Digital Force White Paper

Date: 14th February 2004
(Last Build: 14th February 2004)

Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead

Mario Trams
Mario.Trams@digital-force.net

Digital Force / Mario Trams
http://www.digital-force.net

Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead

Mario Trams

Mario.Trams@digital-force.net

Abstract

This paper deals with the distributed simulation of RTL–like SystemC models. It does not just contain a presentation of a first small library that makes such a distributed simulation possible. It contains as well various design decisions that have been made in order to meet the primary goals. Among other things, these goals were the use of SystemC as it is — i.e. the reference implementation without any changes/additions to the language definition. Also, a high degree of transparency is desired so that the model programmer is not concerned with synchronization details.
The solution demonstrated herein is not completely transparent — primarily caused by restrictions due to the current SystemC implementation. However, besides a few extra–information that needs to be specified during the elaboration phase it can be considered transparent.

Keywords

SystemC, Conservative Distributed Discrete Event Simulation, RTL Simulation

1 Background

For a special project there was a demand to distribute a more or less complex simulation model across multiple computers. An obvious reason for such a distribution is speed, of course. Another reason is the capability to join different kinds of simulation kernels running different partitions of the whole design to be simulated.

The model to be simulated is not just a piece of hardware. Rather it is a mixture of concurrent software behavioral models and dynamic physical processes. As these physical processes are to be simulated based on discrete time steps, SystemC appears to be a good framework to do this.

According official SystemC documentation ([5], page 9) interfacing SystemC with other simulation kernels — and hence as well with instances of itself — is on the road map. However, it is not clear when and if at all this will become part of the SystemC standard and when it will be implemented in the reference implementation.

So the decision has been made to specify and develop some kind of a small plug–in (a C++ class library, in fact) that can be used in junction with the SystemC reference implementation. It should also work in junction with other SystemC implementations, but that has not been tested yet.

The advantage of this approach is that this library does not require any changes of the actual SystemC implementation. This is a crucial point as it does not bind the functionality to a certain version of a SystemC kernel.

Of course, this rather loose integration into SystemC implies various inefficiencies. However, so far the primary goal was to get a running library at all.

It is not a goal of this project to provide a completely transparent solution that can take arbitrary SystemC code and distribute it across multiple simulation hosts.

2 Distributed DES Review

There have been written lots of papers and text books that deal with distributed discrete event simulations (also known as distributed DES, sometimes called parallel DES). A comprehensive summary can be found in [1] or [2]. Therefore just a short review of some basics is given here for completeness.

In a distributed DES, partitions or so–called *logical processes* (LPs) of one simulation model are being simulated on various computers in parallel. All LPs are connected logically by channels.

Figure 1 shows an exemplary configuration with three logical processes that are represented by basically independent simulation kernels.

LP2 generates a signal `C` that is evaluated by LP3. LP3 also receives signals `A` and `B` from LP1. In turn, LP1 receives the signal `D` which is generated by LP3.

An LP usually sends a signal notification to other LP(s) when the signal has been altered locally. Besides other information, such a notification contains the new signal value and some time information.

The setup shown by figure 1 is a non–trivial setup as it contains a loop. This complicates things sub-

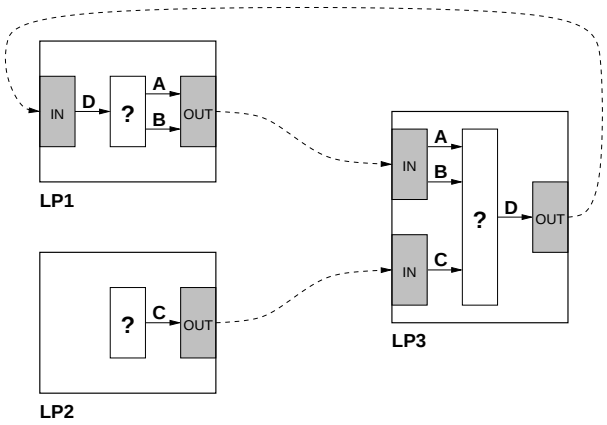

Fig. 1: Exemplary Distributed System

stantially, but in most simulation models loops have a natural occurrence and therefore it is important to deal with them.

When looking at LP2, it has a rather simple job. It can generate new values for `D` and can send out notifications as needed. This is because LP2 does not receive any signal notifications from other LPs. But LP1 cannot continue simulating over a simulation time that has not yet been reached by LP3. The same applies to LP3 which has to wait for both LP1 and LP2.

Two basic mechanisms are known that deal with this situation. The so–called *optimistic* and the *conservative* distributed DES.

2.1 Optimistic Simulation

In an optimistic simulation fashion, an LP that awaits signal notifications from other LPs just continues its simulation based on notifications that have been received some time in history. Signal notifications for other LPs are generated as usual. For the case that an LP receives a signal notification telling the signal has been changed in past, the LP has to react appropriately. The problem is that the LP has advanced the simulation based on an obviously outdated signal value. (In the special case that the value of the signal has not changed, there is nothing to do.)

An obvious task to do by an LP that has received a signal notification for the past is to turn back the simulation time. This is also known as *time warping*. What is required to accomplish that is a log of **all** changes of local signals. Only in this case it will be possible to restore the old state.

Besides this it is required to cancel all signal notifications that have been sent out to other LPs meanwhile.

2.2 Conservative Simulation

In contrast to the optimistic simulation policy where inconsistent states are allowed in hope that they do not appear too often, they are avoided from the beginning in case of conservative simulation. That is, an LP continues simulation at a certain time only when it can be sure that no notifications with time stamps in past will arrive.

This is accomplished by sending out so–called *nullmessages* containing just a time stamp. By sending such a time stamp the sending LP signals that it will not generate regular signal notifications with smaller time stamps. The information for these nullmessages is determined by a mechanism called *lookahead*. That is, the LP needs to look ahead in order to determine when the next signal will change. This is not trivial and it is especially problematic in simulation models with zero propagation delays. As a result, such models cannot be simulated with conservative simulation.

3 Conservative or Optimistic Distributed SystemC?

Now there's the question: What to use for distributed SystemC simulation?

Answering this question was not very difficult, as it appears to be very problematic to take the optimistic approach. The primary concern is the logging of local signals of an LP that is required for time warping. In order to restore an old state it is necessary to track **every** signal in the design. As such a functionality is not provided inherently by SystemC it could only be achieved by attaching every signal in the design manually to some tracking modules. This is definitively not desired as it uglifies the actual design and leads to errors (when one forgets to attach a signal). The only acceptable way would be to have some kind of wrapper for SystemC signal declarations (`sc_signal<>`) that automatically registers signals when they are declared. Whether this is applicable with SystemC has not been evaluated yet. However, one drawback that remains is the overhead that is caused by the logging. Only a single signal that is feed into an LP might require the logging of many many local signals and other state–information.

As an additional difficulty, SystemC does not allow to set the simulation time to arbitrary values. Perhaps one can work around this problem with some dirty trick. But that has also not been evaluated so far.

Apart from these issues there are other concerns regarding optimistic simulation. Memory requirements might grow significantly due to the signal history that needs to be kept for some time. The optimistic simulation policy is also subjected to a higher degree of non–determinism regarding runtime. This is because of the likelihood of extensive and recursive rollbacks that might or might not occur depending on some runtime behavior. That effect is a little

bit problematic when it turns into real time simulations. However, there are several promising mechanisms (also reviewed in [1]) that can lead to improvements of these problems. One rather simple thing is the introduction of so–called *optimistic time windows* that avoid optimistic simulation "too far" into future. This improves the realtime situation as the simulation becomes more deterministic. In addition, it reduces the memory requirements because the history logs can be kept comparatively short rather than potentially infinite.

A mixed approach of both conservative and optimistic distributed simulation has been shown in [3] and [4] in form of a distributed VHDL simulator. This appears to be very promising as well. Though, it seems to be very complicated to be implemented for SystemC without modifying it.

The conservative policy is much more easy to handle in junction with SystemC. First of all there's no need for signal history logs which simplifies the situation greatly. Also it is rather simple to suspend the simulation progress for a certain time and there is no need to set back the simulation time.

Additionally, from a practical point of view the conservative distribution strategy fits the targeted application class very well. There is no simulation down to the delta cycle level required and the states of all signals change on regular time bases.

Hence, the conservative approach has been chosen as underlying distributed simulation paradigm.

4 Availability of SystemC for such Acrobatics

SystemC is not something like a hardware description language such as VHDL or Verilog. Instead, it provides some functionality that is represented by a C++ class library. Simulation models (more correctly: system descriptions) are written in plain C/C++ that makes use of SystemC primitives. If the model is intended for simulation, it can be compiled with an ordinary compiler, linked against the SystemC library, and executed on a host computer.

Due to the use of an ordinary language, an ordinary compiler, and an ordinary simulation host, one can use ordinary libraries and system calls within a SystemC model. Thus, a SystemC process (meaning an executable) can also communicate with other processes — via TCP/IP or MPI, for instance. That makes SystemC a very powerful tool compared with other description languages.

To do the same thing with another HDL, perhaps VHDL, it would be required to develop a new simulator core or at least to change the existing conventional simulator core. Digging into the VHDL–to–C conversion that is also done sometimes is problematic as well.

Another important aspect of SystemC is the cooperative nature of parallelism. Threads within a simulation model are not scheduled preemptively. That is, once code of a SystemC thread is being executed, this thread can't be suspended except by the thread itself. A SystemC thread can therefore stop and continue the simulation as needed. This is exactly what is needed in order to implement the required functionality. And, of course, it can be implemented without modifying the SystemC simulation kernel itself.

5 Terminology

Before discussing more concrete things, here a few words about the used terminology.

- *outbound signal* —
  A local signal that is feed to some remote simulation kernel (via an outbound synchronization module) is denoted as outbound signal.
- *inbound signal* —
  A local signal that is updated by some remote simulation kernel (via an inbound synchronization module) is denoted as inbound signal.
- *outbound synchronization* —
  Outbound synchronization refers to the general process of informing remote simulation kernels about the state of local outbound signals.
- *outbound synchronization module* —
  This is a logical construct of a module that is responsible for performing outbound synchronization of a number of outbound signals. An outbound synchronization module (or in short outbound sync module) is connected with a remote inbound synchronization module.
- *inbound synchronization* —
  Inbound synchronization refers to the general process of handling incoming signal notifications and controlling the local simulation process.
- *inbound synchronization module* —
  dito ...
- *signal notification* —
  Signal notifications are sent out by outbound synchronization modules and are received by inbound synchronization modules. A signal notification contains primarily
  - a *signal designator* that designates the signal unique within the associated outbound/inbound module context.
  - a *signal type* that specifies the type of the signal (`double`, `int`, ...).
  - a *signal value* that represents the new signal value.
  - a *signal period* that states how long the new signal value is valid (lookahead–information; see next sections).

6 More Conceptual Issues

What is required to realize the needed functionality? The following subsections describe some rather general issues while the remainder of the paper discusses some technical details of this implementation.

6.1 Explicit Lookahead

It has been described that the conservative policy has been chosen. This incorporates another problem: The lookahead. It needs to be determined how long remote simulation kernels can safely continue their simulation. To cope with that problem a mechanism has been used that actually does not require the lookahead at all. Even more, it does not require the sending of nullmessages. This has been designated as *explicit lookahead* here.

The idea is as following:
The targeted class of simulation models has a very regular behavior. That is, signals will change their states in a very predictable fashion with a certain frequency. This update frequency is known at the latest in the beginning of the simulation.

So from the practical point of view it is not necessarily senseful to spent lots of efforts in a transparent solution that determines the behavior at run–time. The result of a first investigation was also that this appears to be impossible to achieve with SystemC as it is.

The solution that has been chosen finally was to let the programmer of the simulation model specify the update rate of each individual outbound signal. As a positive side–effect, this eliminates the need for nullmessages as well.

The synchronization system described herein does not send out nullmessages with safe time stamps and signal notifications containing new signal values tagged with according time stamps. Instead, it sends out signal notifications containing new signal values tagged with time stamps containing an information when the signal will change in future. It is not necessary to include information about the current simulation time where the signal actually changed. This information is inherently known to the remote inbound sync module.

Of course, the lookahead information that is being explicitly specified needs to match exactly the actual behavior. In the current implementation, wrongly specified information cannot be discovered during runtime and therefore leads to wrong and/or nondeterministic behavior of the simulation run without any notice.

6.2 Event– or Time–Driven Outbound Synchronization?

This is another important basic question that remains to be discussed. Changed values of outbound signals need to be detected or caught by outbound sync modules in order to feed them to remote simulation kernels. This mechanism can be either event– or time–driven. Event–driven means the outbound synchronization is initiated when the corresponding signal has just changed. In contrast, in a time–driven outbound synchronization the corresponding outbound signal is being read regularly — regardless of the actual signal behavior.

Some first evaluation prototypes of the distributed SystemC library used the event–driven approach. The basic concept is that outbound sync modules in the form of SystemC threads are made sensitive for changes of the corresponding signals. However, there appeared several inconveniences:

- It is problematic (or impossible) to set up SystemC threads that are sensitive to a variable amount of signals.
  The only work–around is to use one thread per outbound signal. However, that appears to create unnecessary overhead in case of large amounts of outbound signals.
- Outbound signals need to be from the class `sc_buffer` instead of `sc_signal`. This is because value–updates of `sc_buffer` generate an event even when the new value equals to old one. That's not the case for `sc_signal`.
- Outbound signals **need** to be updated by the application **exactly** at the same frequency that has been specified for it (see also section 7.4). So even when it is obvious for the application that a signal will not change in the next couple of thousand cycles — no way.

Time–driven outbound synchronization does not show up these problems. Though there are some other issues making it a little bit more complicated to implement.

Nevertheless, event–driven outbound synchronization seems to fit perfectly for optimistic distributed simulation.

6.3 Distributed Simulation Semantics

The time–driven outbound synchronization rises up another problem that requires a more clean definition of the behavior of application threads. The issue of interest is the relation between the exact moment of outbound synchronization of a certain outbound signal and when exactly the value for this signal is being calculated. For two consecutive outbound sync times t_1 and t_2 there are two choices:

1. The value for t_2 is processed at t_1. We want to call this *pre–time processing*. The main loop of application threads would have the following layout:

```
1 while (1) {
2   <calculate and update signal values>;
3   wait( <for some time or some event> );
4 }
```

Listing 1: Pre–Time Processing Thread

2. The value for t_2 is processed at t_2. We want to call this *in–time processing*. In contrast to pre–time processing the application needs to follow the following layout:

```
1 while (1) {
2   wait( <for some time or some event> );
3   <calculate and update signal values>;
4 }
```

Listing 2: In–Time Processing Thread

In–time processing is actually the common way for describing RTL models in VHDL or SystemC (see also [8]).

For event–driven outbound synchronization only the in–time method would be applicable.

For both of these choices there have to be considered several further restrictions:

6.3.1 Pre–Time Processing

As the new outbound signal values for t_2 are processed at t_1, it has to be guaranteed that the values for t_1 have been sent out already. Furthermore, it has to be guaranteed that the inbound sync for t_1 has been finished as well. This is important because the inbound signal information received at t_1 very likely affects the outbound signals to be synced out at t_2.

What is required here is a barrier–functionality that keeps application threads blocked until the synchronization has been finished. This can be easily accomplished by introducing an appropriate barrier function that idles the thread in delta cycles until the requirement is met. This function would be called by the application thread prior to the processing and update of outbound signals.

6.3.2 In–Time Processing

The outbound sync module needs to wait for several delta cycles until it can be sure that the new values have been processed. This is because the outbound signal values are processed at the same simulation time as they are synced out.

Similarly as for pre–time processing there is a barrier required. However, here the synchronization library needs to wait for the application threads rather than vice versa. This turns out to be much more complicated than it appears. How can the synchronization library be sure whether the application threads have updated the signals?

Assuming the application threads are scheduled in the first delta cycle of a given simulation time, this wouldn't be a big deal. The synchronization library could just wait for a delta cycle when it is being woken up. Unfortunately, an application thread does not necessarily need to be scheduled in the first delta cycle. Consider an example where a thread is triggered (clocked) by some signal. The clock signal will change in the first delta cycle which means the thread will be scheduled in the second delta cycle. Even more worse, there can't be given a specific number of delta cycles after what the thread will be scheduled. Finally, this depends on the number of trigger indirections.

In general it is safe to read the most current values of outbound signals when

a) there are no pending signal updates or event notifications.

b) there are no other threads ready to run.

Unfortunately, this information is not directly accessible from the standard SystemC. There has also been not yet found some way to check these conditions indirectly without modifying SystemC.

6.3.3 Final Choice: In–Time Processing

In the end the decision was made to make use of in–time processing as the way to go — despite of the problems of a clean implementation and related limitations. The reasons for this decision are as following:

1. Pre–time processing would require this nasty barrier call that needs to be carried out at the beginning of every thread loop. This would imply to go one more step away from a transparent solution.
2. In–time processing appears to be more clean as the actual signal semantics matches the logical semantics and hence behaves more natural. I.e. a signal is changed when it should be changed, and not sometime in past. Additionally, the local signal and the according remote signals change synchronously at the same simulation time. Of course, a simulation model that is aware of the [pre–time processing] situation can easily deal with it. But one needs to take more care when specifying the model.
3. Pre–time processing is not compatible with event–driven outbound synchronization. Hence

there is more application recoding required when the outbound synchronization is changed from time–driven to event–driven (for whatever reason). In contrast, a switch from time–driven in–time processing semantics to event–driven semantics requires no change of the code at all. Except in these cases where outbound signals are not regularly updated (→ see advantages of time–driven outbound sync in section 6.2).

4. Associated with the last point, there would be no migration to optimistic DES possible as this would imply event–driven outbound synchronization.

5. A very practical issue: There is just no need to deal with any level of trigger indirections for the targeted application. Even a single indirection by use of a simple clock is not really necessary.

6.3.4 Final Semantics

To conclude, the semantics of the signal distribution can be described as follows:

> The new value of an outbound signal that changes at simulation time t becomes effective at the remote simulation kernel at the same simulation time t. Outbound synchronization for t completes in the third delta cycle. Inbound synchronization for t completes **after** the outbound synchronization for t has completed (if there is any). All inbound signals will become updated within **exactly one delta cycle**.

Additional notes (other consequences):

- An inbound signal that affects an outbound signal with zero propagation delay behaves like a register (provided the inbound/outbound update times match each other). That is, asynchronous paths from inbound to outbound modules will be broken up.
- Inbound signals can't be used as clocks for registering other inbound signals in a common RTL context. Instead of taking a snapshot of the old value, the new value would be taken by the register. This behavior is deterministic as the synchronization library ensures that all inbound signals become updated within a single delta cycle.
  Other regular signals can't be clocked by inbound signals as well when they affect outbound signals. This is because the notification of the outbound signals will be already sent out before the clock becomes effective. This would introduce a phase–shift of one cycle for these outbound signals. I.e. the signal values arrive at remote simulation kernels with a delay of one cycle. Btw., this latter issue would not appear with event–driven outbound synchronization.
  This problem might be addressed in future.

6.4 Application Thread Conventions

The basic application thread framework has already been shown in listing 2 (page 5). A more detailed template is shown in listing 3.

```
1 <write initial values for outbound signals>;
2 while (1) {
3   <may NOT read inbound signals>;
4   <may NOT write outbound signals>;
5   wait( <for some time or some event> );
6   <process...>; // read/write as wanted
7 }
```

Listing 3: Template for Application Thread

The reasons for this layout have already been discussed in section 6.3.

A SystemC thread normally consists of an endless loop. Before entering the loop, the thread usually initializes a few or all of its output signals. This is done here as well. This part is optional and includes not only outbound signals but normal signals as well (i.e. signals that are not feed to remote simulation kernels).

Note that prior the call of `wait()` inbound signals should not be read (line 3) and outbound signals should not be written (line 4). Inbound signals should not be read prior to the `wait()` as this will return outdated values. Assuming that an inbound signal changes at the same rate as the reading thread cycles, reading the signal in cycle N would return the signal value from cycle $N-1$. In particular this means that in cycle 0 an undefined value would be read. Similarly, writing an outbound signal prior to `wait()` would overwrite the value that has just been written (lines 1 or 6).

The `wait()` statement in line 5 advances the simulation time. The `wait()` method that is being used can be actually any kind (static sensitivity, dynamic sensitivity, time driven, etc.). See also [7] pp. 19 and [6] pp. 82. The only restriction is that the thread is going to sleep for exactly the same time that matches the explicit lookahead information for the changed outbound signals that has been specified during the setup phase (see later). Further this implies that all outbound signals of a certain thread need to share one and the same period.

For an example how such a thread is looking in practice refer to the example given later in this document.

Of course, application threads can also differ from the template shown in listing 3 — as long as the simulation semantics is respected.

6.5 Application Method Conventions

If `SC_METHOD` is used instead of `SC_THRED`, the template layout follows exactly this one that is used to describe registers for RTL synthesis [8].

```
1 <process...>; // read/write as wanted
2 next_trigger( <for some time or some event> );
```

Listing 4: Template for Application Method

So there's nothing special. `next_trigger()` is usually not present in synthesizable RTL code. It is not required when the method is statically sensitive for a clock.

Note that the first invocation of the method will return junk when it is reading inbound signals as well as other local signals. However, that is normal for this kind of description and is not associated with distributed simulation. Usually one makes use of some reset signal there, hence avoiding the evaluation of uninitialized signals.

In listing 5 a more behavioral template of a method is shown mimicking the behavior exactly as the thread template in listing 3 does.

```
1 if (<simulation time is zero>) {
2   <write initial values for outbound signals>;
3 } else {
4   <process...>; // read/write as wanted
5 }
6 next_trigger( <for some time or some event> );
```

Listing 5: Alternative Template for Application Method

7 Reference Implementation

The following subsections describe the implementation in more detail.

7.1 Outbound Synchronization

The algorithm that is associated with outbound synchronization can be described by the pseudo code shown in listing 6.

```
1  while (1) {
2    wait(0); // for delta delay cycle;
3    wait(0); // for another delta delay cycle;
4    time = current simulation time;
5    while (time==
6           time stamp of an outbound signal) {
7      <read signal value>;
8      <send signal notification>;
9      <update signal time stamp>;
10   }
11   time = min(all outbound time stamps) - time;
12   wait(time);
13 }
```

Listing 6: Outbound Sync Pseudo Code

The shown pseudo code is the body of a SystemC thread and can be used

- on a per–outbound–signal basis (i.e. there is one thread for each individual outbound signal)
- on a per–outbound–sync–module basis (i.e. there is one thread for each outbound sync module that handles all outbound signals attached to this module)
- globally for all outbound signals.

Which one of these choices to use does basically not matter. The current reference implementation makes use of only one outbound synchronization thread for all outbound signals. This minimizes the extra scheduling overhead induced by the synchronization.

The two waits for a delta cycle at the beginning of the processing loop are very important — at least one of them. One delta cycle is required to ensure that all application threads have been scheduled and the corresponding signals have been updated. The second delta cycle attributes to the problem with in–time signal processing and allows to handle applications with at most one trigger indirection. Refer to section 6.3 for details. For the case that it is not really needed, the second `wait()` does cause no harm. It just costs some time.

Note: As a simple workaround to allow more levels of indirections it might me useful to extend the thread in that way, that it can wait for an adjustable number of delta cycles. The number would be set up according the application needs somewhere during the elaboration phase.

7.2 Inbound Synchronization

The pseudo code for the SystemC thread responsible for inbound synchronization does not differ very much from outbound synchronization and is shown in listing 7.

```
1  while (1) {
2    time = current simulation time;
3    while ( outbound sync not finished ) {
4      wait(0); // for delta delay cycle;
5    }
6    while (time==
7           time stamp of an inbound signal) {
8      <receive signal notification>;
9      <update signal value>;
10     <update signal time stamp>;
11   }
12   time = min(all inbound time stamps) - time;
13   wait(time);
14 }
```

Listing 7: Inbound Sync Pseudo Code

The actual algorithm is a little bit more complicated because it has to respect the fact that signal notifications are received not necessarily in the expected order.

Similarly as for outbound synchronization, this algorithm might be used on a per–signal basis, a per–

module basis, or for all inbound signals. The current implementation makes use of the latter approach.

While the outbound sync thread(s) play a more or less passive role, the inbound sync thread(s) can be considered more active. This is because they basically stop the simulation until appropriate signal notifications have been received.

7.3 Data Transmission Medium

There are no special qualitative requirements and almost any medium (including shared memory) or communication library might be used. In fact, MPI (Message Passing Interface) shows up some convenient features. A successful combination of SystemC and MPI has already been demonstrated in [9]. However, this first reference implementation makes use of TCP/IP only.

7.4 System Setup

The synchronization infrastructure is being built up by using various functions provided by the synchronization library. This is done during the elaboration phase usually inside `sc_main()` before the actual simulation is started by `sc_start()`. This is the only occasion were the application programmer needs to make use of special library functions.

The actions to be performed during system setup are as following:

- A number of outbound sync modules need to be created (number can also be zero).
- A number of inbound synch modules need to be created (number can also be zero).
- A number of outbound signals need to be attached to outbound sync modules. This also includes the specification of the signal period for each individual signal (explicit lookahead information). One and the same signal can be attached multiple times to the same or multiple outbound sync modules.
- A number of inbound signals need to be attached to inbound sync modules.
- Once everything has been set up, all local inbound and outbound sync modules have to be connected with their remote counterpart.

After connection of the distributed simulation kernels it is neither allowed to create more inbound or outbound sync modules, nor it is allowed to attach any more signals.

7.5 Library Functions

Currently, the synchronization library provides the following three functions:

- `sc_dfsync();`
- `attach();`
- `sc_dfsync_conall();`

A more detailed description is following here:

```
sc_dfsync(
  int *status, int medium, int direction,
  int designator,
  char *remote_hostname = NULL,
  int remote_port = 0
);
```

`sc_dfsync()` is a constructor that is used to create virtual instances of inbound or outbound synchronization modules. The latter two arguments are used for outbound modules only. The only supported parameter for `medium` is `TCP` so far. `direction` can be either `SYNC_IN` or `SYNC_OUT`. The `designator` is an unique identifier for the inbound or outbound module to be created. The designator acts as some kind of global identifier and has to match the designator of the module's remote counterpart. `remote_hostname` and `remote_port` has to be specified in case of outbound modules. During the connection phase the local simulation kernel will use this information to connect the outbound module with the right remote inbound module. `status` returns usually `0`; or `-1` when some error occurred.

```
int attach(
  int designator, double period,
  sc_signal<double> *signal
);
```

`attach()` is a public member–function of `sc_dfsync`. When used with these arguments it attaches a local signal as outbound signal to an outbound synchronization module. Currently, the only supported signal type is `double`. The `designator` is an unique identifier within the corresponding outbound-/inbound sync module context. `period` specifies how many micro seconds the signal keeps its current value at a time. `attach()` returns usually `0` or `-1` when some error occurred.

```
int attach(
  int designator,
  sc_signal<double> *signal
);
```

This version of `attach()` attaches a signal to an inbound synchronization module.

```
int sc_dfsync_conall(
  int medium, int server_portnum
);
```

`sc_dfsync_conall()` is a friend–function of the class `sc_dfsync`. This function connects all local inbound and outbound sync modules with their remote counterparts and returns when this task has

been finished. The only supported value for `medium` is `TCP` currently. `server_portnum` specifies a free local TCP/IP port where a server is set up to listen for connections from remote simulation kernels.

7.6 Implementation Details

Internal details of the reference implementation are not to be disclosed here. Here are just a few words.

The most important things have been already discussed in sections 7.1 and 7.2. In the last phase of `sc_dfsync_conall()` there are two SystemC threads instantiated — one responsible for inbound synchronization and the other for outbound synchronization. That is, the distinction in unique inbound and outbound module is merely a logical abstraction. These threads handle data structures including various information for specific signals as well as pointers to these signals. This makes it possible to evaluate and manipulate them.

The data structures are currently simple linked lists. This is not very elegant and suboptimal. But it is working fine for a first proof–of–concept.

8 Exemplary Use

Figure 2 shows an exemplary distributed system.

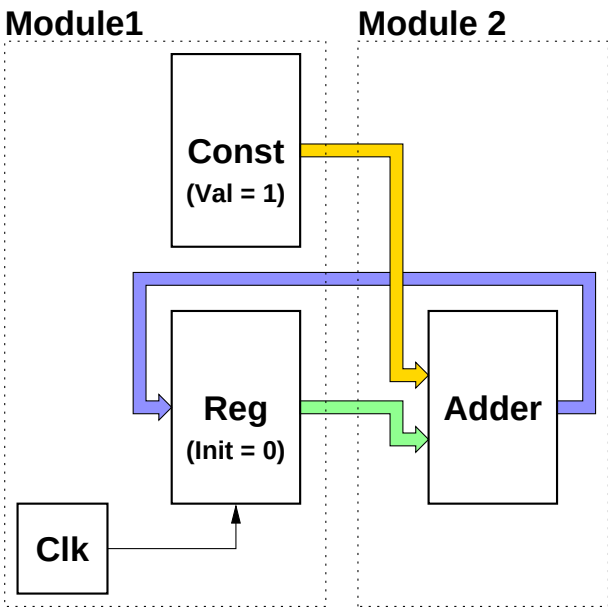

Fig. 2: Exemplary Distributed System

It is basically some kind of counter. In order to keep the example simple, we want to distribute the components across two SystemC simulation kernels as shown in the figure. They could be spread across more kernels, of course. An exception is that the clock and the register can't be put in different simulation kernels with the current implementation (see also section 6.3).

Anyways, with every rising edge of `Clk` the register `Reg` shall pick up the output of `Adder` which adds the `Reg` and `Const` (1 here). As a result, the content of `Reg` is incremented with every rising clock edge: 0,1,2,3,4,5,...

The listing of the main program of Module1 is shown in listing 8.

The code is not very complicated and basically self–explaining. Note that the error–checking has been skipped for this example. Normally, all relevant functions return a status (either as return value or via argument) in order to signal some problems.

The two outbound signals `reg_out` and `const_out` are attached to the created outbound synchronization module (named `outbound_module` here). The update period of `reg_out` is specified with 100μs. As `const_out` will never change, we can basically specify any arbitrary value different from zero. In reality, one would chose a value as large as possible in order to reduce communication to a minimum.

`outbound_module` is set to connect to port `10011` on `localhost`. So both modules will run on the same host. Of course, instead of `localhost` any arbitrary hostname can be specified (provided the second module will be started there).

The code for the actual register thread is shown in listing 9.

This code does contain nothing special that would mark it as specifically written for this distributed application. I.e. there are no specific calls to some synchronization functions. However, the code is following the strict layout that is required for proper operation. It is also important that the update rate (100μs) of the register matches exactly the value that has been specified during `attach()` of `reg_out`. If both do not match, the simulation will not work correctly.

For completion, the header file of the register is shown in listing 10.

The according source code for Module2 is shown in listings 11, 12, and 13.

Note that the adder is working with the double frequency of the register (50μs vs. 100μs). This is required because we are simulating on a Register Transfer Level. The adder is considered as an asynchronous element here. If the frequency of both adder and register would be the same, the signal path register→adder→register would take two clock cycles instead of one only. Basically, it would be the same effect when one would model a normal SystemC or VHDL design where the logic between registers has the same propagation delay like the clock period of the registers.

Both SystemC simulation instances can be started in arbitrary order.

```
#include "systemc.h"
#include "synchronize.h"
#include "clocked_reg.h"

int sc_main(int ac, char *av[])
{
  // Some declarations.
  int status;
  sc_signal<double> reg_out;
  sc_signal<double> reg_in;
  sc_signal<double> const_out;

  // Instantiate an outbound sync module. It is TCP/IP based and will connect
  // to host localhost / port 10011. The module designator is 1.
  sc_dfsync outbound_module( &status, TCP, SYNC_OUT, 1, "localhost", 10011 );

  // Attach our two outbound signals and specify periods (in us).
  outbound_module.attach( 1, 100, &reg_out );
  outbound_module.attach( 2, 1000, &const_out );

  // Instantiate an inbound sync module. It is TCP/IP based and the module designator is 1 here.
  sc_dfsync inbound_module( &status, TCP, SYNC_IN, 1 );

  // Attach the one and only inbound signal.
  inbound_module.attach( 1, &reg_in );

  // Connect all modules. Remote outbound sync modules have to connect to port 10010.
  sc_dfsync_conall( TCP, 10010 );

  // Time base is 1us.
  sc_set_default_time_unit( 1, SC_US );

  // Create a clock: 100us period, 50% duty cycle, start at 100us
  sc_clock clk("clk", 100, SC_US, 0.5, 100, SC_US);

  // Instantiate and connect the register component.
  reg REG("reg");
  REG.clock(clk);
  REG.reg_in(reg_in);
  REG.reg_out(reg_out);

  // Initialize the constant.
  const_out.write(1);

  // Start the simulation.
  sc_start();
  return 0;
}
```

Listing 8: Module1 Main Code

```
#include "systemc.h"
#include "clocked_reg.h"

void reg::process()
{
  double regval;
  // Specify an initial register content and
  // init the SystemC signal as well.
  regval = 0;
  reg_out.write(regval);

  while (1) {
    // Just to display something:
    cout << "Time: " << sc_simulation_time()
         << " => Current Value of Reg is "
         << regval << endl;
    // Wait for our clock.
    wait();
    // Update the register content.
    regval = reg_in.read();
    reg_out.write( regval );
  }
}
```

Listing 9: Register SystemC Thread

```
struct reg : sc_module {
  sc_in<double> reg_in;
  sc_out<double> reg_out;
  sc_in_clk clock;

  void process();

  SC_CTOR( reg ) {
    SC_THREAD( process );
    sensitive_pos << clock;
  }
};
```

Listing 10: Register Header File (`clocked_reg.h`)

9 Tasks to be done

The library in its current form is not yet ready for production use and its function calls are not yet engraved in stone.

Here's a list of several first–order issues that need to be solved:

```
#include "systemc.h"
#include "synchronize.h"
#include "adder.h"

int sc_main(int ac, char *av[])
{
  // Some declarations.
  int status;
  sc_signal<double> a_in, b_in;
  sc_signal<double> sum_out;

  // Instantiate an outbound sync module. It is TCP/IP based and will connect
  // to host localhost / port 10010. The module designator is 1.
  sc_dfsync outbound_module( &status, TCP, SYNC_OUT, 1, "localhost", 10010 );

  // The outbound signal sum_out is being attached to OutboundModule and
  // will become updated every 50us.
  outbound_module.attach( 1, 50, &sum_out );

  // Instantiate an inbound sync module. It is TCP/IP based and the module
  // designator is 1 here.
  sc_dfsync input_module( &status, TCP, SYNC_IN, 1 );
  input_module.attach( 1, &a_in );
  input_module.attach( 2, &b_in );

  sc_dfsync_conall( TCP, 10011 );

  sc_set_default_time_unit( 1, SC_US);

  adder ADD("adder");
  ADD.a(a_in);
  ADD.b(b_in);
  ADD.sum(sum_out);

  sc_start();
  return 0;
}
```

Listing 11: Module2 Main Code

```
#include "systemc.h"
#include "adder.h"

void adder::process()
{
  // Initialize the output.
  sum.write(0);

  while (1) {
    // Wait 50us and ...
    wait( 50, SC_US );
    // calculate the new output value.
    sum.write( a.read() + b.read() );
  }
}
```

Listing 12: Adder SystemC Thread

```
struct adder : sc_module {
  sc_in<double> a;
  sc_in<double> b;
  sc_out<double> sum;

  void process();

  SC_CTOR( adder ) {
    SC_THREAD( process );
  }
};
```

Listing 13: Adder Header File (`adder.h`)

- Other SystemC signal types need to be supported rather than only `double`.
- The time–base that is currently fixed to micro seconds is not acceptable.
- There are some hand work issues needed to be done to improve the general performance. The current signal management is based on linked lists and needs to be replaced by trees or other mechanisms yielding better scalability.
- Signal notifications do not always need to contain a time stamp. For the current version that does not support the change of outbound signal update rates it would be even sufficient to exchange this information only once after connection.
- Support of other communication media such as Shared Memory or MPI (Message Passing Interface). This includes the introduction of some kind of abstraction for connection and data transfers.
- Do some benchmarking and compare some more reasonable models (single vs. distributed SystemC model) in terms of simulation performance.

Some second–order things are:

- A mechanism for changing the outbound signal update periods during runtime could yield to more flexibility as the application might want to adapt this dynamically as needed. This would,

however, require sync library calls outside the elaboration phase.

- It would also be nice when the issue with the distribution of clock signals (see section 6.3.4) could be solved. It has not yet been investigated in deep detail and it is not clear whether it can be solved without modifying SystemC. But it appears that it will require the introduction of explicitly assigned signal priorities.

More distant ideas that come in mind are:

- It needs to be evaluated in more detail which additions to SystemC would be needed in order to implement some functionality more efficiently or to solve some existing odds.
- It could be seriously considered to integrate the synchronization directly into SystemC.

10 Related Work

After an intensive investigation there have been found no suitable projects that deal with distributed SystemC. In fact, this was the reason for the decision to rise up the project described herein. There has been found a project of Masachika Hamabe [9] that deals with the simulation of distributed SystemC models that are connected via MPI. However, this does not handle a common global simulation time and is therefore only suitable for a very limited class of applications.

A very promising practical implementation of distributed simulation is discussed in [3] and [4]. However, this project is aimed at VHDL and not at SystemC. The authors discuss a mixture of conservative and optimistic simulation that has a resolution down to the delta cycle level. In order to meet their goals they had to write a dedicated simulator core.

11 Conclusions

This paper has proposed a first version of a simple–to–use library that can be used in order to spread SystemC RTL models easily across multiple computing nodes. The simplicity has been shown with a simple example.

The synchronization library is based purely on conservative distributed simulation. Vital timing information is not determined by the simulator itself but is specified by the model programmer once at the beginning. For the actual model description the programmer is not concerned anymore with the distribution. Though, he needs to follow a few simple rules.

Further, the library can be used in junction with standard SystemC implementations without the need to modify them.

The use of the synchronization library is not only limited to the distribution of a big SystemC model across many computers or to interface a SystemC model with other simulation cores. The library has also a potential to be used as interface to online visualization modules as well as interactive input modules.

References

[1] Alois Ferscha: *Parallel and Distributed Simulation of Discrete Event Systems.*
Contribution to the "Handbook of Parallel and Distributed Computing", McGraw-Hill, 1995

[2] Richard M. Fujimoto: *Parallel and Distributed Simulation.*
In proceedings of the 1999 Winter Simulation Conference

[3] Dragos Lungeanu and C.J. Richard Shi: *Distributed Simulation of VLSI Systems via Lookahead-Free Self-Adaptive Optimistic and Conservative Synchronization.*
In proceedings of ICCAD, 1999

[4] Dragos Lungeanu and C.J. Richard Shi: *Parallel and Distributed VHDL Simulation.*
In proceedings of Design, Automation and Test in Europe Conference (DATE'00), Paris, France, March 2000

[5] Stuart Swan: *An Introduction to System Level Modeling in SystemC 2.0.*
Cadence Design Systems, Inc. May 2001, Open SystemC Initiative (OSCI)

[6] *SystemC 2.0.1 Language Reference Manual.*
Revision 1.0, 2003, Open SystemC Initiative (OSCI)

[7] *FUNCTIONAL SPECIFICATION FOR SYSTEMC 2.0 (Update for SystemC 2.0.1).*
Version 2.0-Q April 5, 2002, SystemC Language Working Group

[8] *Describing Synthesizable RTL in SystemC.*
Version 1.2, November 2002, Synopsys, Inc.

[9] SystemC project website of Masachika Hamabe:
`http://www5a.biglobe.ne.jp/~hamabe/SystemC/`