Digital Force White Paper

Date: 28th November 2004
(Last Build: 28th November 2004)

A First Mature Revision of a Synchronization Library for Distributed RTL Simulation in SystemC™

Mario Trams

Mario.Trams@digital-force.net

Thanks go to Marko Meyer and Friedrich Seifert of foobar GmbH
as well as Daniel Balkanski of Chemnitz University of Technology.
They provided some help in various technical aspects of the library.

Digital Force / Mario Trams

http://www.digital-force.net

A First Mature Revision of a Synchronization Library for Distributed RTL Simulation in SystemC™

Mario Trams

Mario.Trams@digital-force.net

Thanks go to Marko Meyer and Friedrich Seifert of foobar GmbH as well as Daniel Balkanski of Chemnitz University of Technology. They provided some help in various technical aspects of the library.

Abstract

This paper presents a first production–ready version of a synchronization library for simulation of distributed SystemC models. This synchronization library is the result of a consequent continuation of a rudimentary library that has been used to demonstrate the fundamental feasibility of a concept known as explicit lookahead [1] for the distributed simulation of RTL–like models. The basic idea behind that concept was to make use of the fact that those models follow a strict (clocked) timing regime where the clock frequency is a known constant.

Keywords

SystemC™, Conservative Distributed Discrete Event Simulation, Register–Transfer–Level (RTL) Models

1 Background and Motivation

In [1] a mechanism has been described that allows a more or less easy distribution of RTL–like SystemC simulation models across multiple hosts. The basic idea was the combination of conservative distributed discrete event simulation with the fact of a well–known timing regime. That is, update rates of all signals are known prior to the start of the simulation. This information is explicitly transfered to the synchronization library and was termed *Explicit Lookahead*.

The library described in [1] was intended as some kind of proof–of–concept in order to demonstrate the fundamental operation of such a library. It contained various limitations that made it hard to use. Many of those limitations have been eliminated in the current version 1.1.0 of the library, which can be considered production–ready.

In this paper a few details about the revised library are discussed as well as the library API itself.

The document is organized as follows: After a short review of the basic goals of the library there are discussed various aspects that have been changed and/or improved. Following that, a few particular implementation issues are highlighted. Special emphasis is put on the support of arbitrary signal types which is a fundamental new feature (section 5). Section 9 provides a brief reference of the library API. The paper concludes with a short discussion on future tasks to be done and related work.

Although this paper is widely self–contained, it is based on the work presented in [1]. Even though there have been made a few changes in comparison to the things presented that time, it is strongly recommended to be aware of the contents of the older paper.

2 Terminology

The denotation of various important things discussed within this paper is widely consistent with the one used in [1]. For completeness, some important terms are shortly defined herein as follows:

- *simulation kernel* —
  Refers to a single SystemC application process. A distributed SystemC model consists of multiple simulation kernels. A more theoretical denotation of a simulation kernel is a *logical process*, or *LP*.
- *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 synchroniza-

tion for 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 ...
- *synchronization cycle* —
  A synchronization cycle refers to the processing of all inbound resp. outbound signals for a certain simulation time.

3 Recapitulation of Basic Goals

The primary goal pursued by this development is the creation of a library supporting the partitioning of a large RTL–based SystemC simulation model into single pieces. These pieces might be executed in parallel resulting in a faster overall simulation.

Figure 1 illustrates an exemplary distributed system involving three simulation kernels. (The figure has been already discussed in [1].)

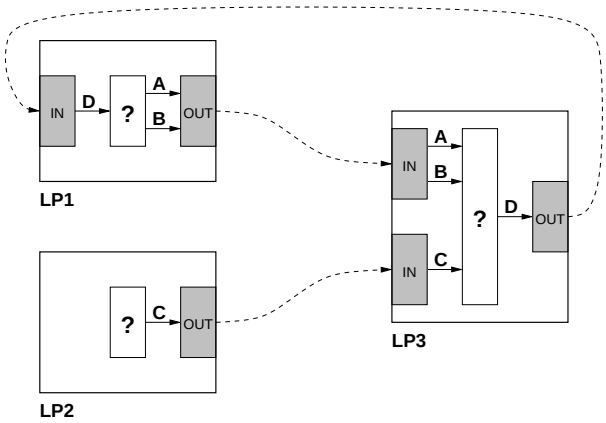

Fig. 1: Exemplary Distributed System

The application writer/programmer has to be able to create inbound and outbound sync modules as needed. Each outbound/inbound sync module pair acts as a tunnel and keeps the signals attached to both modules consistent in their value. In particular, the value of an inbound signal will follow the value of its corresponding outbound signal. Referring to figure 1, the signal `A` of the third simulation kernel (LP3) will be updated regularly with the value of signal `A` of LP1.

The update cycles of individual signals have to be specified explicitly by the application writer/programmer. This is to be done once during the elaboration phase of a SystemC simulation kernel. Although this explicit timing specification appears to be quite problematic, it is no major problem for RTL–like simulation models. In [1] this concept has been discussed in detail.

The actual simulation model is to be influenced in no way by the use of the synchronization library.

4 Important Library Changes and Enhancements

The following subsections describe various aspects of the synchronization library that have been changed or newly introduced in comparison to the first proof–of–concept design that has been presented in [1].

4.1 Slightly changed Simulation Semantics

The distributed simulation semantics that has been initially stated in [1] was:

> 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**.

This semantics has been slightly changed (simplified, actually):

> 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. Both outbound and inbound synchronization are performed within a single delta cycle (normally in the third → see section 4.3)

The initial requirement to carry out the inbound synchronization strictly after outbound synchronization is not needed anymore. This is because a synchronization cycle takes only a single delta cycle now.

That time, it was necessary to execute inbound synchronization after outbound synchronization because it had to be guaranteed that outbound signals are not manipulated by inbound synchronization before they are synced out. As long as outbound synchronization takes place in the same delta cycle as inbound synchronization, the signal values won't have changed. This is because SystemC carries out the actual signal value updates at the end of a delta cycle within the so–called update–phase.

All in all, the modified semantics is compatible with the application thread/method specification as described in [1]. It has therefore no impact on existing applications.

4.2 Only one synchronization Thread

As described in [1], earlier versions of the synchronization library used one SystemC thread for inbound synchronization and another one for outbound synchronization. This has been changed so that there is running just one controlling thread now. This thread schedules (and interleaves) individual inbound and outbound synchronization tasks. This has two primary advantages:

- The whole synchronization cycle can be done within a single delta cycle which brings in several benefits.
- The synchronization tasks can be scheduled more efficiently depending on data–availability from the network connection.

4.3 Delta Cycle Relaxation

Initially, outbound synchronization has been done within the third delta cycle of a certain simulation time. This was necessary in order to allow at least one trigger indirection in the system which is essential for the use of clocks. Refer also to [1], section 7.1. In the same section it has been proposed to provide support for starting the outbound synchronization (and hence the inbound synchronization as well) in an application–selectable delta cycle number.

Now, this has been implemented and the application can set an according parameter.

Note: This feature does still not allow the distribution of clock signals!

4.4 Introduction of a Phase Shift Mechanism

There has been introduced a feature termed *Phase Shift* that is opening various interesting opportunities. This mechanism effectively violates the synchronization semantics in a controlled manner as defined by the application. The basic idea is as following:

> When outbound signals are attached to the library, a second timing parameter (the phase shift) can be specified besides the cycle. This phase shift offsets the individual synchronization times for this signal accordingly.

As an example, a cycle time of 10ns and a phase shift of 1ns results in synchronization times of 1ns, 11ns, 21ns, and so on. The phase shift parameter is treated as a modulo with respect to the corresponding update cycle and does only offset the synchronization times. I.e. a cycle time of 10ns and a phase shift of 11ns results in synchronization times for the signal of 11ns, 21ns, 31ns, and so on.

Well, what purpose does this feature serve? It can be used to separate the signal synchronization from the actual signal calculation in terms of simulation time and not in terms of delta cycles. Consider an example where some register outputs are feed through some combinational logic that calculates the final value of an outbound signal. This increases the level of indirections accordingly. Instead of increasing the relaxation parameter (see section 4.3) we could specify a slight phase shift. So we don't have to care about the number of indirection levels anymore.

Yet another application is the distribution of clock signals. Effectively, the phase shift can be used to define priorities. That is, signals with a smaller phase shift are synchronized earlier. Consider another example that is clocked by a certain frequency (say 100Mhz, or 10ns cycle time). This clock is to be generated within a single simulation kernel and distributed to other kernels. The clock signal would be synchronized every half clock period (i.e. with a cycle of 5ns) and all other signals in the whole model would be synchronized every clock period (i.e. with a cycle of 10ns) and a phase shift of say 1ns.

Though, it is still needed that every kernel knows about the actual clock rate as it has to specify the right update cycle for outbound signals.

Note: Although this mechanism violates the synchronization semantics in a controlled way, it is absolutely conform to the modeling of RTL systems. From a physical point of view the phase shift does nothing more than introducing a clock–to–output delay.

4.5 Slightly changed Module Designation Policy

The library presented in [1] used the following logical scheme for determining which outbound sync module is to be connected with which inbound sync module:

- Each inbound sync module receives a designator that is unique in a global context (i.e. unique within the whole distributed application).
- Each outbound sync module receives a designator that is unique in a global context as well.
- Each outbound sync module receives an information to which simulation kernel it has to be connected.
- During module connection, inbound and outbound sync modules with matching designators are mated together.

This scheme contains an inconsistency. The requirement for globally unique designators is absolutely necessary when the simulation kernels have an N:N relation (i.e. each kernel is connected to each other one). But when not all kernels know from each other, it is easily possible to construct valid scenarios where

more than one inbound or outbound sync module carry the same designator.

In order to remove this inconsistency, the connection scheme has been slightly changed:

- Each inbound sync module receives a designator that is unique in a local context (i.e. unique within the corresponding simulation kernel).
- Each outbound sync module receives a designator that is also unique in a local context.
- Each outbound sync module receives an information to which simulation kernel and to which inbound sync module located in the specified simulation kernel it has to be connected to.

In the context of module designation, this allows to consider individual simulation kernels more decoupled from the whole distributed application. As an interesting side effect, it also opens the door for proprietary simulation kernel executables that can be easily plugged into other simulation applications.

4.6 Support for Arbitrary Signal Types

The library does support any signal type now rather than only `double`. This includes user–defined types as well. Even more, this does not only include simple, flat types such as simple structs without pointers. The library does provide necessary fundamental support to deal with arbitrary abstract and complex signal type structs and classes. Although the library cannot provide such a support out–of–the–box for all unknown types, it can be easily customized.

The introduction of this important feature required a fundamental internal reorganization of the library. More details on this can be found in section 5.

4.7 Extended Time Support

While the first experimental version of the library supported only a granularity of one micro second for outbound signal update cycles, any arbitrary time can be specified now. These times can be specified either in form of a previously created `sc_time` object or as `double`/`sc_time_unit` pair.

In addition, each simulation kernel can make use of arbitrary simulation time resolutions and default time units. Though, the simulation resolution of a given kernel needs to be sufficiently high for handling inbound signals. When an outbound signal changes at a rate that is below the resolution of the according remote inbound sync module (resp. the corresponding simulation kernel), this will fail, obviously.

Such error conditions will be detected and reported within a consistency check (see section 4.8).

4.8 Introducing Consistency Check

In order to guarantee a flawless synchronization operation, a signal consistency check has been introduced. This consistency check is carried out for individual outbound/inbound sync module pairs and ensures that various requirements are met.

The whole check is carried out during the elaboration phase before the actual simulation starts. Hence, it does not introduce significant runtime penalties.

More details on this consistency check can be found in section 6 later in this document.

4.9 Introducing Flow Control

Most distributed simulation models will have a closed–loop nature. That is, each simulation kernel will somehow depend from each other (directly or indirectly). As a result of this dependency, all simulation kernels will uniformly advance in time so that no one will be significantly ahead in time compared to others. This can be quite different in case of open–loop simulations such as in simple producer–consumer scenarios.

Figure 1 actually contains such an open–loop relation involving simulation kernels 2 and 3. As LP2 depends on no other kernel, it could possibly generate signals significantly faster than they can be processed by LP3. As a result, LP2 is much more ahead of LP3 in simulation time.

Although such condition is principally not semantically problematic it rises up some practical issues such as:

- Increased CPU and network load. Hence, the faster progressing simulation kernel could draw performance from the slower ones that have more work to do, actually.
- In case of (potentially) endless simulations the simulation time of the faster progressing simulation kernel might overflow. This could lead to some unpredicted behavior which can terminate the whole simulation prematurely.

Therefore a flow control mechanism has been integrated in order to prevent such an aggressive proceeding. This flow control is rather simple. Outbound sync modules, and hence the whole associated simulation kernel, are allowed to be maximally a certain amount of synchronization cycles ahead of the remote inbound sync module. The amount of cycles can be specified by the application by changing an according library parameter (default value is 10).

4.10 Restriction to TCP/IP only

Early library designs took some first preparations for supporting different communication media, per-

spectively. This support consisted of a `medium`–parameter in corresponding library calls. In fact, only TCP/IP was supported that time.

The library presented herein supports only TCP/IP as well. However, the formal support for other media has been removed temporary. The primary reason for this decision is the fact that the old way of specifying information related to specific communication media directly in the application code is not very elegant anyways. It has been found much more flexible when both the actual simulation kernel and information related to kind and parameters of the communication medium become separated. This allows to change some parameters (i.e. port numbers and host names in case of TCP/IP) without the need to recompile the application itself.

Of course, the general idea to support arbitrary communication media has not been dropped. Merely, the formal support has been delayed into a future mechanism based on a configuration file. Details of this mechanism need to be worked out.

4.11 Logical Library Appearance

The appearance of the library has slightly changed in comparison to the library discussed in [1]. There are three classes now which are organized in a so–called factory model. Figure 2 illustrates the dependencies in a simplified diagram.

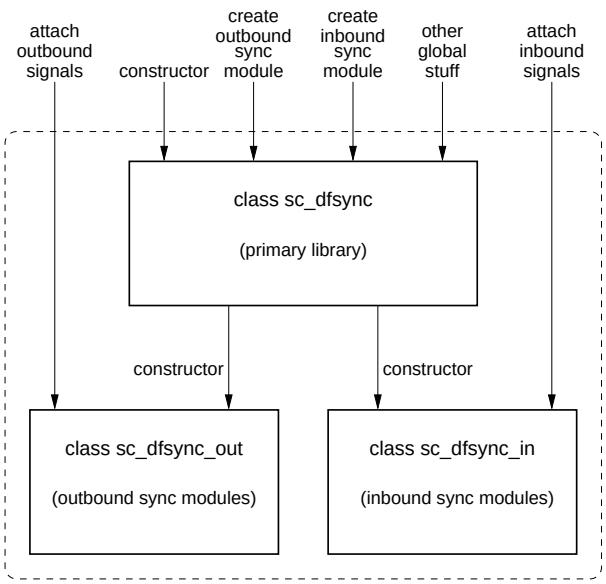

Fig. 2: Basic Class Structure

The class `sc_dfsync` represents the synchronization library as a whole. The two other classes represent outbound resp. inbound sync modules (`sc_dfsync_out` and `sc_dfsync_in`). Instances of `sc_dfsync_out` and `sc_dfsync_in` can only be created by calling member functions of `sc_dfsync`. Attaching individual signals is done by directly calling member functions of the according class. Although it is equivalent from a functional point of view, this solution has been found more clean than the previous single–class approach.

Note that the context of the primary class `sc_dfsync` is completely static. That is, it is not possible to create more than one instance of the library.

In addition, all functions and publicly available structures have been defined within a C++ name space called `dfsync`.

More details of the library functions can be found later in this paper.

Note: This class structure is just a logical structure of the external appearance that is relevant for the application writer/programmer. In fact, the actual library classes and related stuff are hidden by a mechanism also known as *Cheshire Cat*.

4.12 Optimized Signal Processing

Although not visible to the outside, the processing of inbound and outbound signals has been improved significantly. Signals are managed in lists on a per–synchronization–module basis. Operations on these lists have been optimized by making use of some sophisticated mechanisms. These mechanisms take advantage of the specific characteristics of the list operations according their special nature.

There have been made no benchmarks how much these optimizations bring quantitatively. But formally they reduce the overhead for signal list modifications significantly. In fact, under special (not very unlike) scenarios this overhead does even not depend on the signal count at all.

5 Handling of Arbitrary Signal Types

One of the most crucial parts of the library is the support of signals with arbitrary types. When it comes to the wire or a low–level data transmission, signal values are to be transferred as simple blocks of bytes. So it is required to convert the current value of a signal into a plain binary block on one side, transmit it, convert it back into the signal type, and update the SystemC signal with it. In general, this concept is also known as *Serialization/Deserialization*.

5.1 Type Support in regular Communication Libraries

In ordinary communication libraries such as MPI [2], the support of different types is an important feature as well. Whenever a message is to be transferred, the application specifies a void pointer for the data as well as a type identifier. MPI supports inherently a number of intrinsic types and knows how to serialize

them. This does even include type conversion for heterogeneous architectures.

Special user–defined types such as structs have to be registered with the library during runtime. That is, the application tells the library how a certain structure is composed out of existing known data types. Finally, the outline of such a registered structure is stored in a so–called *typemap* that holds all relevant information. For instance: There is a `double` located at offset 0, then there is an `int` at offset 8, then a `char` at offset 12, etc. Based on this information the MPI library can handle the type correctly.

Clearly, this mechanism forbids the use of complex types containing pointers. Objects of such types have to be flattened before they can be sent (the reverse operation is needed during receive). This conversion has to be done explicitly by application code and disfigures it therefore.

In any case, such a mechanism appears to be almost unusable for a SystemC synchronization library. While one can imagine that there is provided support for registering user–defined structures just like in case of MPI, this is impossible for more complex types. This is because in a SystemC model the communication happens implicitly as there are no send/receive calls.

However, C++ provides powerful mechanisms that allows the support of arbitrary complex types. This is discussed in the following sections.

5.2 How to Serialize/Deserialize Signal Values?

In case of intrinsic C types such a serialization is trivial: The value of a variable of a certain type can be accessed by simply casting its pointer to a character array and read or write a number of bytes depending on the size of the type.

There is no difference when the type is a simple, flat struct or class. "Simple and flat" means that the struct contains only a number of variables of intrinsic types or other simple and flat structs. A class can contain in addition constructors and member functions etc. But they are not relevant here as they are not related to the state of the object. Like the intrinsic types, those types span across a consecutive number of bytes in memory and can be accessed by casting according pointers into a character array as well.

It is getting problematic when it comes to really complex and abstract classes and structs that contain pointer variables. Obviously, it does not make sense to carry around those pointers. So those kinds of types cannot be handled in a generic way as it is possible for the other kinds discussed above. Even worse, there is no generic mechanism to serialize or deserialize such types. In that point, every type is individual and cannot be supported out–of–the–box by any library — the meaning of the internal type structure cannot be guessed.

However, when the library cannot provide a direct support for those types, it can at least provide appropriate mechanisms allowing the extension of the library.

When it is coming to different types, the template concept of C++ becomes very important. Hence, the synchronization library makes extensive use of this feature. But actually, this statement should be slightly corrected: The public header files complementing the library use them extensively.

5.3 Anonymifying Signal Types

A crucial point for dealing with arbitrary signal types is to introduce some kind of anonymification. That is, when the actual library cannot be prepared for all particular kinds of types, it must not be interested in the type and its peculiarities at all. All it needs to know of a certain signal type is

- the byte–count of the serialized value
- a function used to read (serialize) and write (deserialize) a signal value

With regard to the latter one, the use of so–called functors has been found very useful. These functors can be used for hiding type–specific functionality behind type–unspecific functions. This is exactly what is needed here.

There are two kinds of functors involved:

- A signal–write functor `TFunctor_w` that encapsulates a function for updating a signal value from a plain serialized character array.
- A signal–read functor `TFunctor_r` that encapsulates a function for reading a signal value and serializing it into a plain character array. A second function for determining the current size of a the serialized signal value is included as well. Later we will see what purpose this function does serve.

Listing 1 shows exemplary the functor used for writing signal values. This code is based on [4] and has been slightly adapted to the needs of this application.

The details of this code are not to be discussed here (refer to [4] instead). All it does is to encapsulate a type–specific member function of a class `TClass` that accepts a pointer to a character array. This function can be called by making use of a type–unspecific function (or operator `()` in this case).

The class `TClass` and its member function that is carrying out the actual signal update and whose pointer is stored in the functor does finally depend on the according signal type. Details about this class are discussed next.

```
1  class TFunctor_w {
2   public:
3    virtual void operator()(char* array)=0;
4  };
5
6  template <class TClass>
7  class TSpecificFunctor_w
8    : public TFunctor_w {
9   private:
10   void (TClass::*fpt)(char*);
11   TClass* pt2Object;
12  public:
13   TSpecificFunctor_w(
14     TClass* _pt2Object,
15     void(TClass::*_fpt)(char*)
16   ) {
17     pt2Object = _pt2Object;
18     fpt=_fpt;
19   };
20
21   virtual void operator()(char* array) {
22     (*pt2Object.*fpt)(array);
23   };
24 };
```

Listing 1: Functor Classes

5.4 Providing Support for individual Types

So far, the basic support of arbitrary types has been discussed. That is, when the library wants to read a signal value into a plain character array in order to send it out, it just calls a function that is stored inside the functor that is kept along with other information about the signal. The same applies for updating a signal value. What remains to be discussed is how these functors are set up.

There is a general template class named `functorize_class`. This class provides two member functions `functorize_inbound_signal()` and `functorize_outbound_signal()` that are responsible for handling inbound and outbound signals. Initially, there is no implementation code behind that class. This code has to be provided by template specializations of `functorize_class` for the desired types.

The two member functions of the `functorize_class` are responsible for carrying out the following tasks:

- Set a type identifier for the according type. This is mostly only interesting for all known predefined types (see below). All other types receive a special identifier marking them as unknown.
- Set a clear–text type name for type–identification purposes. This is not needed for the known types as they can be identified using the type identifier.
- Set the size for the serialized signal value so that the library knows how much bytes it has to handle. In case of directly supported (known) SystemC types this specifies the width of the vector. The size can also be set to zero which means the serialized signal value can have an arbitrary size.
- Create and initialize the according functor so that it wraps the right serialization or deserialization functions. The serialization/deserialization functionality is to be provided by according classes.

The library provides a number of specializations for various known types (see below). User–defined template specializations can also be provided by including an according header file after the header file of the synchronization library. This makes the library very flexible as it can be customized easily.

The following subsections discuss the handling of various types in more detail.

5.4.1 Built–in C Types and simple flat UDTs

As shortly discussed in section 5.2, those types can be handled rather easily by doing some casting acrobatics. Because of their similarity, they are handled by the same mechanism. That is, they utilize one and the same specialization of the `functorize_class`. In fact, they do not utilize a specialization at all but use the default implementation.

Note: This means that every signal type with no specialization will be handled in that way. This leads to serious problems when the signal type contains pointers nonetheless!

Listing 2 shows the implementation of the according `functorize_inbound_signal()` function.

```
1  template < typename T >
2  TFunctor_w* functorize_class<T>::
3  functorize_inbound_signal(
4    sc_signal<T>& signal,
5    int& signal_type,
6    size_t& signal_type_size,
7    const char*& signal_type_name
8    ) {
9
10   signal_type = determine_signal_type( signal );
11   signal_type_size = sizeof(T);
12   signal_type_name = typeid(T).name();
13
14   deserialize_flat<T>* deserialize_object;
15   TSpecificFunctor_w< deserialize_flat<T> >*
16     write_functor;
17
18   deserialize_object =
19     new deserialize_flat<T>( &signal );
20
21   write_functor =
22     new TSpecificFunctor_w< deserialize_flat<T> > (
23       deserialize_object,
24       &deserialize_flat<T>::do_deserialize
25     );
26
27   return write_functor;
28
29 }
```

Listing 2: `functorize_inbound_signal()` for flat Types

The signal type is determined (line 10) using a small helper function that compares the formal type `T` against all known intrinsic types using `typeid()`. In case of an UDT, there is returned a special ID

marking unknown types. Type size and type name is determined as shown.

The actual functor is created and initialized in lines 21–25. As the functor needs to be initialized with the right parameters for signal value deserialization, there has to be created an according object (`deserialize_object`, lines 18/19). This object is here from the template class `deserialize_flat`. The definition of this class is shown in listing 3.

```
1  template < typename T >
2  class deserialize_flat {
3   protected:
4    sc_signal<T>* my_signal;
5   public:
6    // The constructor just stores the signal pointer.
7    deserialize_flat( sc_signal<T>* signal ) {
8      my_signal = signal;
9    }
10   void do_deserialize( char* new_value ) {
11     my_signal->write( *( (T*)new_value ));
12   }
13 };
```

Listing 3: Class `deserialize_flat`

As it can be seen, this deserialization class is fairly simple. The constructor just stores the signal pointer and the `do_deserialize()` member function casts the plain character array into the according type and writes it into the signal.

The corresponding functionality for handling of outbound signals (i.e. `functorize_outbound_signal()` and the class for serialization) is almost identical and does not need to be discussed in detail here.

5.4.2 Types brought by SystemC

There is provided support for all special SystemC types. These types fall in several different categories each of which requiring different serialization/deserialization methods.

Values of `sc_bit` and `sc_logic` are externally represented by single ASCII characters. The serialization function makes use of the `to_char()` method provided for both types. This is at least necessary for `sc_logic`, as there is no numerical representation for `'X'` and `'Z'`. Deserialization is a simple assignment, as both types support an assignment operator for `char`.

Values of type `sc_lv<>` are represented by a NULL–terminated ASCII string that is extracted via `to_string()`/`c_str()`. Similar as for `sc_logic`, the string handling is needed because of `'X'` and `'Z'`.

The integer types `sc_bigint<>`, `sc_biguint<>`, `sc_int<>`, `sc_uint<>`, and `sc_bv<>` as well as the fixed–point types `sc_fixed<>`, `sc_ufixed<>`, `sc_fix`, and `sc_ufix` can be represented in a compact binary format. In fact, values are extracted for serialization in blocks of 32 bits using the `range()` and `to_uint()` functions. Deserialization is done similarly.

The "fast" fixed–point limited precision SystemC types `sc_fixed_fast<>`, `sc_ufixed_fast<>`, `sc_fix_fast`, and `sc_ufix_fast` are represented internally (in SystemC) as `double` and hence are treated externally as `double` as well by making use of the `to_double()` method. The cannot be safely accessed with the `range()` and `to_uint()` functions because they are not bit–true when their size exceeds the precision of `double`.

The basic schemes of the functorization classes as well as the classes for serialization/deserialization of the SystemC types follow the same principle that is used for the flat types. So there is no need to show them up here.

5.4.3 Special Types with flexibly sized Values

An interesting class of types are those whose values have no specific fixed size. A very simple example is the class `string` that is provided by the standard C++ library. When reading the signal value, the synchronization library needs to know the current value size so that it can prepare accordingly sized buffers etc. As shortly mentioned in section 5.3, the functor mechanism for reading signals provides a second function intended to determine the current value size. So before the signal value is read into a plain character array, the library can determine the current value size.

Note that the size determination is not needed in case of fixed–sized types and the according function pointer can be set to `NULL`.

As an example, the synchronization library in its current revision provides an header file containing according functionality for the handling of `string` types. Although this type is a little bit unusual for SystemC signals it is working fine.

Note: Note that SystemC is normally not prepared for dealing with `string` as signal type. In order to use `string`, there needs to be provided an additional `sc_trace()` function. Refer also to [9] page 98.

5.4.4 Unknown arbitrary complex UDTs

For all kinds of types not discussed so far there have to be provided separate header files with according template specializations for `functorize_class` as well as specific serialization/deserialization classes. The header file supplying support for the `string` class serves as an excellent example for own types.

5.5 Far–reaching Capabilities

Perspectively, the use of the template concept for providing type–specific templates is very powerful

in view of portability and platform interoperability (although this is not that important right now as the synchronization library is exclusively available for x86 architectures).

As an example, imagine a little–endian and a big–endian machine and the particular type `int`. When such and `int` is transferred from one machine to the other, this will fail because of different endianess. But one could provide a specialization for `int` that takes care about the situation and makes use of (whatever) well–defined external representation for `int`. In addition, instead of using `typeid().name()` for type name determination some unmistakable clear text such as ¨`32 Bit Integer`¨ could be used. In the end, the actual synchronization library just serves as some kind of tunnel transporting anonymous blocks of bytes from one simulation kernel to another one.

Although the library is currently well aware about the intrinsic types as well as the SystemC types and is principally able to hide such conversions, The according specializations could be also changed so that the library treats those types as UDTs.

5.6 Automated Template Specialization Generation?

As described in section 5.1, an MPI application has to register special data types with the MPI library. This is a nasty task that has to be done once at the beginning. In case there are complex types with pointers involved, these types have to be serialized/deserialized by the application every time they are transmitted. This is an even more nasty task.

Well, as described above, in case of the synchronization library such a registration and serialization is not a part of the actual simulation model code. Nevertheless the creation of according template specializations is required at least for types with pointers. Although this code is not part of the actual simulation model, it's creation can be considered "nasty" as well.

As for MPI, once there has been developed an interesting framework that simplifies the handling of complex types [3]. In particular, there is a tool called AutoMap that creates MPI data types from C structures. This is done by analyzing the source code. In junction with another tool called AutoLink, complete dynamic data structures can be sent and received without the need for explicit serialization done by the application.

One can imagine a similar tool for automated creation of according serialization classes and related stuff. It would read in the source code of the type definition and creates an according header file.

6 Ensuring Consistency

As shortly introduced in section 4.8 there has been included a consistency check that ensures a flawless synchronization operation between individual outbound/inbound sync module pairs. In detail, the following requirements have to be met:

- Both outbound and inbound signal count have to match each other.
- For each given outbound signal designator, there has to exist a corresponding inbound signal with the same designator.
- The specified outbound signal timing parameters have to be large enough so that they can be handled at the inbound sync module. This test can fail due to an insufficient simulation resolution at the inbound sync module.
- The signal type of the outbound signal has to match the type of the corresponding inbound signal, of course.

While the check for most conditions is rather trivial, this is not the case for the signal type. As described in section 5, there are three parameters for types of attached signals determined: A numerical type identifier, a type name in form of an ASCII text, and the size of the type. The type name can be a generic one determined by `typeid().name()` or some other text defined by the according template specialization. The numerical type identifier is only of use when the type is a known one.

In general, two types are considered identical when the type identifiers match each other. When the type identifier indicates a SystemC vector type (integer as well as fixed–point), the width of the vector needs to be identical too. In case both identifiers mark an unknown signal type, the type name as well as the type size need to be identical.

Note that fixed–point types are currently only compared based on their total word length. That is differences in binary point position, overflow and quantization modes, and number of saturation bits won't be detected. This issue could be addressed in future.

Although this type checking mechanism is working well for flat UDTs in most scenarios, it is not perfect. The primary issue is compiler dependency. For instance there is no standard about the formation of generic type names returned by `typeid().name()`. This name can vary from compiler to compiler although the underlying structure is exactly the same.

However, as described in section 5.5, those odds could be principally removed by providing template specializations for ANY individual type.

7 Physical Module Connection Establishment

Outbound sync modules do not connect directly to the corresponding inbound sync module of the remote simulation kernel. Instead, there is set up one server at a user–defined TCP/IP port. All outbound sync modules that want to connect to some inbound sync module within one simulation kernel have to connect to this port in a first instance. The final module–to–module connection is done by the protocol.

In order to make the module connection somewhat systematic from a technical point of view, it has been decided to take a threaded approach. That is, for each inbound and outbound sync module a thread is created. Each thread is responsible for carrying out all tasks required for establishing the connection physically as well as logically. The latter includes especially the signal consistency check.

The threads used for connection establishment are no SystemC threads. Instead, the GNU Portable Thread library, or Pth [5] has been used. Pth is a general–purpose thread library based on non–preemptive multithreading. Actually, SystemC brings along its own threading mechanism that could be exploited as well for these purposes. I.e. it is well possible to create SystemC threads that perform all required actions before they terminate. However, a potential problem is that these threads will be runnable after the first call of `sc_start()`. The policy that has been used so far is to have all modules successfully connected before the simulation becomes started at all. Another issue is that these SystemC threads would have to release CPU control by waiting for a delta cycle. This could also disturb the simulation semantics. For a final decision and an eventual move to SystemC threads these issues need to be analyzed in more detail.

8 Synchronization Operation

The actual simulation can start only when the consistency check has been passed successfully for all inbound and outbound sync modules. Otherwise the simulation kernel will terminate — as it is usual for SystemC–related errors.

During synchronization, outbound sync modules send signal change notifications containing only a signal designator and a binary copy of the new signal value. Any time–information has been removed from the notifications. The first proof–of–concept version of the synchronization library [1] used to transmit a time information telling when the next notification for this signal is to be expected. Now, the update cycle is transmitted once along with the consistency check and the inbound sync module is responsible for determining the according synchronization times.

As described in section 4.9, the synchronization operation incorporates a flow control mechanism now. This configurable mechanism avoids a too fast progressing of the outbound sync module compared to its corresponding inbound sync module. The flow control mechanism is implemented by tracking acknowledges that are sent by inbound sync modules after a synchronization cycle has been completed.

On a low level, there is used a special packet–based communication protocol involving intelligent buffers. These buffers are intended to collect data and transmit it in larger blocks. This leads to a vast reduction of OS calls and hence reduces overhead. The size of these buffers can be customized in view of performance tuning (large buffers do not mean highest performance). Refer to [12] for more details.

9 Library API

The basic structure of the library has been shortly discussed in section 4.11. It provides three classes: The primary library class (`sc_dfsync`) and inbound and outbound sync module classes (`sc_dfsync_in` and `sc_dfsync_out`).

Note that all names are defined within the name space `dfsync`. For simplicity, the `dfsync::` prefix has been omitted here.

Also note that the function reference given here is kept somewhat short and not all details are discussed. Refer to [12] for a more detailed description.

9.1 `class sc_dfsync`

This class represents the basic synchronization library. There can be instantiated only one instance of this class.

9.1.1 Constructor `sc_dfsync()`

This simple constructor is to be used for creating an instance of the synchronization library.

9.1.2 Destructor `~sc_dfsync()`

This is the destructor for the library and is normally not explicitly used by the application.

9.1.3 `sc_dfsync_in inbound_module()`

This member function has to be called for creating a new inbound synchronization module. It returns an object from the class `sc_dfsync_in` which is used for later interactions with that particular module.

Full Synopsis:

```
sc_dfsync_in inbound_module(
  const unsigned int designator
);
```

There is only one argument that specifies the designator of this particular inbound sync module.

9.1.4 sc_dfsync_out outbound_module()

This member function has to be called for creating a new outbound synchronization module. It returns an object from the class `sc_dfsync_out` which is used for later interactions with that particular module.

Full Synopsis:

```
sc_dfsync_out outbound_module(
  const unsigned int designator,
  const char* remote_hostname, const int remote_port,
  const unsigned int remote_designator
);
```

`designator` specifies the designator of this particular outbound sync module.

The pair `remote_hostname` and `remote_port` specifies hostname and port number (TCP/IP) of the remote simulation kernel where this outbound sync module shall connect to later.

Finally, `remote_designator` specifies the designator of the according remote inbound sync module at the simulation kernel that has been specified by hostname/port.

9.1.5 int set_parameter()

By using this member function it is possible to manipulate some parameters relevant for the synchronization library as a whole (i.e. not specific to individual inbound or outbound sync modules).

Full Synopsis:

```
int set_parameter(
  const unsigned int parameter,
  const unsigned int value
);
```

`parameter` specifies a parameter identifier and `value` the new parameter value. Refer to [12] for details.

9.1.6 int connect_all()

This member function is used for connecting all created inbound and outbound sync modules with their remote counterparts according the information that has been specified during module creation. The function does not return until all local inbound and outbound sync modules have been successfully connected.

After connection, neither new inbound/outbound sync modules can be created, nor new signals can be attached.

Full Synopsis:

```
int connect_all(
  const int portnum
);
```

The `portnum` argument specifies the TCP/IP port number that is to be used for setting up a server socket. Remote outbound sync modules that want to connect to one of the local inbound sync modules have to use this port number.

When there has been created no inbound sync module, the value of the `portnum` argument does not matter.

9.2 class sc_dfsync_in

The class `sc_dfsync_in` represents inbound sync modules. The constructor of this class is protected and can only be invoked indirectly through the class `sc_dfsync`.

9.2.1 int attach()

`attach()` is used for attaching a signal to the inbound sync module.

Full Synopsis:

```
template <typename T>
int attach(
  const unsigned int designator,
  sc_signal<T>& signal
) {}
```

The `signal` argument can be any SystemC signal with arbitrary type.

Note that now a reference variable is used instead of an explicit pointer for the signal. This makes the application code to look more beautiful :-)

9.3 class sc_dfsync_out

The class `sc_dfsync_out` represents outbound sync modules. The constructor of this class is protected and can only be invoked indirectly through the class `sc_dfsync`.

9.3.1 int attach()

`attach()` is used for attaching a signal to the outbound sync module.

Full Synopsis:

```
template <typename T>
int attach(
  const unsigned int designator,
  const sc_signal<T>& signal,
  const sc_time cycle,
  const sc_time phase_shift
) {}
```

Note that for convenience there have been provided various other templates differing in the way `cycle` or `phase_shift` are specified. Refer to [12] for details.

The `signal` argument can be any SystemC signal with arbitrary type. The update cycle can be specified either by handing over a pre–defined `sc_time` variable, or by specifying a

value/time unit pair (whatever appears to be more applicable). The same applies to the phase shift which can also be omitted (assumes zero phase shift). Of course, a cycle of 0 is forbidden and is rejected by the library.

9.3.2 `int set_parameter()`

The class `sc_dfsync_out` has a public member function that allows the setting of parameters which are related to individual instances of outbound sync modules.

Full Synopsis:

```
int set_parameter(
  const unsigned int parameter,
  const unsigned int value
);
```

`parameter` specifies a parameter identifier and `value` the new parameter value. Refer to [12] for details.

10 Example

Well, it has been decided to skip a detailed example description here due to space considerations. Instead, an according description of an updated example that has been initially described in [1] has been outsourced into [12].

In general, the example has been changed mostly due to the new library API. The only more or less fundamental change is that the result of the combinational adder component does not need to be synced out at twice the clock rate of the register. This is possible because of the newly introduced phase–shift mechanism.

It is planned to release somewhere in future a more complex exemplary model incorporating a CPU (perhaps even more than one CPU), some memory, and some IO.

11 Future Work

Actually, the library development is in a state where it fulfills all vital requirements that have been planned for the targeted application. Nevertheless there are still a few things that can be improved regarding some internal/external handling and data transfer issues.

Another qualitative improvement is planned with the introduction of a configuration file mechanism. That is, connection information such as host names, port numbers, etc. becomes outsourced from the actual application code into a configuration file. This avoids a code recompilation in case there is made only a change of a host name, for instance.

In junction with the configuration file mechanism there are also plans to include final support for other communication media rather than TCP/IP only. This support of different media should be transparent for the actual application code. That is, neither changes in the code nor recompilations shall be required, ideally. Instead of reinventing the wheel, existing flexible communication libraries for heterogeneous networks could be used. VMI 2.0 [6] is a good candidate here, as this is based on dynamically loadable media–modules. The direct use of MPI is also interesting and perhaps more future–proof than VMI 2.0. In any case, TCP/IP should be directly supported as an always–working fallback solution.

Other tasks include, but are not limited to:

- There are a few issues regarding error–handling that can be improved.
- Analyze in more detail whether SystemC threads instead of Pth threads can be utilized for connection setup purposes (see section 7).
- From an application point–of–view it might be useful to consider synchronization modules as bidirectional. I.e. rather than independent inbound and outbound sync modules there is a unified class of synchronization modules accepting both inbound and outbound signals.
- In view of performance evaluation, there have been still made no tests with more complex realistic models. The models used so far were only intended for basic functionality and stress tests. They have a very bad (low) computation/communication ratio yielding in largely increased runtimes.

Practical deployment of the library will show whether there are additional enhancements/changes needed.

12 Availability

As the synchronization library is in a state where it becomes interesting for serious practical use, it has been made publicly available and can be downloaded for free from

http://www.digital-force.net/projects/dist_systemc

Along with the library there is provided a more or less extensive user manual.

Note that the library is exclusively available for x86 Linux.

13 Ongoing Related Work

While there are many active projects dealing with distributed simulation as such, it is still rather silent in the field of distributed SystemC simulation. At least there has been started a new project with quite ambitious goals. Within the context of a rather large

project targeted at the modeling of more or less complete computer systems (RITSim Microarchitectural Simulator [10]) there is a sub–project with the aim of distributing the simulation transparently [11]. The development of the basic concepts behind that mechanism is still in progress. One distinguished goal is to modify the SystemC library directly, which is an absolute must for the desired transparency.

In this context it remains open whether it is really that worthwhile to manipulate SystemC itself. Unless the result does not become part of SystemC as such, changes will have to be ported from version to version. This can be very problematic. Not to speak of proprietary (=very fast) implementations where changes of the code are not possible at all. So rather than changing a given implementation, it appears to be better to make an analysis and determine which additional fundamental functionality should be provided by SystemC itself. This additional (presumably small) set of functions could be adopted by the official SystemC specification and synchronization libraries can be developed exclusively on top of the SystemC library.

14 Summary and Outlook

The early rudimentary library used as proof–of–concept for a distributed SystemC library exploiting the concept of explicit lookahead has been successfully turned into a library for first practical use. This document presented various vital mechanisms that have been developed in order to put the generalized library into practice.

Nevertheless, there is still potential for improvement in both qualitative and quantitative domains in sight. Apart from that, practical use within the targeted field of application will show whether there are more fundamental enhancements required or desirable.

References

[1] Mario Trams: *Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead.* Digital Force White Paper, February 2004. Available from http://www.digital-force.net/publications

[2] Primary MPI web site: http://www.mpi-forum.org

[3] Martial Michel and Judith E. Devaney: *A Generalized Approach for Transferring Data–Types with Arbitrary Communication Libraries.* In Proceedings of the Workshop on Multimedia Network Systems (MMNS'2000) at the 7th International Conference on Parallel and Distributed Systems (ICPADS '2000), July 2000.

[4] Lars Haendel: *The Function Pointer Tutorials; Introduction to C and C++ Function Pointers, Callbacks and Functors.* August 2003. See also: http://www.function-pointer.org/functor.html

[5] Ralf S. Engelschall: *GNU Pth - The GNU Portable Threads.* See also project web site: http://www.gnu.org/software/pth/pth.html

[6] Scott Pakin and Avneesh Pant: *VMI 2.0: A Dynamically Reconfigurable Messaging Layer for Availability, Usability, and Management.* In Proceedings of HCPA–8, 2002. See also: http://vmi.ncsa.uiuc.edu

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

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

[9] *SystemC™ Version 2.0 User's Guide.* Update for SystemC 2.0.1, 2002

[10] *RITSim Microarchitectural Simulator*, see also: http://www.ce.rit.edu/~gpseec/Research.html

[11] David R. Cox: *RITSim: Distributed SystemC Simulation.* Proposal for Master Thesis, Rochester Institute of Technology, Dept. of Computer Engineering, July 2004. Available from http://www.ce.rit.edu/~gpseec/Research.html

[12] *User Manual for Distributed SystemC™ Synchronization Library Rev. 1.1.0* Digital Force Public Documentation, November 2004. Available from http://www.digital-force.net/projects/dist_systemc