an optimized mapping for its application onto an MpSoC architecture. To find this mapping with DOL, the functional simulation has to be performed. Here the designer has again his application and the architecture, now the simulation architecture. Again a mapping is needed which specifies which process runs on which simulator. Optimally, this mapping distributes the workload equally among the simulators. It seems like a chicken and egg problem.

Compared to the final embedded system no deadlines have to be guaranteed and performance is more nice to have than absolutely required. The only objective is that the simulation should run in a reasonable amount of time. The mapping is therefore not required to be perfectly optimal. As the designer has knowledge about the application she can roughly estimate which processes are most computationally intense. This knowledge should be sufficient to fairly distribute the workload. If it is observed that single simulators are very busy while others are idle, the mapping can still be changed.

Even if network channels do not have to be specified for the architecture, the designer should consider the network architecture for the mapping. Processes which are connected by channels which are expected to have a high throughput, should be mapped to the same simulator or to simulators which are connected by a fast link. Mapping such processors to simulators which are connected over the internet would slow simulation down.

The mapping process for the functional simulation is currently manual work. Computer aided mapping would be helpfull but is not part of this thesis. The same XML format is used as it is already defined by DOL. Because only processors are specified for the architecture only processes have to be mapped.

3.3 Validation and Visualization

DOL can validate specifications and can generate DOT files to visualize process networks. The validator has been extended to validate the semantics of architectures and mappings for the functional simulation. Architecture specifications are checked such that every remote simulator has a unique name and a unique `address`/`port` pair. However, it is not able to detect

if different domain names map to the same IP address. Further it checks that exactly one master simulator is defined. The mapping specification is checked to map every process of the process network to exactly one processor of the architecture.

The command

```
java dol.main.Main -P pn.xml -p arch.xml \
        -m map.xml -c -D mapping.dot
```

checks a set of process network, architecture and mapping specifications and generates a DOT file. A sample visualization of a process network with three processes being mapped onto two remote simulators is shown in figure 3.1.

3.4 Code Generation

After the specification is done, it can be used to automatically generate the source code of the remote simulators with the command:

```
java dol.main.Main -P pn.xml -p arch.xml \
        -m map.xml -H systemcd
```

This command stores the source code in the directory `systemcd`. Processes running on the same simulator are connected by local FIFO channels as before. For remote channels local channel endpoints of type `scd_rem_fifo_out` and `scd_rem_fifo_in` are generated. These have been described in section 2.3.2. Local FIFO channels have a fixed capacity and write operations will block, if the channel is full. Remote channels do not have such a strict capacity limitation. Each endpoint contains a buffer with the capacity of a local channel. Further unknown capacities exists in the inbound and outbound buffers of the operating system and in intermediate networking devices. Strictly limiting the capacity of a remote channel would require read acknowledgements which would degrade the simulation speed. The FIFO size of a channel as specified in the process network only provides a lower bound for remote channels.

The remote channel endpoints are automatically registered at the channel manager with the information from the architecture specification. Also the control structure is set up according to this information. A Makefile is generated which allows to compile all the simulator binaries. The binary for a simulator with name "`sim1`" will be called "`scd_sim1`".

Per default the simulators will only display informational messages. If problems have to be analyzed in detail, debug level has to be enabled. By using option `--debug` for code generation, this can be easily enabled for all simulators without having to modify every simulator's main source file.

3.5 Running the Simulation

Together with the source code a shell script has been generated which allows to easily distribute the binaries to the simulation hosts and run the simulation. The script is called `scd.sh` and is stored in the same directory as the source code. It must be set executable before being run. As paths to the binaries have been hard coded, it can only be run on the system which compiled the binaries. Because of library issues this system must have the same software configuration as the simulation hosts.

To copy files and execute commands on the simulation hosts SSH is used. Per default the username of the user running the script is used to login into the simulation hosts. Another username can be specified as described in section 3.1. Key based authentication should be used to prevent interruptions for entering passwords. For this to work a user identity key must be present in the user account of the user running the script. This key must be stored in the list of authorized keys on the simulation hosts. The computing infrastructure at ETH is set up accordingly. For more information refer to the SSH documentation. The system is fail-safe. If key based authentication fails the user is prompted for a password. This is done for every interaction which would reduce usability of the script.

Following the commands supported by the script are explained:

- `scd.sh distribute`
  Creates the simulation directories on the simulation hosts. The simulation directory has the name of the simulator. If a base directory has been specified as described in section 3.1, it will reside in there. Else the simulation directory is created in the user's home directory. The simulator binary is then copied into the simulation directory for every simulator. After having issued this command the user will have to copy input data to the simulation directory of the simulators which requires it.
- `scd.sh run`
  Runs the simulation. Standard output of the master simulator is displayed. Standard output of every simulator is written to a text file in the current working directory. For a simulator with name "`sim1`" the file is called `stdout_sim1.txt`. These text files could for example be ongoingly displayed during simulation with `tail -f`. They also serve to check the simulation result or to analyze problems. Output data such as profiling data or application specific output is stored in the simulation directories.
- `scd.sh check`
  Displays which remote simulators are currently running. If errors occur all simulators will usually terminate. But in case the connection to the master could not have been established yet or the simulation hangs for some reason, this command is helpfull.
- `scd.sh kill`
  Kills all simulation processes. This is especially usefull in combination with the `check` command.

- `scd.sh profile pn.xml`
  Collects profiling data and analyzes it. Every simulator generates a profile during simulation. This command iteratively copies the profile to the current working directory and uses the DOL profiler to summarize and then annotate the result into the process network specification. For this to work the classpath to DOL has to be set in the `CLASSPATH` environmental variable. The output will be a new XML file annotated with the profiling result. In this example this file will be called `pn_annotated.xml`.
- `scd.sh cleanup`
  Removes the simulation directories from the simulation hosts. This command is typically issued after a simulation and after the profiling.
- `scd.sh help` or `scd.sh -h`
  Prints a help page which shortly explains these commands.

3.6 Profiling

Functional simulation of a complex application will generate a big amount of profiling data. Analyzing this data is as computationally intensive as running the simulation. Currently profiling takes place on the host which compiles the code and starts the simulation. Of course this does not scale. However, it is easily possible to perform the profiling on the simulation hosts and then merge the annotated process network specifications.

4 Results

This chapter analyzes the achievements of the SystemC distribution library for the functional simulation on the base of an application example.

4.1 Methodology

To analyze the performance of a distributed functional simulation in DOL, an MPEG-2 video decoder [12] is used as an example application. This decoder operates in parallel, decodes the video frames from a compressed video sequence and saves them as picture files. Its process network is shown in figure 4.1. The measurements are done for decoding a video sequence of 15 s length. It has a resolution of 704 x 576 pixels and a frame rate of 25 fps. No profiling data is collected during the measurements.

The simulation hosts have an identical hardware and software configuration. Each machine has two AMD Opteron 2218 processors with 2.6 GHz. This is a dual core processor. Each machine has therefore four cores in total. Linux is used as operation system. The kernel has version 2.6.23 and runs in 64 bit mode. The simulator binaries run in 32 bit hardware compatibility mode. The systems have been idle during the time of the measurements, so the influence of other processes running on these machines is neglected. The simulation hosts are connected by Gigabit Ethernet. Round trip times measured with `ping` were below 0.1 ms.

To measure the execution of the simulation, the shell command `time` of the Berkeley UNIX C shell (CSH) is used. It reports the real, user and system time of an execution. Real time corresponds to the time from the start of the execution until it completes. This time would be measured with a normal stop watch. It gives information about how fast an execution is. User and system time is the time during which the process has been running in user or kernel mode, respectively. The sum of user and system time is called computation time. It indicates how computationally intense an execution of a program is or how long the application has been using the CPU. All times are rounded to seconds.

4.2 Measurements and Discussion

Table 4.1 shows the runtime of several simulations. The different simulation cases and their differences are discussed in this section.

Case	Simulation	Hosts	Sims	Real	Comp.
1	Pthread	1	1	31 min 46 s	80 min 47 s
2	SystemC	1	1	12 min 13 s	12 min 13 s
3	SCD	1	1	12 min 14 s	12 min 13 s
4	SCD	1	4	5 min 21 s	12 min 34 s
5	SCD	4	1	5 min 23 s	12 min 42 s
6	SCD	1	20	4 min 19 s	12 min 48 s
7	SCD	5	4	2 min 41 s	13 min 24 s
8	SCD Timed	1	4	10 h 27 min 14 s	–

Table 4.1: Runtime of the MPEG example application. Hosts denote the number of simulation hosts involved. Sims denotes the number of simulators per simulation host.

DOL provides different implementations for the functional simulation. SystemC has already been mentioned before. Another implementation is based on Posix Threads (Pthreads). This implementation has been measured in case 1. Posix Threads are in contrast to the SystemC threads preemptive. While running the simulation monitoring utility `top` indicates that all available CPU cores are in use. This is the reason why the computation time is higher than the real time. Using all cores is in principle a good thing. However the Pthread implementation is very inefficient compared to the other simulations. Running almost 30 min and computing more than 80 min is one of the worst results. This huge difference is mainly caused by synchronization overhead which is necessary because of preemption. Another reason is that Pthreads bases massively on system calls [13] which cause a context switch overhead. This is also indicated by spending 62 % of the computation time in kernel space.

The existing implementation with SystemC is much faster as shown in case 2. This is caused by the low switching and synchronization overhead of the non-preemptive threads. But the OSCI SystemC implementation of the non-preemptive scheduling has also the effect, that only one CPU core can be used. This is also confirmed by `top` and the equality of the real and computation time. As the simulation host has four cores available, only using one of it is a waste of resources. This simulation has mainly been used in DOL until now and serves as the main reference point.

To be able to directly compare the SystemC simulation with the new distributed simulation, only one remote simulator is running the simulation in case 3. The difference between these implementations is that the distributed simulation runs delta cycle wise as presented in section 2.2.2 whereas the existing simulation is just started with `sc_start()`. The result shows that by running the simulation delta cycle wise only a very small computational overhead of 100 ms is introduced. This is not even explicit in table 4.1 as times are rounded to seconds. The difference in real time is about 0.5 s. This can be explained by the overhead of logging into the simulation host by SSH.

In a next step the application is mapped onto four simulators as shown in figure 4.1. Case 4 runs all simulators on the same simulation host which allows the use of all four CPU cores. The simulation speed is more than doubled. But why is the speedup not by factor 4? Having

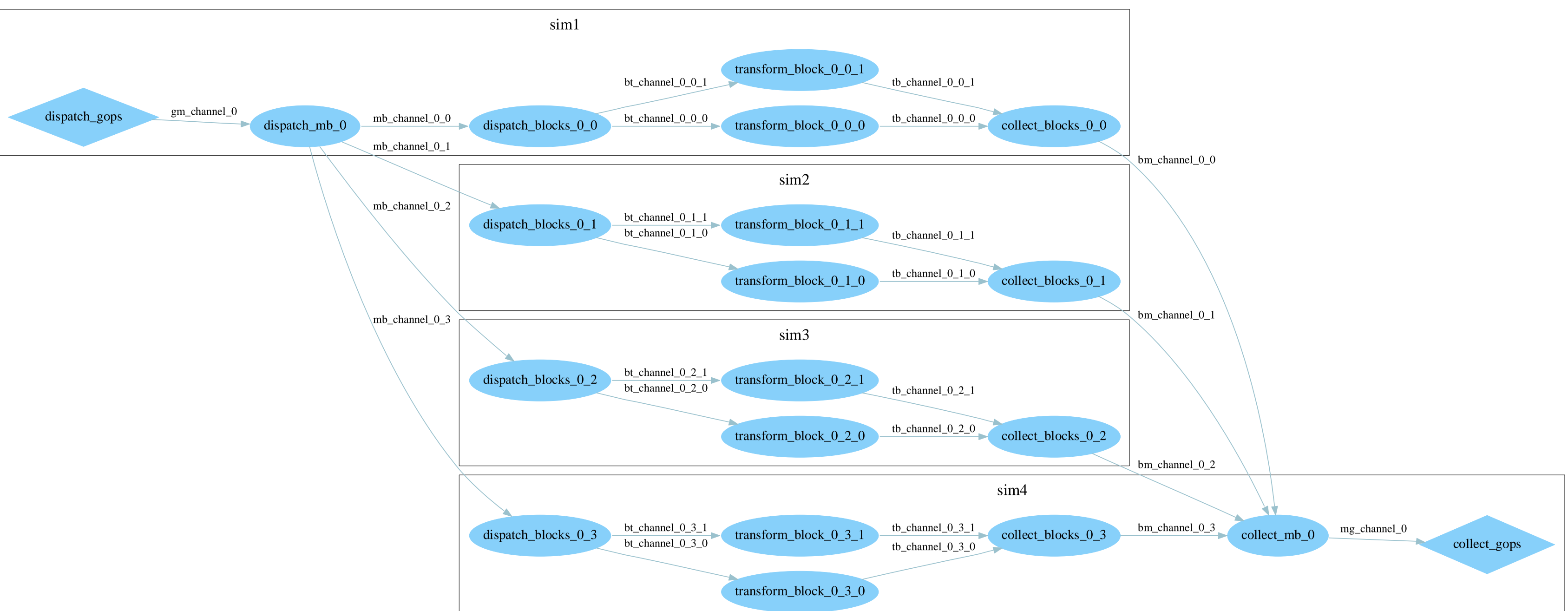


Figure 4.1: Mapping of the MPEG-example onto four remote simulators.

Simulator	Comp.	Usage
sim1	2 min 58 s	55 %
sim2	2 min 09 s	40 %
sim3	2 min 09 s	40 %
sim4	5 min 17 s	99 %

Table 4.2: Computation time of single simulators in case 4.

a look at table 4.2 gives the answer to this question. It shows how long each of the four simulators has been computing. Further it shows the CPU usage for every simulator. This means the ratio between a simulator's computation time and the real time of the simulation. Simulator *sim4* has to compute for more than five minutes while the others are computing only about half of this time. Because the simulator can not use another CPU core in parallel this poses a lower bound on the simulation runtime. The root of this differences is a non-optimal mapping. Processes *dispatch_blocks*, *transform_block* and *collect_blocks* are equally mapped to every simulator. Additionally simulator *sim1* has to run processes *dispatch_gops* and *dispatch_mb* and simulator *sim4* processes *collect_mb* and *collect_gops*. One or both of *sim4*'s additional processes must be computationally very intense which causes the bigger load of the simulator. Such non-optimal mappings can not be avoided, because the partitioning is done by the designer and bases on her experience and estimation of the process complexity. By using multiple simulators a computational overhead of 21 s has been introduced. This is caused by communication between the simulators and the controlling which both need additional cycles. Communication within a simulator is much cheaper.

Case 5 uses the same mapping onto four simulators, but runs them on distinct simulation hosts which makes network communication necessary. Compared to case 4 which runs all simulators on one simulation host, the simulation is only 2 s slower. This is very impressive, even if a very fast network has been used. Profiling showed that almost 3 GB of data has been transferred over remote channels. This result is possible because the MPEG decoder is stream oriented. A part of the sequence is processed and the result is forwarded immediately. Therefore communication takes place all the time. An application which processed the input sequence as a whole, would not only be unoptimal for parallelization but would also have a much bigger delay caused by communication. Besides the real delay of 2 s a computation overhead of 8 s is caused by network communication. A closer look shows that this additional computation is done in kernel space. Reading data from the network interface is slower than only reading from a logical interface as this is the case for intra-host communication.

To circumvent the effects of a non-optimal mapping as discussed in case 4, a maximal mapping is used in case 6. With maximal mapping is meant to have a simulator for each process. Like this each simulator has the smallest possible load. To remember the simulation time is lower bounded by the most loaded simulator. Of course simulation will take long if a process is computationally very intense. But as this is the only process on its simulator, other processes can be executed on other CPU cores in the mean time. Table 4.1 shows that simulation time could be improved by more than 1 min. Compared to case 4 computation time has increased

Process	Comp.
dispatch_gops	1.8 s
dispatch_mb	45.2 s
dispatch_blocks	34.6 s
transform_block	25.2 s
collect_blocks	49.9 s
collect_mb	147.0 s
collect_gops	35.2 s

Table 4.3: Average computation time in case 6 broken down by process.

by 14 s. Again more synchronization and more inter-simulator communication is necessary as there are more simulators and now every channel is a remote channel. To get the overall CPU usage, computation time is divided by the maximum computation time which would have been available during the simulation. This is computation time divided by 4 divided by the real time. This gives a CPU usage of only 74 %. It would be expected to be almost 100 %. The reason is that the delay for (Linux) process switches is not measured by the `time` command. Using a simulator for each process is in fact using preemptive scheduling. The operating system decides when to run which simulator and for how long. If processes only perform short computations the overhead of switching simulators becomes essential.

Another interesting aspect of the maximal mapping is that the computation time of each simulator correlates with the computational intensity of the process it executes. Table 4.3 shows the computation time of simulators which run a process of a certain type. For processes which occur multiple times in the process structure the average computation time is shown. This information could be used as a rough runtime estimation for the mapping process. Of course the execution times are only meaningfull as relative values. These results confirm the assumption that either process *collect_mb* or *collect_gops* must be computationally intense. This was the reason why simulator *sim4* had a much higher load than the other simulators in case 4.

In case 6 the simulation has been started on the simulation host directly without using the shell script. It turned out that it is not possible to login to a Linux system by SSH many times within a very short time frame. Starting many simulators on a single simulation host would require to delay the script between each login.

Case 6 runs all simulators on one simulation host. Therefore they have to share 4 CPU cores. Case 7 distributes the 20 processes among 5 simulation hosts. Each simulation host runs the same number of simulators as it has CPU cores. This results in the maximum possible speedup. The simulation runs now in 2 min 41 s. This time is lower bounded by sequential dependencies. Compared to the SystemC implementation the speedup is 4.5. Because more network communication takes place than in case 5, an additional computation overhead is the consequence. Compared to the SystemC implementation the computational overhead is 10 %. Distribution allows therefore to decrease the simulation time but uses slightly more computational resources.

With the distributed simulation communication takes place after every delta cycle. The implementation in DOL fires every process at most one time per delta cycle. Case 8 replaces the delta cycle with a time step of one nanosecond. This makes global synchronization necessary after every simulation step. The result is disappointing. Simulation runtime is over 10 hours. Simulation time at the end is 594002 ns. If actual computation is neglected, the synchronization for one time step takes 63 ms. 60 ms of this time is spent for checking, if a transmission is in progress. This delay is necessary to get a stable synchronization, but makes the distribution library unsuited for simulations with very few computation per time step, as this was the case in this example or might be the case for a simulation on register transfer level. For a simulation on a higher level of abstraction the synchronization delay is negligible. Computation time of this simulation was below one minute. This indicates that the `time` command is not accurate for an execution where the process runs only for a very short time in sequence.

5 Conclusions and Future Work

5.1 Conclusions

The results have shown that the SystemC distribution library is well suited for functional simulations. It allows the distribution of the simulation workload among several simulation hosts which reduces the simulation time. The library can also be used for timed simulations which perform a lot of computation per time step. This is the case for simulations of approximate-timed transaction-level models (TLM) [14]. Because the synchronization overhead is huge, the library is not suited for simulations on lower levels of abstraction such as the RTL. Basically it is possible to connect SCD simulators with other cosimulators, if they support the same protocol and apply proper data type conversion.

The distribution library keeps the performance advantages of non-preemptive scheduling, but allows the usage of all computational resources available. This is particularly important as modern computer systems have multiple CPU cores which SystemC by itself is not able to use. The ability to include an arbitrary number of systems in a simulation enables scalability which is the core objective of the SHAPES project.

Even if the functional simulation can be parallelized, it can not be accelerated arbitrarily. The design of the application and the mapping have a big impact on the simulation speed. Therefore the designer and her experience play a central role. To get the best performance she should obey the following rules:

- Expose the parallelism of the application as good as possible, but not too fine grained as this would increase the complexity.
- Have a detailed knowledge about the application and be able to roughly estimate the performance requirements.
- Map processes to simulators such that all of them have approximately the same load.
- Avoid remote channels with high throughput, especially if slow links are used. Better run two processes connected by heavily used channels on the same simulator.
- Run at least one simulator per CPU core. The more simulators per simulation host, the better all cores can be used, even if the mapping is not optimal.
- Do not run too many simulators per simulation host as this would cause process switching overhead.

In principle any simulation with SystemC can be distributed. However the distribution library has some minor drawbacks:

- It is implementation specific and requires OSCI SystemC 2.2.0 or later. Minor changes to the simulation kernel would become necessary, if important functions were removed in future releases.
- Delta cycles are not synchronized globally. This is the reason why no changes to the simulation kernel have been necessary until now without loosing performance for untimed simulations. As delta cycles are a construct which can not be observed in the real world, this should be acceptable for most simulations.
- IEEE 1666 requires that the order of process execution shall not vary from run to run as long as the same implementation, application and input data is used. This is violated because process activation depends on data delivery over the network which is not deterministic. As an implementation is free to chose a deterministic way how to schedule processes and processes shall only communicate through channels and implement proper synchronization themselves, this violation is acceptable.

5.2 Future Work

This thesis solves the problem of the distributed functional simulation in DOL and provides a prototype of a library to distribute arbitrary simulations on higher levels of abstraction. However, it was not possible to address all related aspects and still a lot of future work is possible.

The SystemC distribution library can be further analyzed and improved. This is not essential for DOL, but might be interesting for other applications. It includes the following tasks:

- Try to improve the synchronization algorithm. Right now it makes the library unsuitable for cycle accurate simulations which perform only low computation per time steps.
- Find ways to avoid global synchronization. For example globally synchronize time only at certain points, whereas time is advanced without synchronization within single modules.
- Get more experience with timed simulations which compute a lot per time step as this is the case for TLM simulations. Analyze how big the impact of the slow synchronization is on such simulations.
- Implement more complex channels to extend the library.
- Introduce some form of authentication to prevent denial of service attacks and theft of intellectual property.
- Analyze the effect of slower networks.

Also DOL can be improved:

- Develop a tool such as a graphical user interface which supports the designer in creating the mapping for the functional simulation.
- Implement a FIFO channel with a strictly limited maximum capacity and analyze it regarding its performance drawbacks.
- Enable distributed profiling.
- Improve the DOL interface which seems to be a little overloaded at the time.

Bibliography

[1] Scalable Software Hardware Architecture Platform for Embedded Systems (SHAPES). Website. `http://www.shapes-p.org`.

[2] Pier S. Paolucci, Ahmed A. Jerraya, Rainer Leupers, Lothar Thiele, and Piero Vicini. Shapes: A tiled scalable software hardware architecture platform for embedded systems. In *CODES+ISSS '06: Proceedings of the 4th International Conference on Hardware/-Software Codesign and System Synthesis*, pages 167–172, 2006.

[3] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *ACSD '07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, pages 29–40, 2007.

[4] Giles Kahn. The semantics of a simple language for parallel programming. In *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. 1974.

[5] Alessandro Fin, Franco Fummi, and Denis Signoretto. The use of SystemC for design verification and integration test of IP-cores. In *Proceedings of the 14th Annual IEEE International ASIC/SoC Conference*, pages 76–80, September 2001.

[6] Mario Trams. Conservative distributed discrete event simulation with SystemC using explicit lookahead. `http://digital-force.net/publications/`, 2004.

[7] Samy Meftali, Anouar Dziri, Luc Charest, Philippe Marquet, and Jean-Luc Dekeyser. SOAP based distributed simulation environment for System-on-Chip (SoC) design. In *Forum on Specification and Design Languages, FDL'05*, December 2005.

[8] David Richard Cox. RITSim: Distributed SystemC simulation. Master's thesis, Rochester Institute of Technology, Rochester, NY, USA, August 2005.

[9] IEEE Computer Society. IEEE 1666 Standard SystemC Language Reference Manual. `http://standards.ieee.org/getieee/1666/index.html`, December 2005.

[10] Open SystemC Initiative (OSCI). Website. `http://www.systemc.org`.

[11] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *First Pattern Languages of Programs Conference*, August 1994.

[12] Simon Mall. MPEG-2 decoder for SHAPES DOL. Master's thesis, Federal Institute of Technology (ETH), Zurich, Switzerland, April 2007.

[13] Ulrich Drepper and Ingo Molnar. The native POSIX thread library for Linux, February 2005.

[14] Lukai Cai and Daniel Gajski. Transaction level modeling: An overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 19–24, 2003.

A CD-ROM Content

The following list describes the content of the CD-ROM. Archives contain a file `README.txt` which further explains their content.

`doc/thesis.pdf`	Thesis PDF version
`doc/thesis.ps`	Thesis PostScript version
`doc/thesis.zip`	Thesis LaTeXsource
`doc/presentation.pdf`	Thesis final presentation PDF version
`doc/presentation.odp`	Thesis final presentation OpenOffice source
`scd.zip`	SystemC distribution library
`example1_scd.zip`	Example usage of the SystemC distribution library based on DOL example 1
`dol_ethz.zip`	DOL with integrated distributed functional simulation
`example9_scd.zip`	MPEG-2 decoder DOL application example 9 with example architecture and mapping specifications
`systemc-2.2.0.tgz`	OSCI SystemC 2.2.0

B Step-by-Step Instructions

B.1 Preparing DOL

1. Download OSCI SystemC version 2.2.0 or later from `www.systemc.org` and compile it according to its documentation.
2. Install Apache Ant version 1.65 or later.
3. Unzip DOL from file `dol_ethz.zip` to an arbitrary directory referred to as the DOL base directory.
4. Edit file `minibuild.xml` in the DOL base directory and adopt the paths to your SystemC installation (properties `systemc.inc` and `systemc.lib`).
5. Change the current working directory to the DOL base directory.
6. Configure DOL:

   ```
   ant -f minibuild.xml config
   ```

7. Compile DOL:

   ```
   ant -f minibuild.xml compile
   ```

8. Set the `CLASSPATH` environmental variable to contain the DOL specific paths. For BASH:

   ```
   DOLPATH=<dol base directory>
   CLASSPATH=$DOLPATH/jars/jdom.jar:\
       $DOLPATH/jars/xercesImpl.jar:\
       $DOLPATH/build/bin/main:.
   export CLASSPATH
   ```

B.2 Running the MPEG-Example

1. Unzip the MPEG example from file `example9_scd.zip` to an arbitrary directory and use this as the current working directory.
2. Configure the structure of the application by setting the number of GOP, macroblock and block ports in `example9.xml` and `src/global.h`. For details refer to Mall [12]. The default instantiation is 1 GOP, 4 macroblock and 2 block ports.

3. Flatten the process network:

```
java dol.helper.flattener.XMLFlattener example9.xml DoFlat
javac DoFlat.java
java DoFlat > example9_flattened.xml
```

4. Create an architecture specification describing your simulation environment or use a predefined one. `arch_localhost.xml` for example defines 20 simulators running on localhost.
5. Create a mapping of the application onto the remote simulators. If the application structure has not been changed in step 2 and a predefined architecture specification is used, also a predefined mapping can be used. `map_4.xml` for example maps the application to four remote simulators as shown in figure 4.1.
6. Validate the specifications and generate DOT files to visualize the process network and the mapping:

```
java dol.main.Main -P example9_flattened.xml -c -D pn.dot
java dol.main.Main -P example9_flattened.xml \
    -p arch_localhost.xml -m map_4.xml -c -D map.dot
```

7. Display the DOT files:

```
dotty pn.dot
dotty map.dot
```

8. Generate the source code of the remote simulators:

```
java dol.main.Main -P example9_flattened.xml \
    -p arch_localhost.xml -m map_4.xml -c -H systemcd
```

9. Compile the code:

```
cd systemcd/src
make
cd ../..
```

10. Make the shell script executable:

```
chmod a+x systemcd/src/scd.sh
```

11. Make sure the simulation hosts run an SSH daemon and the user's identity is authorized to login. The simulation hosts should have the same software configuration as the host on which the code has been compiled.
12. Distribute the simulator binaries to the simulation hosts:

```
systemcd/src/scd.sh distribute
```

13. Copy the test video sequence `test.m2v` to the simulation directory of the simulator which runs process *dispatch_gops*. With the default configuration described here this is directory `~/sim1/` on localhost.

14. Run the simulation:

    ```
    systemcd/src/scd.sh run
    ```

15. Follow how the images are written by process *collect_gops* while the simulation is running. In a new shell:

    ```
    tail -f stdout_sim4.txt
    ```

16. Collect and analyze the profiling data:

    ```
    systemcd/src/scd.sh profile example9_flattened.xml
    ```

17. Remove the simulation directories from the simulation hosts:

    ```
    systemcd/src/scd.sh cleanup
    ```