Digital Force Public Documentation

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

User Manual
for Distributed SystemC™ Synchronization Library
Rev. 1.1.0

Mario Trams
Mario.Trams@digital-force.net

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

Contents

Chapter 1

Preface

1.1 Why does this Library exist?

Back in 2001, the author has started a certain technological long–term project. Within that project a rather large and interdisciplinary simulation framework is planned. This framework is intended for simulation of a broad range of topics including physical/kinematic processes, electronics hardware, and processing algorithms. If possible, this simulation is to be carried out in real time.

Instead of its initial purpose — primarily the description of hardware and partly software with the goal to simulate and synthesize digital systems — the SystemC library has been found very useful for modeling even physical processes. In addition to this fact, SystemC models are not necessarily just some kind of passive model descriptions as it is the case with VHDL, for instance. No, basically a SystemC model is nothing more than a "normal" C/C++ program. That is, such a model can easily make use of almost all existing libraries and system calls one can imagine. From the author's point of view, this fact makes the SystemC library a massively powerful tool — perhaps even more powerful than initially thought by the inventors.

The simulator framework to be developed is going to become rather large and it has been planned from the beginning to spread different simulation domains or parts across multiple computers. Apart from that, some parts of the simulator might be not based on the SystemC library for some reason. In any case, there is a mechanism required that is keeping all involved simulation processes somehow in sync and is exchanging information about signal values among these processes.

Well, just for evaluation purposes, such a mechanism was quickly developed within days. In particular, data was exchanged between some GTK–sliders, a simulation core based on SystemC, as well as another process responsible for some graphical OpenGL–representation. Within that exemplary system, everything was basically hand–made. This means the mechanism for exchanging data was part of the actual simulation model.

Of course, this has been found not very elegant and it is getting more problematic with increased model complexity. So there was born the idea to develop a library that can perform this synchronization more or less alone — i.e. without interaction of the actual simulation model.

So there has been made some investigation regarding the general state of distributed simulation from a science point of view in order to see which concepts have been developed so far. Also, there has been made an analysis of what can be done in this direction with SystemC at all, and what is required for the actual application. The resulting findings of this investigation have been presented in [6]. The essential conclusion was to go a semi–transparent way where information about the change rate of signals whose values are to be sent to other processes is specified once at the beginning of the simulation. This feature has been called *Explicit Lookahead*.

Despite the fact that such a synchronization library is required for a special purpose, it is believed that a distributed simulation of SystemC models will become very important in future as the acceptance of SystemC as such increases. This is especially true in view of the ever increasing complexity of digital systems. In that context, this concept for distributed simulation of SystemC models matches almost perfectly with the functional cycle–accurate simulation of models based on Register–Transfer–Level (RTL). This is because such models are clocked with a certain (fixed) frequency and register outputs change (if they change) at discrete times known from the very beginning. Therefore the library has also a high potential to be used in this area. This potential use of the library in a broader range of applications was also one reason for

providing it on a rather professional level.

1.2 Why is this Library made publicly available?

As described above, the library has been not just developed for the sake of itself but it is rather a byproduct of another project. Nevertheless it has been decided to make the library available to the public in hope that it will be found useful and can help others to solve their problems more efficiently.

Another reason is to draw more public interest onto the field of distributed SystemC simulation. With a broader use of the library there is a higher probability for the detection of possible bugs as well as weak points that could be improved. Apart from that, the existence of this library as the world's first publicly available exemplary reference implementation in its field could lead into the development of similar, much more advanced libraries.

Because of various reasons, the library is currently not available as open source. Nevertheless it is planned to release the library under the LGPL some day.

1.3 What will be the Future of the Library?

The library in its current version described herein fulfills all of the requirements that have been appointed in view of the targeted application. But there are still several issues regarding both quantitative and qualitative characteristics that are planned to be improved. Therefore the development of the library will be continued in the future.

In this context it is important that users should be aware of the fact that software written for this version of the library does not necessarily need to be compatible with future major revisions. The point is that this synchronization library is considered as a research project that is entering a formerly unexplored area. That is, it is impossible to specify and code the perfect solution within one step. Instead, with every new version and with growing experience in practical use there appear new ideas how things could be implemented more efficiently or which additional features would be desirable.

Though, it is believed that possible changes in application codes will be limited to the elaboration phase of the SystemC code. That is, the actual simulation model (i.e. all the instantiated components etc.) can be left untouched. This is because the synchronization library is explicitly targeted at the simulation of RTL–like models. And these models follow a very specific way of description that is not necessarily unique for this project. In particular, the library fits perfectly with the guidelines that have been specified for SystemC RTL coding (see also [5]).

In order to be up–to–date, users should regularly check whether there is a new revision or other information regarding the library available.

Apart from the continuing development process that will be driven by the initial author, anybody (individual, company, ...) is invited to contribute to the synchronization library by reporting problems, bugs, or proposals regarding library improvement.

Chapter 2

Legal Matters

In the following text, the "LIBRARY" (all uppercase letters) refers to the "Distributed SystemC™ Synchronization Library Rev. 1.1.0" as it has been obtained from the author and is described herein (i.e. a tar–archive with various files including this documentation).

As soon as you are directly using and/or distributing the LIBRARY, it is assumed that you are aware of the topics discussed throughout this chapter and have agreed to the terms of use (section 2.1). "Directly using" means writing code that is calling functions of the LIBRARY.

Note that there might apply different conditions for future releases of the LIBRARY.

2.1 Terms of Use / License Agreement

The LIBRARY is not open–source and is published under special conditions described herein.

The LIBRARY can be redistributed "as is" free of charge provided that it is distributed in exactly the same package (i.e. the original tar–archive) including the same contents as it has been initially obtained from the author or downloaded from the author's web site. It is not allowed to redistribute packages with any changes applied to any files contained in the original LIBRARY.

You can embed the LIBRARY into final applications (including proprietary/commercial applications) and distribute and/or sell those applications under your desired conditions. In this case it is up to you to apply any changes to the LIBRARY you need. Though, it is not allowed to change or remove any copyright notice inside the LIBRARY. In addition, you should be aware of the fact that there is given NO WARRANTY for the LIBRARY (see also section 2.2). In case you found the LIBRARY to be stable for your application, it is up to you to provide a warranty for your final application as a whole — AT YOUR OWN RISK.

With respect to the Pth library that is used by the LIBRARY and hence by your final application as well, you have to take care when distributing your application in order to not violate the LGPL. Refer to LGPL clause 6 for a description of the requirements.

2.2 NO WARRANTY

The following text does almost match the warranty clauses of the LGPL (Version 2.1, February 1999) while reflecting the fact that the major part of the LIBRARY is not available in source code.

Official warranty clauses:

> Because the LIBRARY is provided free of charge, there is NO WARRANTY for the LIBRARY, to the extend permitted by applicable law. The LIBRARY is provided "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the LIBRARY is with you as you are integrating the library into a final application. Should the LIBRARY prove defective, you assume the cost of all necessary servicing.
>
> In no event unless required by applicable law will the author be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or

inability to use the LIBRARY (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the LIBRARY to operate with any other software), even if the copyright holder has been advised of the possibility of such damages.

2.3 The LIBRARY and the GNU Portable Thread Library

The LIBRARY makes use of the GNU Portable Thread Library (Pth) that has been developed and released by Ralf S. Engelschall under the LGPL. In particular, Pth version 2.0.0 has been used during development of the LIBRARY.

The Pth library is not provided together with the LIBRARY and needs to be installed separately, if not already done.

It is understood by the author that the LIBRARY falls under clause 6 of the LGPL (version 2.1). In this context it is believed that this distribution of the LIBRARY fulfills all requirements of LGPL clause 6:

- There is given prominent notice that the Pth library is used and that the Pth library is licensed under the LGPL.
- A copy of the license is supplied with the LIBRARY (see file LGPL.txt in the LIBRARY archive).
- When used in an application, the LIBRARY displays a copyright message dedicated to itself as well as to the used Pth library.
- In consistency with clause 6 a) or 6 b) of the LGPL, it is possible for you to combine the LIBRARY with any compatible version of the Pth library depending on your needs.

Chapter 3

Library Installation and Usage

3.1 Requirements

The library is currently only available for 32Bit x86 architectures and is only running under Linux. The library has been developed under Mandrake Linux 8.2 and Linux kernel 2.4.18. However, it is believed that the library is actually not dependent on any Linux distribution or kernel version. Regarding compiler versions, refer to section 3.3 below.

The most important non–standard software that is required for using the library is an existing SystemC version 2.0.1 or 2.1 (2.1beta11) installation. Information on how to get and install this package can be found on the SystemC home page at `http://www.systemc.org`

Another piece of third–party software required by the synchronization library is the GNU Portable Thread Library (Pth). In section 3.5 there are collected a few information on how to obtain and install this library.

Apart from the above mentioned things you need various other standard development tools and libraries. A properly set up Linux system should provide all that is needed.

In order to be able to distribute a simulation across multiple computers, those computers need to be reachable by each other via TCP/IP. It is outside the scope of this documentation to describe how to setup such a network.

3.2 Installation

The installation of the library is not very automated (i.e. via rpm) but still rather simple. All that has to be done is to go to a desired directory where the library is to be installed and unpack the package by

```
tar xvjf systemc_sync-1.1.0.tar.bz2
```

This creates a directory `systemc_sync-1.1.0` that is containing the following subdirectories:

- `doc` contains this manual.
- `include` contains the header files for the synchronization library. Refer to chapter 7 for a brief description of the files.
- `lib_sc-2.0.1_gcc-2.X` contains the library compiled for SystemC 2.0.1 with gcc-2.96.
- `lib_sc-2.0.1_gcc-3.X` contains the library compiled for SystemC 2.0.1 with gcc-3.4.2.
- `lib_sc-2.1beta11_gcc-2.X` contains the library compiled for SystemC 2.1beta11 with gcc-2.96.
- `lib_sc-2.1beta11_gcc-3.X` contains the library compiled for SystemC 2.1beta11 with gcc-3.4.2.
- `example` contains a small example (see also section 3.7).

Depending on the gcc/g++ and SystemC version you are using, you need to select the right library when linking your application. See also sections 3.3 and 3.4.

The library directories contain a static library archive (`libsystemc_sync.a`).

3.3 Compiler Version

According official documentation, SystemC 2.0.1 requires gcc version 2.95.2 or 2.95.3. It is not really clear where this restriction does come from. Presumably, SystemC version 2.0.1 has only been extensively tested with these compilers. SystemC version 2.1beta11 has been officially released for newer compilers as well.

In fact, this synchronization library has been successfully tested with gcc versions 2.96, 3.2.2, and 3.4.2. Although for SystemC 2.0.1 the 3.X compilers generate several warnings, the executables appear to work fine.

With gcc version 3.0 there has been introduced a new name–mangling scheme for C++. This makes it object–code incompatible with object files generated by earlier gcc versions. That is, when a gcc-2.X is used for application compilation, the SystemC library as well as the synchronization library need to be compiled by gcc-2.X as well. Similarly, when gcc-3.X is used, both libraries have to be compiled by gcc-3.X.

In order to deal with this situation, the synchronization library is provided for both cases. The directories ending with `gcc-2.X` contain a library compiled with gcc-2.96 and should be suitable for all gcc-2.X versions. The directories ending with `gcc-3.X` contain a library compiled with gcc-3.4.2 and should be suitable for all gcc-3.X versions.

Note: gcc-2.X means all gcc versions with a major version 2, and gcc-3.X all gcc versions with a major version 3. Though, it cannot be guaranteed that this does really mean all particular subversions, although it is believed that this is the case. If you are unsure which version you are using, you can check this with `gcc --version`.

3.4 SystemC 2.0.1 and 2.1beta11

The synchronization library version 1.1.0 has been developed and tested using SystemC 2.0.1 as well as SystemC 2.1beta11 (distribution from 1st May 2004). The library has been found working fine for both versions. However, there is one exception:

When using SystemC 2.0.1 you should be aware that it contains a bug in the partial selection for types `sc_bigint<>` and `sc_biguint<>`. As the synchronization library makes use of exactly this partial selection, the handling of these two types fails promptly. When your application makes use of these types, you have to migrate at least to SystemC 2.1beta11 where this bug has been fixed. An alternative would be to write other template specializations for `sc_bigint<>` and `sc_biguint<>` that avoid the use of the partial selection (i.e. by serializing these values with `to_string()`). You can find some information on how to do this in chapter 5 (page 17).

Note that both SystemC versions appear to be object–code incompatible. That is, an object file compiled against the headers of SystemC 2.0.1 cannot be linked against a library of SystemC 2.1beta11 — even when the same compiler versions are used. Because of that reason, the synchronization library has been provided for both SystemC versions.

3.5 The GNU Portable Thread Library

The synchronization library makes use of the GNU Portable Thread Library in order to implement some tasks more clean. In particular, those threads are used during the connection of the inbound and outbound sync modules.

Important Note: The Distributed SystemC Synchronization Library Rev. 1.1.0 has been tested exclusively with version 2.0.0 of the Pth library. Although it is believed that the library is working as well with later Pth versions, there might appear unexpected problems. So just give it a try...

The GNU Portable Thread Library can be obtained from http://www.gnu.org/software/pth/pth.html

Depending on your Linux distribution, there might be also precompiled packages available. However the building and installation from the sources is rather easy and straight–forward as well. The `INSTALL` file contained in the Pth source package describes all steps and parameters in detail.

Here is a short description how to build and install the Pth library on your system. We assume the Pth library is to be installed globally on your system in `/usr/local/pth`. This usually requires that you are logged in as root.

- After unpacking the Pth package enter the created directory (e.g. `pth-2.0.0/`).
- If you want to use another compiler than the default `gcc` you can set the `CC` environment variable. For instance:
  `export CC=/usr/local/gcc-3.4.2/bin/gcc`
  (assumes that you are using the bash)
- Compiler flags can be also set. For instance
  `export CFLAGS=-O3`
  advises the compiler to use the `-O3` optimization level.
- The Pth library is configured by
  `./configure --prefix=/usr/local/pth`
- And then
  ```
  make
  make test
  make install
  ```
- As we usually want the Pth library to be found automatically during program execution in case of dynamic linking, we edit the configuration file of the dynamic linking mechanism (`/etc/ld.so.conf`) and add the line
  `/usr/local/pth/lib`
  After this we have to execute
  `ldconfig`

Note that the last step (changing `/etc/ld.so.conf` and executing `ldconfig`) is only really useful for a global, system–wide installation. In case you want (or need) to install the Pth library somewhere in your home directory, you will have to add the according library path to your `LD_LIBRARY_PATH` environment variable.

3.6 Application Compilation and Linking

The synchronization library is currently only available as static library.

During linking, the library should be always located in front of the SystemC library (i.e. `-lsystemc_sync -lpth -lsystemc`). The reverse order showed up some strange linking problem with recent gcc versions (3.X; definitively in case of 3.2.2 and 3.4.2).

In particular, the problem is an undefined reference to `wait()`. Interestingly, this problem appears only when the application itself does not make use of `wait()`.

Also note that you have to link against the Pth library as well.

The Makefiles of the provided example application can be used as example for creating your own Makefiles.

3.7 Example Compilation

The example that is provided together with the library is described in more detail in chapter 6 of this document. This section does just describe how it can be compiled and started.

The example directory contains two directories: `register` contains the source code for the register simulation kernel and `adder` contains the sources for the adder simulation kernel.

Before compilation it is necessary to adjust the Makefiles according to your system:

- The variable `SYSTEMC` has to be changed so that it is pointing to your SystemC installation directory.
- The variable `PTH_LIB` has to be changed so that it is pointing to the directory where the Pth library files are installed (see also section 3.5). Do not forget to include the actual library directory in the path (i.e. the path has to end with `/lib`).
- The variable `SYNC_LIB` has to point to the directory containing the synchronization library. Depending on the gcc version and the SystemC version you want to use it has to point to one of the four library directories (see section 3.2).

- `SYNC_INC` has to point to the include directory of the library (should be already fine for the example).
- The `CC`–variable has to be set according your desired compiler.
- You might want to specify some other compiler options as well.

Provided that all settings have been adjusted, both simulation kernels can be build by calling `make` in the corresponding directory.

Then both simulation kernels can be started with entering `./register` and `./adder` on two terminals. Note that the example has been prepared to work on a single machine as the host name specified at according places is `localhost`. You might also want to change this, recompile the applications, and run them on different hosts. When you are linking the application statically, you don't need to install all the libraries on each system.

Note: As described in chapter 6, there are provided two alternatives for the register model — One as `SC_METHOD` and another as `SC_THREAD`. By default, `SC_METHOD` is used. You can change that by changing the symbolic link `clocked_reg.h`, which normally points to `clocked_reg_method.h`:

```
rm clocked_reg.h
ln -s clocked_reg_thread.h clocked_reg.h
make clean
make
```

Note: In case one of the used port numbers is already occupied by another process running on your system, the connection fails with an according error message. In this case you have to change the used port numbers (refer also to chapter 6).

3.8 Other Documentation

Apart from the documentation that is provided by this document, there is a white paper [7] which describes various aspects regarding this particular revision 1.1.0 of the synchronization library in more detail. An older paper [6] discusses some more fundamental issues regarding the basic concept behind the method of distributed SystemC simulation that is used by the library described herein.

It is strongly recommended to read those papers before proceeding with own practical experiments.

From time to time there might also appear additional information on the author's according project web site:

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

Chapter 4

Things to take Care for

The following sections describe some minor issues one should be aware of when using the synchronization library. Ignoring them can cause nondeterministic behavior that is difficult to debug.

4.1 Do not exceed the maximal Simulation Time!

The synchronization library takes some precautions for avoiding problems when the maximally possible simulation time is being exceeded. In particular, the library automatically removes all signals that would need to be processed after the simulation time has been overflown.

However, SystemC itself does not take similar precautions. In fact, it seems to ignore the situation completely and behaves somewhat strange. Because of the general confusion, the process scheduling seems to be confused when a `wait()`–call exceeds the maximal simulation time. It is important to note that the problems do already start when the maximal simulation time has not been exceeded but the `wait()` argument exceeds the maximal simulation time when added to the current simulation time.

Therefore it should be ensured that at no time no `wait()` is called that overlaps a simulation time wrapping. To illustrate this by example, when the simulation time lasts for 10 seconds, no `wait()` must suspend the thread for more than 10 seconds. Although it has not been tested in detail, it is believed that the same applies for otherwise triggered threads and methods that are waiting for some event (directly by calling `wait()` without a time or indirectly such as in case of `SC_METHOD`).

As this is a little bit difficult to achieve, it is recommended to not exhaust the complete available total simulation time until the last femto second. Instead, the simulation should be executed for a limited time that inherently avoids the problematic cases. For instance, when the maximal allowable simulation time is 10000s (just a hypothetical value!) and the largest time a process can be suspended is 10s, then a simulation up to 9990s ensures that no timing value overflows.

Notes:

- This is a general SystemC issue and is not only limited to the use of the synchronization library.
- For tracking the simulation time, SystemC makes use of an internal 64 bit counter that is counting the simulation time in simulation resolution ticks. That is, the maximal allowable simulation time is

$$Resolution * (2^{64} - 1)$$

  As an example, when the simulation resolution is set to 1ns, the maximal allowable simulation time is 18446744073709551615fs, or 18446.744073709551615s.

- As long as your specific application stays far below the according maximum (even if you start an endless simulation that becomes terminated otherwise), you do not have to worry about this issue at all.

4.2 Do not change Simulation Resolution after Connection!

The function that is used for connecting the simulation kernels (`connect_all()`, see also section 8.2.6 on page 40) internally performs several calculations based on timing–related parameters. In particular, this involves the simulation resolution and the default time unit. Therefore it is forbidden to change either parameter by `sc_set_time_resolution()` or `sc_set_default_time_unit()` after `connect_all()` has been called. Well, according the SystemC 2.0.1 Language Reference Manual ([2], page 416) it is not possible to adjust both parameters after an `sc_time` object has been created (the synchronization library creates such objects). While this is working well for the simulation resolution and the application terminates in case of an error, this is not the case for the default time unit. Tests have shown that the default time unit can be changed anywhere during the elaboration phase, and in particular also after the creation of the first `sc_time` object. Therefore care should be taken here because changing the default time unit after connection causes unpredictable results.

It is recommended to call `connect_all()` just before the simulation is started with `sc_start()`.

4.3 Be aware of Time–Rounding!

SystemC does round time values (i.e. objects of `sc_time`) according the selected simulation time resolution. When different simulation kernels make use of different simulation resolutions and times are getting rounded, this can lead to problems that become not directly visible. This is best explained based on a small example.

- Let's assume we have a simulation kernel A with a resolution of 10ns and a simulation kernel B with a resolution of 1ns.
- Our simulation model (including both A and B) assumes that a certain signal becomes updated every 8ns and is created in simulation kernel A. Hence, the signal is attached with an update cycle of 8ns to an according outbound sync module in kernel A.
- Because 8ns cannot be represented in a resolution of 10ns, SystemC automatically rounds it. In this case it is rounded up and the actual update cycle will be 10ns instead of 8ns.
- The update rate is reported to the remote inbound sync module in kernel B and the signal will become updated every 10ns.
- Because the actual simulation model (especially this part of the model that is represented by kernel B) expects the signal to change every 8ns, the whole simulation becomes corrupted — silently.

Of course, the problem is finally a direct cause of the badly designed simulation model. That is, normally nobody would specify timing values that are below the actual simulation resolution. But the bad thing is that such a condition is not detected by the synchronization library.

The reverse case where A has a resolution of 1ns and B has a resolution of 10ns would be discovered by the synchronization library. However, the case described above cannot be discovered as the library already receives wrong (rounded) time information. So it has no chance to react appropriately.

Note: There is a theoretical chance to detect such conditions. That is, when the synchronization library does exclusively accept `double`/`sc_time_unit` pairs as update cycle parameter. However, specifying the time in form of an `sc_time` object is much more convenient in most cases.

In general, it is recommended (but not needed) to set the simulation time resolution in all involved simulation kernels to one and the same value.

4.4 Selecting the right Buffer Sizes

As described in section 8.6.2, the synchronization library allows the customization of the size of a special buffer used for low–level communication. Every outbound sync module has such a buffer. The size of the according buffer at the corresponding inbound sync module cannot be set but is automatically adjusted.

The intention of this buffer mechanism is to combine data transfers on user–level. That is, instead of performing an operating system call for transferring individual bytes or words, these buffers take up a

number of smaller transactions. When a buffer becomes full or becomes explicitly flushed, there is made a single operating system call. This mechanism leads to large savings of overhead.

One might think that the larger the buffer size the better is the performance. This is a little bit short–sighted, however.

At a first glance, large buffers do indeed reduce the overhead. But they induce also a higher latency. That is, while a small buffer becomes transmitted in rather short cycles (because it is filled up quickly), large buffers become transmitted more seldom. As a result, data that has been inserted into the buffer at the very beginning cannot be processed for a comparatively long time.

As long as this data cannot be processed by the receiver anyways (because it is busy with other things), this is not a problem. But when the receiver urgently needs this data to continue its work, it is a problem.

Because the actual impact of the buffering depends heavily on parameters such as computing power, network characteristics, and — as an unmeasurable component — the behavior of the application itself, there cannot be given an optimal value. The only way is to play a little bit with the buffer sizes...

Because every outbound sync module has its own adjustable buffer, and every inbound/outbound sync module context might have a different communication characteristics, finding the optimal solution is a complex multi–dimensional problem for complex simulation models. However, it should be not that difficult to find some almost optimal setting with some experimenting and experience.

Chapter 5

Preparing the Synchronization Library for your own Signal Types

The library does provide functionality for handling of most user–defined types (UDTs) out–of–the box. This is valid for all kinds of structures or classes whose state is represented flat in memory. That is, they do not contain any pointer variable.

All signals of these types are handled by the same mechanism that is used for signal types of one of the built–in C types (such as `bool`, `char`, `int`, etc.).

As soon as your type includes pointer variables you have to provide your own code for handling this type. This includes serialization/deserialization functionality that is converting the state of such an object into a couple of bytes and vice versa, as well as some additional type–dependent preparation.

ATTENTION: In case you do not provide this additional code, the library will fall back to the mechanism used for the flat types. As a result, signals of your special type won't be handled correctly. If you have luck, this results in a segmentation fault. If you don't have luck, you will encounter a very strange behavior that you cannot really explain or the synchronization library appears to be broken. So you have to be very careful!

It is theoretically possible to avoid the fall–back to the default handler. In fact, you can do this by yourself by changing the headers appropriately. However, for the sake of convenience the default fall–back is very useful for the regular case. That is, very likely most UDTs in the context of SystemC signals will consist of a flat architecture such as a simple accumulation of intrinsic types. When there is no default handling mechanism, one would need to write appropriate handlers for each of those types.

5.1 What is needed in Detail?

Let's assume you want to let the synchronization library deal with your complex type `your_type`. You have to provide a header file (say `your_type.h`) that contains the following functionality:

- A base class for serializing objects of `your_type` into a character array (say `serialize_your_type`). This can also be a template class depending on your needs. This class has to
  - contain a (non–static) variable of a `const` pointer to an `sc_signal` of `your_type` (`const` because we are not going to change the signal behind).
  - provide a constructor with a `const` pointer argument pointing to an `sc_signal` of `your_type`. This constructor does at least store the pointer in the above mentioned variable.
  - provide a member function `do_serialize()` returning `void` and accepting a single `char*` argument. This member function has to extract the current value of the signal whose pointer has been stored locally. The serialized value is then written into the specified character buffer. How this serialization is to be done depends completely on your type.
    Note: The synchronization library in its revision 1.1.0 allocates the buffer space in 64 bit increments. So if that helps to speed up your code, you can safely use 64 bit accesses when accessing the buffer — even when the actual size is smaller.

  - provide a parameterless member function `return_size()` returning `unsigned int`. This function is only needed when serialized values of `your_type` objects can have different sizes from time to time. Accordingly, this function has to determine the current size of the `your_type` object that has been stored within this object of the `serialize_your_type` class. Again, how this size is do be determined depends solely on `your_type`.

- A base class for deserializing objects of `your_type` from a character array (say `deserialize_your_type`). This can also be a template class depending on your needs. This class has to
  - contain a (non–static) variable of a pointer to an `sc_signal` of `your_type` (here no `const` because we are going to change the signal behind).
  - provide a constructor with a pointer argument pointing to an `sc_signal` of `your_type`. This constructor does at least store the pointer in the above mentioned variable.
  - provide a member function `do_deserialize()` returning `void` and accepting a single `char*` argument. This member function has to update the value of the signal whose pointer has been stored locally. The value is to be taken from the specified character buffer. How this deserialization is to be done depends completely on your type.
    Note: In the same way as for serialization, the synchronization library in its revision 1.1.0 allocates the buffer space in 64 bit increments. So if that helps to speed up your code, you can safely use 64 bit accesses when accessing the buffer — even when the actual size is smaller.

- A template specialization of the class `functorize_class` for `your_type`. This can also be a partial template specialization.
  This class has to provide two public member functions `functorize_inbound_signal()` and `functorize_outbound_signal()` that are characterized as following:

  `functorize_inbound_signal()`:

  - The prototype of the `functorize_inbound_signal()` function has to look more or less exactly like:

```
TFunctor_w* functorize_inbound_signal(
  sc_signal<your_type>& signal,
  int& signal_type,
  size_t& signal_type_size,
  const char*& signal_type_name
);
```

  - `signal_type` is to be set to `DFSYNC_TYPEID_UNKNOWN`.
  - `signal_type_size` is to be set to the size of a serialized value of `your_type`. Note that you cannot make use of `sizeof()` here as you are dealing with pointers in `your_type`. In case the size of the serialized value of `your_type` changes from time to time you either have to specify the largest possible size or (strongly recommended) zero. A size of zero marks dynamically sized values.
  - `signal_type_name` gets assigned some constant character string. This string can be determining by making use of `typeid(your_type).name()`. Another solution would be to assign some text defined by yourself. The latter one has the advantage that the type name will be not dependent from the compiler. This can be useful when you need to compile different simulation kernels with different compilers.
  - The `TFunctor_w*` returned by `functorize_inbound_signal()` is the pointer to the created functor. How this is done is best explained by example (see section 5.2).

  `functorize_outbound_signal()`:

  - The prototype of the `functorize_inbound_signal()` function has to look more or less exactly like:

```
TFunctor_r* functorize_outbound_signal(
  const sc_signal<your_type>& signal,
  int& signal_type,
  size_t& signal_type_size,
  const char*& signal_type_name
);
```

  - The remaining things are actually the same as for `functorize_inbound_signal()` and do not need to be discussed again.

Additional Notes:

- All code in the header file has to be included in the `dfsync` name space.
- Sometimes it might be necessary to include additional control information besides the actual signal value. In general, you have to be able to deserialize the value again in order to reconstruct the value. The string example given below does this by appending a zero at the end of the string.

That's all!

This header file has to be included by your application after the header file of the synchronization library (`systemc_sync.h`).

Well, although this appears to be quite a lot of work, the following example will show that this is not the case. Most of the work is just copy–and–paste.

5.2 The `std::string` Example

As an example for a non–trivial type, there is made available an according header file that is providing support for the class `std::string`.

This header file can be used as example for developing according code for the support of other types and is shortly discussed in this section.

Listings 5.1 and 5.2 show the code of the `functorize_string.h` header file. Now let's walk through the code.

As it should be done for all header files, the whole code is enclosed by the standard preprocessor directives (lines 18, 19, and 124). Of course, you have rename `DFSYNC_FUNCTORIZE_STRING_H` for your header file.

It is also important to include the code in the `dfsync` name space (lines 23 and 122).

The serializing class (`serialize_string`) is defined in lines 27 to 47. The class constructor just stores the specified signal pointer in the local pointer variable `my_signal`. `do_serialize()` first reads out the signal value into a temporary object `temp_value` (might be not really needed). Then the size of the current string is determined using the `length()` method. In this case 1 is added because we have to include the zero–termination. Finally, the string is converted to a NULL–terminated C–string using the `c_str()` method and transferred into the given buffer using `memcpy()` (line 39).

Because `string` contains strings of arbitrary length, a function for returning the current size of the serialized value is needed (lines 42 to 45). This function does also return one more than the actual length because we transmit a NULL–termination together with the string.

The deserialization class (`deserialize_string`, lines 51 to 65) does not differ very much from the serialization class. Fortunately, `std::string` provides an assignment operator for NULL–terminated (C–like) strings. So the new value can be directly written from the specified buffer (line 62).

```
18 #ifndef DFSYNC_FUNCTORIZE_STRING_H
19 #define DFSYNC_FUNCTORIZE_STRING_H
20
21 #include <string>
22
23 namespace dfsync {
24
25   ////////////////////////////////////////////////////////////////////////////
26   // Base class for serialization of std::string
27   class serialize_string {
28    protected:
29     const sc_signal<std::string>* my_signal;
30    public:
31     // constructor just stores the signal pointer
32     serialize_string( const sc_signal<std::string>* signal ) {
33       my_signal = signal;
34     }
35
36     void do_serialize( char* value ) {
37       std::string temp_value = my_signal->read();
38       unsigned int length = temp_value.length() + 1;
39       memcpy(value, temp_value.c_str(), length);
40     }
41
42     unsigned int return_size() {
43       // return one more because we need the zero-termination as well
44       return ( my_signal->read().length() + 1 );
45     }
46
47   };
48
49   ////////////////////////////////////////////////////////////////////////////
50   // Base class for deserialization of std::string
51   class deserialize_string {
52    protected:
53     sc_signal<std::string>* my_signal;
54
55    public:
56     // constructor just stores the signal pointer
57     deserialize_string( sc_signal<std::string>* signal ) {
58       my_signal = signal;
59     }
60
61     void do_deserialize( char* new_value ) {
62       my_signal->write( new_value );
63     }
64
65   };
```

Listing 5.1: `functorize_string.h` (Part 1)

Next comes the template specialization of the class `functorize_class` for the type `std::string`. This class provides the two member functions `functorize_inbound_signal()` and `functorize_outbound_signal()`. Both functions are almost symmetric. In lines 79/80 and 103/104 the type id is set to `DFSYNC_TYPEID_UNKNOWN` and the type size is set to 0. Especially the latter one marks the type as one with flexibly sized values.

The name of the type is set to "`c++ string`" (lines 81 and 105). As shown in the listing, an alternative would be the use of `typeid().name()`.

In line 83/84 resp. 107/108 there are declared temporary pointers. The `deserialize_object` resp. `serialize_object` is a pointer to an object of the according base class for serialization/deserialization. In line 86 resp. 110 this pointer becomes initialized by creating an according object.

In lines 89–90 resp. 112–115 the actual functors are created. The `TSpecificFunctor_w` and `TSpecific_Functor_r` template classes have to be instantiated for the class `deserialize_string` resp. `serialize_string`. The first argument for the functor instantiation is the pointer of the according `deserialize_object` resp. `serialize_object` that has been created before. The second argument is a reference to the deserialization/serialization function of the according class.

As for functorizing outbound signals, the constructor of `TSpecificFunctor_r` supports a third argument that has to be a reference of the `serialize_string` member function that is determining the current size of

```
67  ////////////////////////////////////////////////////////////////////////////////
68  // functorize_class template specialization for the type std::string
69  template <>
70  class functorize_class<std::string> {
71   public:
72    TFunctor_w* functorize_inbound_signal(
73      sc_signal<std::string>& signal,
74      int& signal_type,
75      size_t& signal_type_size,
76      const char*& signal_type_name
77      ) {
78
79      signal_type = DFSYNC_TYPEID_UNKNOWN;
80      signal_type_size = 0; // marks together with DFSYNC_TYPEID_UNKNOWN flexibly sized values
81      signal_type_name = "c++ string"; // alternatively: typeid(std::string).name();
82
83      deserialize_string* deserialize_object;
84      TSpecificFunctor_w<deserialize_string>* write_functor;
85
86      deserialize_object = new deserialize_string( &signal );
87
88      write_functor = new TSpecificFunctor_w<deserialize_string> (
89        deserialize_object, &deserialize_string::do_deserialize
90      );
91
92      return write_functor;
93
94    }
95
96    TFunctor_r* functorize_outbound_signal(
97      const sc_signal<std::string>& signal,
98      int& signal_type,
99      size_t& signal_type_size,
100     const char*& signal_type_name
101     ) {
102
103     signal_type = DFSYNC_TYPEID_UNKNOWN;
104     signal_type_size = 0; // marks together with DFSYNC_TYPEID_UNKNOWN flexibly sized values
105     signal_type_name = "c++ string"; // alternatively: typeid(std::string).name();
106
107     serialize_string* serialize_object;
108     TSpecificFunctor_r<serialize_string>* read_functor;
109
110     serialize_object = new serialize_string( &signal );
111
112     read_functor = new TSpecificFunctor_r<serialize_string>(
113       serialize_object, &serialize_string::do_serialize,
114       &serialize_string::return_size
115     );
116
117     return read_functor;
118
119   }
120 };
121
122 } // namespace dfsync
123
124 #endif // DFSYNC_FUNCTORIZE_STRING_H
```

Listing 5.2: `functorize_string.h` (Part 2)

the serialized value. As we are dealing with a type of flexible value size here, this function must be specified. In other cases it can be omitted and `NULL` is assumed in this case.

Finally, the pointers of the prepared functors are returned and the library can do its work with them.

5.3 Modification of Code for known SystemC Signal Types

Basically, it is possible to modify the code that is provided for handling SystemC types. For instance, it might be you discovered a bug in the handling of the type `sc_fix_fast` and you want to serialize according

signal type values as ASCII string instead of `double`. You can do this by removing the according template specialization and/or serialization code and replace it according your needs. However, in this particular case you must not use the original type identifier but `DFSYNC_TYPEID_UNKNOWN`. The reason for this is that the synchronization library determines the size of serialized values internally. Only when your new size is smaller or equal to the size used by the synchronization library you can continue to use the original identifier from a formal point of view.

The following table shows how the synchronization library determines the number of bytes used for various signal types. W marks the word length as it is handed over via `signal_type_size` to the library during signal attachment.

SystemC Type	Serialized Value Size (Bytes)	Comment
`sc_bit`	1	single ASCII char
`sc_logic`	1	single ASCII char
`sc_int<>`	$W \div 8$; add 1 if $W\%8 > 0$	compact integer format
`sc_uint<>`	$W \div 8$; add 1 if $W\%8 > 0$	compact integer format
`sc_bigint<>`	$W \div 8$; add 1 if $W\%8 > 0$	compact integer format
`sc_biguint<>`	$W \div 8$; add 1 if $W\%8 > 0$	compact integer format
`sc_bv<>`	$W \div 8$; add 1 if $W\%8 > 0$	compact integer
`sc_lv<>`	$W + 1$	ASCII string with NULL–Termination
`sc_fixed<>`	$W \div 8$; add 1 if $W\%8 > 0$	compact integer format
`sc_ufixed<>`	$W \div 8$; add 1 if $W\%8 > 0$	compact integer format
`sc_fixed_fast<>`	8	`double` value
`sc_ufixed_fast<>`	8	`double` value
`sc_fix`	$W \div 8$; add 1 if $W\%8 > 0$	compact integer format
`sc_ufix`	$W \div 8$; add 1 if $W\%8 > 0$	compact integer format
`sc_fix_fast`	8	`double` value
`sc_ufix_fast`	8	`double` value

Although possible as described above, it is not recommended to reuse existing type identifiers for own serialization/deserialization mechanisms. The advantage of doing so, is that the connection of executables that have been built with different template codes will fail at the very beginning during the consistency check. In case the type identifiers remain the same, the consistency check will be passed but the according deserialization codes do not match the serialization codes. This results in unpredictable and nondeterministic behavior. Therefore it is recommended to declare the type as unknown and code according type–specific information as ASCII–text into the type name. As this name can have an arbitrary length, there is plenty of space for detailed descriptions.

Chapter 6

Example Application

This chapter gives a detailed description of a small example that is provided together with the synchronization library. This is an evolved version of the example that has been discussed in [6].

In section 3.7 on page 11 there are given instructions how to compile and start the example.

6.1 Basic Model

The purpose of the exemplary model is to realize a simple counter composed of a register and an adder. Both components are located in separate simulation kernels. Figure 6.1 illustrates the exemplary system and shows how the signals are connected.

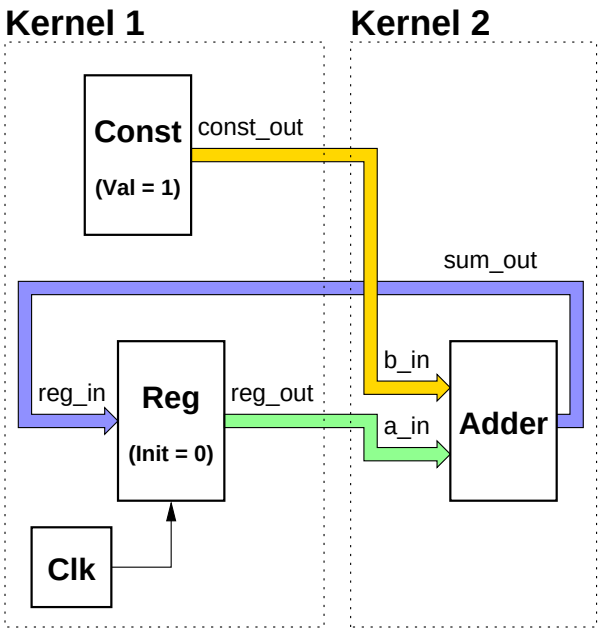

Figure 6.1: Exemplary Distributed System

Besides the register, kernel 1 does also include a constant that is to be added to the current register value as well as a clock generator.

Kernel 1 will need one outbound sync module where the signals `reg_out` and `const_out` are to be attached to. Also a single inbound sync module is needed where `reg_in` is attached to.

Say we want to clock the register with 100Mhz. So the register will change every 10ns. Accordingly, `reg_out` will have to be attached with an update cycle of 10ns. A phase shift is not needed.

`const_out` is a signal with a value that does never change. So actually this needs to be synchronized only once at the beginning. We cannot specify an infinite update cycle, but at least a rather large one — say

1 second. We could also use a small value. But keep in mind that every synchronization takes some time in the end...

Kernel 2 has one inbound sync module for the two signals `a_in` and `b_in`, and one outbound sync module for `sum_out`. `sum_out` has to be attached with an update cycle of 10ns — this has to match the frequency of the register clock. If it does not match, the simulation won't work correctly. In addition, we specify a small phase shift of 1ns. We will discuss later in section 6.4 why this is needed.

6.2 Code for Kernel 1 (Register)

6.2.1 Main Code of Kernel 1

Listing 6.1 shows the main code of kernel 1.

In line 18 the `dfsync` name space is selected. This avoids that we have to specify the `dfsync::` prefix for every name related to the synchronization library.

The three signals are declared in line 21. We use `sc_int<8>` for all signals. The clock signal is created in line 27 after there has been created an `sc_time` object representing a time of 10ns. This object will be also used later when the signal `reg_out` is attached to the outbound sync module.

In line 30 the primary synchronization library object is instantiated with the name `sync_lib`.

In line 35 the outbound sync module is instantiated and named `outbound_module`. It receives the designator `1` and will be connected to host `localhost`, port `10011` and inbound sync module with designator `1`.

In lines 40 and 41 the two outbound signals are attached to `outbound_module`. `reg_out` receives a signal designator of `1` and will be synchronized every `clock_cycle` (i.e. every 10ns, see 24). `const_out` receives signal designator `2` and will be synchronized every second.

In line 44 the inbound sync module is created with an inbound sync module designator `1`. Note that this designator does not conflict with the designator of `outbound_module` which is `1` as well!

The one and only inbound signal is attached in line 47 with a signal designator `1`. This does also not conflict with the designator `1` of the outbound signal `reg_out`. Signal designators are solely defined within the context of individual inbound and outbound sync modules. Even when there would be another inbound sync module, there could be attached another signal with designator `1`.

A register component is instantiated in lines 50 – 53. The code of this register is discussed below.

In line 56 `const_out` is initialized with `1`.

The last important call regarding the synchronization library is done in line 59. There, the connection of all modules is forced by calling the member function `connect_all()` of the library object. The argument `10010` tells the library to set up a server on TCP port 10010 where the other simulation kernel has to connect to.

Finally, the simulation is started for some time.

6.2.2 Code for the Register Component (Alternative I)

One alternative for the actual register code is shown in listing 6.2. For convenience, the actual implementation code has been directly included in the module prototype declaration.

Note: Normally one would separate prototype declaration and implementation and place the implementation into a separate C++ file. However, in this case a small exception is made in order to reduce the amount of involved files so that the focus can be set on the essential things to be demonstrated by this example.

Actually, there is nothing special with the register code as it is mostly following the coding standard for synthesizable D–FlipFlops as stated in [5] (in particular, refer to page 4-4). Though, our register writes out some messages telling us what it is doing. In addition, there is implemented a power–on reset setting the register content to zero initially. This is done by checking whether the simulation time is zero (line 18). Of course, this is not synthesizable. But one could also check the state of a reset signal there. However, this reset signal would need an additional reset–circuitry. For the sake of an as simple as possible example design this has been omitted.

```
11 #include "systemc.h"
12 #include "systemc_sync.h"
13 #include "clocked_reg.h"
14
15 int sc_main(int ac, char *av[])
16 {
17   // use dsync namespace (avoids dfsync:: prefix)
18   using namespace dfsync;
19
20   // some declarations
21   sc_signal< sc_uint<8> > reg_in, reg_out, const_out;
22
23   // our clock cycle (10ns for 100MHz)
24   sc_time clock_cycle( 10.0, SC_NS );
25
26   // create a clock (50% duty cycle, cycle according clock_cycle)
27   sc_clock clk("clk", clock_cycle, 0.5, clock_cycle);
28
29   // create synchronization library object
30   sc_dfsync sync_lib;
31
32   // create an outbound sync module with designator 1
33   // (will be connected later to inbound sync module 1 located on
34   // simulation kernel running on localhost port 10011)
35   sc_dfsync_out outbound_module = sync_lib.outbound_module( 1, "localhost", 10011, 1 );
36
37   // attach our two outbound signals and specify update cycles
38   // (the cycle for const_out can be basically infinite for this example)
39   // A phase shift is not needed for both signals.
40   outbound_module.attach( 1, reg_out, clock_cycle );
41   outbound_module.attach( 2, const_out, 1.0, SC_SEC );
42
43   // create inbound sync module (designator 1)
44   sc_dfsync_in inbound_module = sync_lib.inbound_module(1);
45
46   // attach the one and only inbound signal
47   inbound_module.attach( 1, reg_in );
48
49   // instantiate and connect register component
50   reg REG("reg");
51   REG.clock(clk);
52   REG.reg_in(reg_in);
53   REG.reg_out(reg_out);
54
55   // initialize the constant
56   const_out.write(1);
57
58   // connect all modules and listen on port 10010
59   sync_lib.connect_all(10010);
60
61   // start the simulation
62   sc_start(10000,SC_NS);
63
64   return 0;
65 }
```

Listing 6.1: `register.cpp`

6.2.3 Code for the Register Component (Alternative II)

If we are interested in a more behavioral model, we might want to use a slightly different register description as shown in listing 6.3.

This model realizes exactly the same functionality as the code shown in listing 6.2. The difference is that the register is realized as `SC_THREAD` now rather than `SC_METHOD`.

```
11 SC_MODULE( reg ) {
12   sc_in< sc_uint<8> > reg_in;
13   sc_out< sc_uint<8> > reg_out;
14   sc_in_clk clock;
15
16   void process() {
17
18     // A power-on reset (not synthesizable!)
19     if ( sc_simulation_time() == 0.0 ) {
20       cout << "Time: 0ns"
21            << " => Resetting Register to 0" << endl;
22       reg_out.write( 0 );
23     } else {
24       cout << "Time: " << sc_simulation_time() << "ns"
25            << " => Loading Register with "
26            << reg_in.read() << endl;
27
28       // Update the register content.
29       reg_out.write( reg_in.read() );
30
31     }
32   }
33
34   SC_CTOR( reg ) {
35     SC_METHOD( process );
36     sensitive_pos << clock;
37   }
38 };
```

Listing 6.2: `clocked_reg_method.h`

```
11 SC_MODULE( reg ) {
12   sc_in< sc_uint<8> > reg_in;
13   sc_out< sc_uint<8> > reg_out;
14   sc_in_clk clock;
15
16   void process() {
17
18     // Initialize the Register.
19     cout << "Time: 0ns"
20          << " => Resetting Register to 0" << endl;
21     reg_out.write( 0 );
22
23     while (1) {
24       // Wait for our clock.
25       wait();
26
27       cout << "Time: " << sc_simulation_time() << "ns"
28            << " => Loading Register with "
29            << reg_in.read() << endl;
30
31       // Update the register content.
32       reg_out.write( reg_in.read() );
33     }
34   }
35
36   SC_CTOR( reg ) {
37     SC_THREAD( process );
38     sensitive_pos << clock;
39   }
40 };
```

Listing 6.3: `clocked_reg_thread.h`

6.3 Code for Kernel 2 (Adder)

6.3.1 Main Code of Kernel 2

Listing 6.4 shows the main code of kernel 2.

```
11
12 #include "systemc.h"
13 #include "systemc_sync.h"
14 #include "adder_module.h"
15
16 int sc_main(int ac, char *av[])
17 {
18   // use dsync namespace (avoids dfsync:: prefix)
19   using namespace dfsync;
20
21   // some declarations
22   sc_signal< sc_uint<8> > a_in, b_in, sum_out;
23
24   // create synchronization library object
25   sc_dfsync sync_lib;
26
27   // create an outbound sync module with designator 1
28   // (will be connected later to inbound sync module 1 located on
29   // simulation kernel running on localhost port 10010)
30   sc_dfsync_out outbound_module = sync_lib.outbound_module( 1, "localhost", 10010, 1 );
31
32   // attach sum_out which will by synced out every 10ns with a phase shift of 1ns
33   outbound_module.attach( 1, sum_out, 10, SC_NS, 1, SC_NS );
34
35   // create an inbound sync module with designator 1
36   sc_dfsync_in inbound_module = sync_lib.inbound_module(1);
37
38   // attach inbound signals
39   inbound_module.attach( 1, a_in );
40   inbound_module.attach( 2, b_in );
41
42   // instantiate and connect the adder component
43   adder ADD("adder");
44   ADD.a(a_in);
45   ADD.b(b_in);
46   ADD.sum(sum_out);
47
48   // connect all modules and listen on port 10011
49   sync_lib.connect_all(10011);
50
51   // start the simulation
52   sc_start(10000,SC_NS);
53
54   return 0;
55 }
```

Listing 6.4: `adder.cpp`

Actually, there is not much to describe as this is comparable with the code for kernel 1.

In line 30 an outbound sync module is created an will be later connected to TCP port `10010` on host `localost` and inbound sync module `1`. Note that the port number matches this one that has been specified in the `connect_all()` call of kernel 1 (listing 6.1, line 59).

There is only one outbound signal which is attached in line 33 and receives a designator `1`. This designator is the same as the designator of the inbound signal `reg_in` that has been attached by kernel 1. Hence, both signals will be tied together. `sum_out` will be synchronized every 10ns with a small phase shift of 1ns (see later). Note that the update cycle cannot be derived from the actual clock rate that is defined by kernel 1. As a result, when the register clock changes, the update cycle of the adder output has to be changed accordingly.

The inbound sync module is created in line 36 and the two inbound signals are attached to it in lines 39 and 40. So `a_in` will be connected with `reg_out` of kernel 1 while `b_in` will be connected to `const_out`.

In lines 43 – 46 the actual adder component is instantiated and the connection of the synchronization modules is initiated in line 49. The given port number does again match this one that has been specified for the outbound sync module instantiation in kernel 1 (listing 6.1, line 35).

The simulation is started in line 52 and should last for the same time as the other kernel. In case one terminates earlier, the other kernel will become terminated automatically by the synchronization library.

6.3.2 Code for the Adder Component

Listing 6.5 shows the code of the actual adder.

```
9  SC_MODULE( adder ) {
10   sc_in< sc_uint<8> > a;
11   sc_in< sc_uint<8> > b;
12   sc_out< sc_uint<8> > sum;
13
14   void process() {
15     sum.write( a.read() + b.read() );
16   }
17
18   SC_CTOR( adder ) {
19     SC_METHOD( process );
20     sensitive << a << b;
21   }
22 };
```

Listing 6.5: `adder_module.h`

This code is as simple as one can imagine. It is a simple `SC_METHOD` that is triggered each time there is made a change on one of the inputs `a` or `b`.

6.4 A closer Look on the Timing

Figure 6.2 illustrates the behavior of all involved signals over the time.

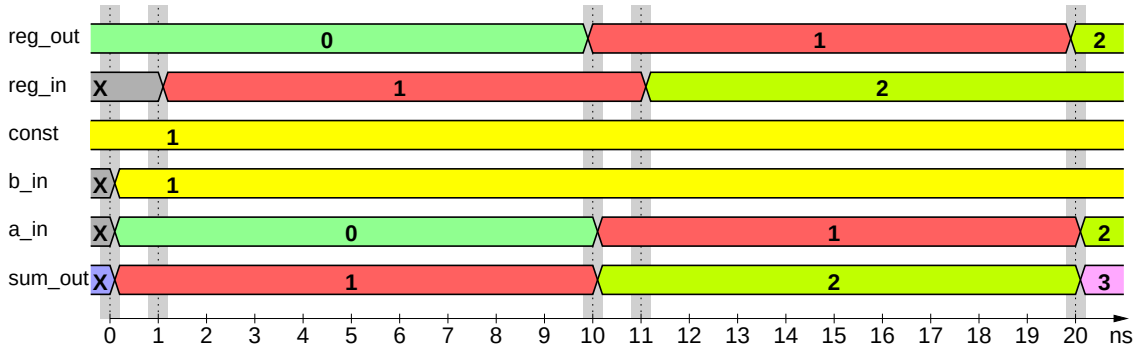

Figure 6.2: Timing of the Exemplary Distributed System

The figure shows how the values are propagated through the system. This is also illustrated by colors. The vertical dotted lines and grey bars at 0, 1, 10, etc. nano seconds mark the times when inbound and/or outbound synchronization takes place. Although the bars are graphically stretched along the time axis, they are dedicated to the according instant simulation time. This is intended for demonstrating the delta cycle behavior. This means when an outbound signal is to be synchronized, there will be sent out this value that is graphically above the dotted line. In contrast, an inbound signal will always change after the dotted line.

In particular, we can observe that `reg_out` (our actual register content) changes at rather early delta cycles. In comparison, `reg_in`, which is an inbound signal, changes rather late. The same is true in case of `a_in`. It can also be seen that the value of `a_in` is immediately following the value of `reg_out`. There is just a delay of a few delta cycles.

Well, one might assume the same behavior in case of `sum_out` and `reg_in`, which have both the same outbound/inbound signal relation as `reg_out` and `a_in`. However, things are a little bit different here. Remember that the adder component used in this example is triggered by the change of either `a_in` or `b_in`. When `a_in` changes as consequence of the inbound synchronization, the outbound synchronization has been already completed. Hence, the new value of `sum_out` won't be anymore synced out at that time but the next time by. As a result, the new value would be not available at `reg_in` right before the next clock tick is coming.

In order to cope with that situation, we have specified a phase shift for the signal `sum_out` (1ns in particular). That is, the outbound synchronization of `sum_out` (which depends directly from the inbound signal `a_in`) is delayed by 1ns. Because of this delay, the right value of `sum_out` becomes synchronized out before the next clock.

In the example that has been discussed in [6] this situation has been handled by setting the update cycle of `sum_out` to half the clock cycle. While this would still work fine here, the phase shift is more elegant. Another advantage is that `sum_out` does not need to become synchronized twice as often as actually needed from a general design point of view. This is important in view of performance considerations.

Chapter 7

Header Files provided for the Synchronization Library

7.1 Header File Hierarchy

The following list shows the header file hierarchy and illustrates header file dependencies.

```
systemc_sync.h
      type_identification.h
      parameter_definition.h
      functor_classes.h
      functorize_baseclass.h
      sc_dfsync_in.h
      sc_dfsync_out.h
      sc_dfsync.h
      functorize_flat.h
            serialize_flat.h
      functorize_sc_u_int.h
            serialize_int.h
      functorize_sc_big_u_int.h
            serialize_int.h
      functorize_sc_bit.h
            serialize_char.h
      functorize_sc_bv.h
            serialize_int.h
      functorize_sc_logic.h
            serialize_char.h
      functorize_sc_lv.h
            serialize_sc_lv.h
      functorize_sc_u_fixed.h          (only included when SC_INCLUDE_FX defined)
            serialize_int.h
      functorize_sc_u_fixed_fast.h     (only included when SC_INCLUDE_FX defined)
            serialize_sc_u_fixed_fast.h
      functorize_sc_u_fix.h            (only included when SC_INCLUDE_FX defined)
            serialize_sc_u_fix.h
      functorize_sc_u_fix_fast.h       (only included when SC_INCLUDE_FX defined)
            serialize_sc_u_fix_fast.h
      functorize_string.h
```

7.2 Header File Descriptions

The following subsections list all header files that are provided for the synchronization library in the `include` directory. Also there is given a brief description what they do contain. Note that an application has to include only the top–level header file (`systemc_sync.h`).

There is also given a short notice whether it is permissible to change parts of the header file. But note that it is actually not needed to change anything inside any header file. Nevertheless there might be some occasions where a change appears to be suitable. When changing/replacing code for known SystemC types, do not use the predefined type identifiers! Use `DFSYNC_TYPEID_UNKNOWN` unless the serialized value size remains unchanged. For details refer also to section 5.3 on page 21.

ATTENTION: Be aware of the fact that most changes mean that applications compiled with changed library headers cannot be interfaced with applications that are compiled with the original library headers. Apart from that, you can easily hurt the overall library functionality. So be very careful when applying changes!!!

7.2.1 `systemc_sync.h`

This is the top–level header file that has to be included by the application code. `systemc_sync.h` itself does only merge several other headers.

7.2.2 `type_identification.h`

This header contains some defines for different types as well as a function that is used to determine type identifiers for intrinsic C types.

If needed for whatever reason, you can add more type identifiers and expand the `determine_signal_type()` template function. Do not change any existing type identifiers!

7.2.3 `parameter_definition.h`

There are defined all parameter identifiers that have to be used when some default settings need to be changed using `set_parameter()`.

Do not change these parameter identifiers!

7.2.4 `functor_classes.h`

This header file provides the functor classes needed for signal type anonymization.

Do not change these class definitions!

7.2.5 `functorize_baseclass.h`

This header file contains the prototype definition of the template class used for “functorizing” signals that are attached to the synchronization library.

Basically, it is possible to change this prototype definition. However, this requires at the same time to adapt various other header files that make use of this class or provide template specializations.

7.2.6 `sc_dfsync_in.h`

`sc_dfsync_in.h` contains the prototype definition for the class of inbound sync modules.

If needed, the template member function `attach()` can be changed without hurting the actual library.

7.2.7 `sc_dfsync_out.h`

`sc_dfsync_out.h` contains the prototype definition for the class of outbound sync modules.

If needed, the template member functions `attach()` can be changed without hurting the actual library.

7.2.8 sc_dfsync.h

`sc_dfsync.h` contains the prototype definition for the primary library class.

There is nothing that can be changed!

7.2.9 functorize_flat.h

This header file provides the default implementations for the `functorize_class`. This header file is needed for the handling of signals with intrinsic types as well as flat (i.e. pointerless) user–defined types.

These templates can be changed, if needed.

7.2.10 functorize_sc_u_int.h

This header file provides partial template specializations of `functorize_class` for the SystemC types `sc_int<>` and `sc_uint<>`.

These template specializations can be changed, if needed.

7.2.11 functorize_sc_big_u_int.h

This header file provides partial template specializations of `functorize_class` for the SystemC types `sc_big_int<>` and `sc_big_uint<>`.

These template specializations can be changed, if needed.

7.2.12 functorize_sc_bit.h

This header file provides a template specialization of `functorize_class` for the SystemC type `sc_bit`.

This template specialization can be changed, if needed.

7.2.13 functorize_sc_bv.h

This header file provides a partial template specialization of `functorize_class` for the SystemC type `sc_bv<>`.

This template specialization can be changed, if needed.

7.2.14 functorize_sc_logic.h

This header file provides a template specialization of `functorize_class` for the SystemC type `sc_logic`.

This template specialization can be changed, if needed.

7.2.15 functorize_sc_lv.h

This header file provides a partial template specialization of `functorize_class` for the SystemC type `sc_lv<>`.

This template specialization can be changed, if needed.

7.2.16 functorize_sc_u_fixed.h

This header file provides partial template specializations of `functorize_class` for the SystemC types `sc_fixed<>` and `sc_ufixed<>`. According SystemC behavior, this file is only included when `SC_INCLUDE_FX` has been defined by the application or is specified as compiler option (`-DSC_INCLUDE_FX`).

These template specializations can be changed, if needed.

7.2.17 functorize_sc_u_fixed_fast.h

This header file provides partial template specializations of `functorize_class` for the SystemC types `sc_fixed_fast<>` and `sc_ufixed_fast<>`. According SystemC behavior, this file is only included when `SC_INCLUDE_FX` has been defined by the application or is specified as compiler option (`-DSC_INCLUDE_FX`).

These template specializations can be changed, if needed.

7.2.18 functorize_sc_u_fix.h

This header file provides template specializations of `functorize_class` for the SystemC types `sc_fix` and `sc_ufix`. According SystemC behavior, this file is only included when `SC_INCLUDE_FX` has been defined by the application or is specified as compiler option (`-DSC_INCLUDE_FX`).

These template specializations can be changed, if needed.

7.2.19 functorize_sc_u_fix_fast.h

This header file provides template specializations of `functorize_class` for the SystemC types `sc_fix_fast` and `sc_ufix_fast`. According SystemC behavior, this file is only included when `SC_INCLUDE_FX` has been defined by the application or is specified as compiler option (`-DSC_INCLUDE_FX`).

These template specializations can be changed, if needed.

7.2.20 functorize_string.h

This header file provides a template specialization of `functorize_class` for the C++ class `std::string`. At the same time, it provides serialization/deserialization classes that are unique for `std::string`.

This code is discussed in more detail in this document (see section 5.2, page 19).

These templates can be changed, if needed.

7.2.21 serialize_flat.h

This header file provides serialization/deserialization classes for intrinsic and flat (i.e. pointerless) user–defined types. It is only needed by `functorize_flat.h`.

These templates can be changed, if needed.

7.2.22 serialize_char.h

This header file provides serialization/deserialization classes for SystemC types that are serialized by making use of the `to_char()` method. This is the case for `sc_bit` and `sc_logic`. Correspondingly, this header file is needed by `functorize_sc_bit.h` and `functorize_sc_logic.h`.

These templates can be changed, if needed.

7.2.23 serialize_int.h

This header file provides serialization/deserialization classes for SystemC types that are serialized by making use of the `to_uint()` method. This is the case for `sc_int<>`, `sc_uint<>`, `sc_bigint<>`, `sc_biguint<>`, `sc_bv<>`, `sc_fixed<>`, and `sc_ufixed<>`. Correspondingly, this header file is needed by the following headers:

`functorize_sc_u_int.h`
`functorize_sc_big_u_int.h`
`functorize_sc_bv.h`
`functorize_sc_u_fixed.h`

These templates can be changed, if needed.

7.2.24 serialize_sc_lv.h

This header file provides serialization/deserialization classes for the SystemC type `sc_lv<>`. It is only needed by `functorize_sc_lv.h`.

These template classes can be changed, if needed.

7.2.25 serialize_sc_u_fixed_fast.h

This header file provides serialization/deserialization classes for the SystemC types `sc_fixed_fast<>` and `sc_ufixed_fast<>`. It is only needed by `functorize_sc_u_fixed_fast.h`.

These template classes can be changed, if needed.

7.2.26 serialize_sc_u_fix.h

This header file provides serialization/deserialization classes for the SystemC types `sc_fix` and `sc_ufix`. It is only needed by `functorize_sc_u_fix.h`.

These template classes can be changed, if needed.

7.2.27 serialize_sc_u_fix_fast.h

This header file provides serialization/deserialization classes for the SystemC types `sc_fix_fast` and `sc_ufix_fast`. It is only needed by `functorize_sc_u_fix_fast.h`.

These template classes can be changed, if needed.

Chapter 8

Library Function Reference

8.1 Library State Diagram

Figure 8.1 shows different states of the library and transitions between those states. These states are not directly visible externally.

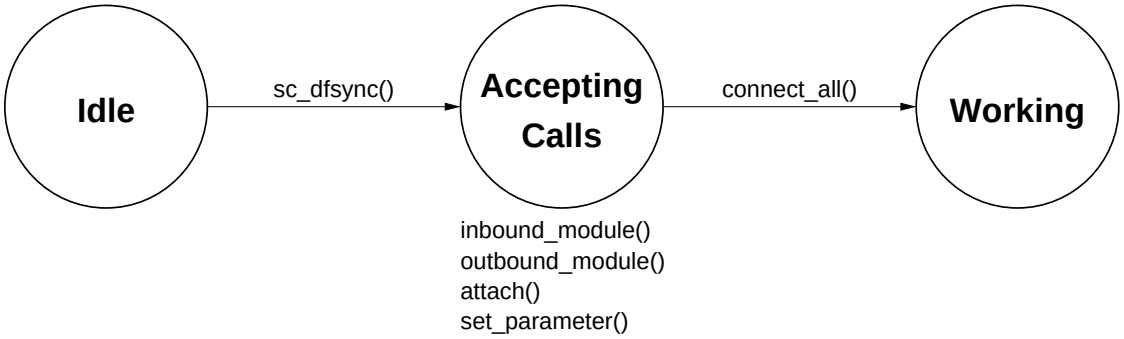

Figure 8.1: Relation between Library Calls and Library States

The function `attach()` shown there refers to member functions of both outbound and inbound sync module classes (`sc_dfsync_in::attach()` as well as `sc_dfsync_out::attach()`). Similarly, `set_parameter()` refers to `sc_dfsync::set_parameter()` as well as `sc_dfsync_out::set_parameter()`.

When a primary library object is being created by calling the constructor `sc_dfsync()` the library enters the *Accepting Calls* state. When in this state, the library accepts all calls that are needed for configuring the setup.

When `connect_all()` is called, the library goes from the *Accepting Calls* into the *Working* state. In this state there are no further library calls accepted anymore. When called anyways, the functions for creating new inbound or outbound sync modules report an error message and the application terminates. The functions for connecting the modules and changing parameters just report a warning and return an error status, but do not perform any action.

The individual classes and according member functions are described in the following sections.

8.2 class sc_dfsync

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

8.2.1 Constructor sc_dfsync()

This simple constructor is to be used for creating an instance of the synchronization library. As noted above, there can be created only one library object. The application becomes terminated when the `sc_dfsync()` contructor is called repeatedly.

Example

```
sc_dfsync sync_lib;
```

creates an object called `sync_lib` that represents the library.

8.2.2 Destructor ~sc_dfsync()

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

8.2.3 sc_dfsync_in inbound_module()

This member function has to be called in order to create 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.

Argument Restrictions/Errors

The specified designator needs to be larger than 0 and needs to be unique within the local SystemC simulation kernel. That is, there must be no other inbound sync module created before that is carrying the same designator.

Creating new inbound sync modules is not allowed after the modules have been connected.

When an error occurred, there is written out an appropriate error message and the whole application is being terminated. The reasons for an error can be an inbound sync module designator 0, a designator that is already in use, or the attempt to create a new inbound sync module after connection.

Example

```
sc_dfsync_in inbound_module1 = sync_lib.inbound_module(1);
```

creates a new inbound sync module called `inbound_module1` with designator `1`.

8.2.4 sc_dfsync_out outbound_module()

This member function has to be called in order to create 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
);
```

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

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

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

Argument Restrictions/Errors

The specified module designator needs to be larger than 0 and needs to be unique within the local SystemC simulation kernel. That is, there must be no other outbound sync module created before that is carrying the same designator.

As hostname either the IP address of the remote machine or a qualified name can be specified.

Creating new outbound sync modules is not allowed after the modules have been connected.

When an error occurred, there is written out an appropriate error message and the whole application is being terminated. The only reason for an error can be an outbound sync module designator 0, a designator that is already in use, or the attempt to create a new outbound sync module after connection.

Example

```
sc_dfsync_out outbound_module1 = sync_lib.outbound_module(1, "localhost", 10010, 1);
```

creates a new outbound sync module called `outbound_module1` with designator `1`. Later, it will be attempted to connect this module with a remote inbound sync module designated with `1` which is located in the simulation kernel running on host `localhost` and listens on TCP/IP port `10010`.

8.2.5 `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
);
```

Currently, there are only parameters of type `unsigned int`. `parameter` specifies a parameter identifier and `value` the new parameter value. For a description of individual parameters refer to section 8.3 (page 40).

`set_parameter()` returns `0` normally, or `1` when some problem occurred.

Argument Restrictions/Errors

The specified parameter needs to be a valid one, of course. Also, there might apply some restrictions for the value of certain parameters (refer to section 8.3). In addition, some or all parameters might only be changed before the synchronization library has been connected with remote simulation kernel(s).

When some violation has been detected, a warning is written out and a `1` is returned. No parameter will have changed in this case.

Example

```
sync_lib.set_parameter( Param_Relax_Delta_Cycles, 10 );
```

sets the parameter `Param_Relax_Delta_Cycles` to the value `10`. For convenience, there exist constant declarations for all parameters in order to give them human–readable synonyms.

8.2.6 `int connect_all()`

This member function is used to connect all created inbound and outbound sync modules with their remote counterparts according the information that has been specified during module creation. The function is blocking until all local inbound and outbound sync modules have been successfully connected.

After establishing the raw connection, there is carried out a signal consistency check in order to ensure that only signals of the same type are connected together. Possibly missing signals are reported as well and result in an error. Additionally, it is being checked whether an inbound sync module is able to handle the time resolution of the signals. This is required as it might be that an outbound signal changes at a resolution that is higher than the resolution of the inbound sync module resp. its simulation kernel.

Full Synopsis

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

The `portnum` argument specifies the TCP/IP port number that is 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.

The return value of this function is normally `0`. In case the library was already connected (i.e. is in working state), a warning is written out and a `1` is returned.

Argument Restrictions/Errors

The port number can be any valid number that is not already in use. All errors that appear during connection are considered fatal and result in a termination of the whole application. There are basically two kinds of error conditions: Connection–related errors (i.e. the port number is already in use) and consistency errors such as non–existing inbound sync modules, inconsistent signals on both sides, etc.

These errors are all reported in detail.

Example

```
sync_lib.connect_all(10010);
```

If there has been instantiated at least one inbound sync module, a TCP/IP socket server is set up to listen at port `10010` and is waiting for remote outbound sync modules to connect. In parallel, all instantiated outbound sync modules are connected to remote inbound sync modules according the parameters specified during outbound sync module creation.

Important Note: `connect_all()` internally performs several calculations based on timing–related parameters. In particular, this involves the simulation resolution and the default time unit. Therefore it is forbidden to change either parameter by `sc_set_time_resolution()` or `sc_set_default_time_unit()` after `connect_all()` has been called. Well, according the SystemC 2.0.1 Language Reference Manual ([2], page 416) it is not possible to adjust both parameters after an `sc_time` object has been created. While this is working well for the simulation resolution and the application terminates in case of an error, this is not the case for the default time unit. Tests have shown that the default time unit can be changed anywhere during the elaboration phase, and in particular also after the creation of the first `sc_time` object. Therefore care should be taken here because changing the default time unit after connection causes unpredictable results.

8.3 Parameters for `class sc_dfsync`

The following parameters for the synchronization library as a whole are currently defined. All parameter identifiers are of type `unsigned int`. However, there exist clear–name constants that should be used always.

8.3.1 Param_Relax_Delta_Cycles (3)

Value Type

unsigned int

Default Value

2

Purpose

This parameter defines the number of delta cycles that are let pass at the beginning of each delta cycle. This number should (must) be at least one more than the largest number of trigger indirections of any outbound signal.

As an example, when all outbound signals are triggered by a clock signal created with `sc_clock` there is one trigger indirection and the `Param_Relax_Delta_Cycles` parameter has to be set to 2 (this is also the default value). When the clock signal is additionally gated before it is feed to the registers, the trigger indirection level is two and the parameter has to be set to 3.

Note: Although it has only limited use, the library allows to set `Param_Relax_Delta_Cycles` to zero. This means that the signal values that are synced out have not been changed by the according threads or methods in the same simulation time. Effectively, this causes a phase shift of 360°. I.e. assuming a register changed in cycle N, this change is visible on the remote simulation kernel in cycle $N + 1$.

Note: In general, the use of the feature for changing the number of relaxation delta cylces is not recommended as it is prone for error. If possible, make use of the phase shift for outbound synchronization instead.

8.4 class sc_dfsync_in

This is the class of inbound sync modules. The class constructor is not a public function and cannot be called by the application.

8.4.1 int attach()

`attach()` is used for attaching a signal to the inbound sync module. `attach()` returns always `0`.

Full Synopsis

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

Argument Restrictions/Errors

The designator needs to be larger than 0 and, of course, must not be already used for another signal attached to the same inbound sync module. Additionally, `attach()` must not be called after the modules have been connected.

In case there is any error detected, an error message is printed and the application becomes terminated.

The `signal` argument can be of any kind of SystemC signal type including user user defined types.

Note: Note that in case of complex, abstract signal types or classes there have to be provided according handling templates.

Example

```
inbound_module1.attach(1, signal1);
```

attaches the signal `signal1` with designator `1` to inbound sync module `inbound_module1`.

8.5 class sc_dfsync_out

This is the class of outbound sync modules. Similarly as for `class sc_dfsync_in`, the class constructor is not a public function and cannot be called by the application.

8.5.1 int attach()

`attach()` is used for attaching a signal to the outbound sync module. `attach()` returns always `0`.

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
) {}

template <typename T>
int attach(
  const unsigned int designator, const sc_signal<T>& signal,
  const double cycle_value, const sc_time_unit cycle_time_unit,
  const double phase_shift_value, const sc_time_unit phase_shift_time_unit
) {}

template <typename T>
int attach(
  const unsigned int designator, const sc_signal<T>& signal,
  const double cycle_value, const sc_time_unit cycle_time_unit
) {}

template <typename T>
int attach(
  const unsigned int designator, const sc_signal<T>& signal,
  const sc_time cycle,
  const double phase_shift_value, const sc_time_unit phase_shift_time_unit
) {}

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

template <typename T>
int attach(
  const unsigned int designator, const sc_signal<T>& signal,
  const double cycle_value, const sc_time_unit cycle_time_unit,
  sc_time phase_shift
) {}

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

There are various kinds of `attach()` differing in the way the update cycle and the phase shift is specified. Generally, the phase shift can be omitted and 0 is assumed in this case.

Argument Restrictions/Errors

The designator needs to be larger than 0 and, of course, must not be already used for another signal attached to the same outbound sync module. The specified update cycle has to be always greater than zero. Additionally, `attach()` must not be called after the modules have been connected.

In case there is any error detected, an error message is printed and the application becomes terminated.

The `signal` argument can be of any kind of SystemC signal type including user user defined types. The update cycle can be specified either by handing over a predefined `sc_time` variable, or by specifying a value/time unit tuple (whatever appears to be more applicable). The same is true for the phase shift which can also be omitted.

Example

```
sc_time update_cycle(100.0, SC_NS);

outbound_module1.attach(1, signal1, update_cycle);
outbound_module1.attach(2, signal2, 100.0, SC_NS);
```

attaches two signals to outbound sync module `outbound_module1` that are synchronized every 100 nano seconds and have no phase shift.

8.5.2 `int set_parameter()`

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

Full Synopsis

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

Currently, there are only parameters of type `unsigned int`. `parameter` specifies a parameter identifier and `value` the new parameter value. For a description of individual parameters refer to section 8.6 (page 43).

`set_parameter()` returns `0` normally, or `1` when some problem occurred.

Argument Restrictions/Errors

The specified parameter needs to be a valid one, of course. Also, there might apply some restrictions for the value of certain parameters (refer to section 8.6). In addition, some or all parameters might only be changed before the synchronization library has been connected with remote simulation kernel(s).

When some violation has been detected, a warning is written out and a `1` is returned. No parameter will have changed in this case.

Example

```
outbound_module.set_parameter( Param_Flow_Ctrl_Max_Cycles, 100 );
```

sets the parameter `Param_Flow_Ctrl_Max_Cycles` to a value of `100`.

8.6 Parameters for `class sc_dfsync_out`

The following parameters for individual outbound sync modules are currently defined. All parameter identifiers are of type `unsigned int`. However, there exist clear–name constants that should be used always.

8.6.1 `Param_Flow_Ctrl_Max_Cycles` (1)

Value Type

`unsigned int`

Default Value

`10`

Purpose

This parameter affects the flow control mechanism and limits the amount of synchronization cycles that the outbound sync module can be ahead of the corresponding remote inbound sync module. This parameter is useful for avoiding a too fast progression of simulation kernels compared to others and is mostly only relevant for open–loop simulations.

8.6.2 Param_Send_Buffer_Size (2)

Value Type

`unsigned int`

Default Value

`4096`

Purpose

This parameter sets the buffer size (in bytes) that is used to optimize the data transfer from the outbound sync module to the remote inbound sync module.

Note: Large values do not necessarily mean better performance. The send buffer takes up signal notifications as long as it does not overflow or it becomes flushed. When we assume a very large buffer and a similarly large amount of outbound signals, it will take some time to fill the buffer. In the mean time, the remote inbound sync module has nothing to do. When the buffer size is decreased, the operation of the inbound and outbound sync module can be overlapped yielding in an increased overall throughput.

Recommended Readings

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

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

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

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

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

[6] 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

[7] Mario Trams: *A First Mature Revision of a Synchronization Library for Distributed RTL Simulation in SystemC™.*
Digital Force White Paper, November 2004. Available from
http://www.digital-force.net/publications