Simplus Quick Guide
V. 0.9

a simple C++ library for discrete event-driven simulations

Content

1 Overview....3
2 Installation....4
3 Basic Concepts....5
Modeling....5
3.1 Building and Running....6
3.2 Debugging....8
3.3 Summary....9
4 Advanced Techniques....10
4.1 Custom messages....10
4.2 Groups....10
4.3 Connectors....10
4.4 The configuration database....12
4.5 Probes....12
4.5.1 The SimProbe classes....12
4.5.2 The Probe subsystem....13
4.6 Dynamic Processes....14
4.7 Tools....14
4.7.1 The FIFO....14
4.7.2 The Alarmclock....15
4.7.3 Random Generator....15
5 Customizing....16
5.1 Overview....16
5.2 User provided Main Routine....16
5.3 Custom Debugger....16

1 Overview

Simplus is a simple C++ library for implementing object oriented, discrete event driven simulations. Main development goals are ease of use, high performance and high portability. Use cases include the simulation of telecommunication or software systems, traffic scenarios, social simulations and more.

Rather than trying to provide a full-blown simulator, the focus is more to provide a portable, general purpose framework to build more sophisticated simulators or simulation based games.

It provides an abstraction for processes, a virtual time and some functionality used in typical simulations. A command line parser allows easy script controlled execution of simulations. A build-in simple debugger allows tracking of events and more.

Simplus is a standard C++ library, no hacks, no preprocessor magic, no dependencies to external libraries or tools. It can be used with any recent compiler and any IDE. It has a good performance and a low memory footprint.

Simplus is usable and quite stable. The low version number has basically two reasons:

- it still lacks some features.
- so far, it has been tested with various versions of the Gnu C++ tool chain, on Linux and Windows XP (using the Cygwin environment) and partially with Microsoft Visual C++
- the API may change in future versions

This guide explains briefly the use of Simplus by means of a simple example. Additional information is available in the html based reference guide in doc/html/index.html and by looking into the examples.

simplus is published under GNU Lesser General Public license, please read the file *Copying.txt*.

Comments, feature requests and bug reports are highly welcome.

Hans-Peter Huth, hapehu@users.sourceforge.net

2 Installation

Prerequisites for installing and building of the library are a recent C++ compiler and either an IDE or the make utility. For creating the API reference, doxygen is required (see http://www.stack.nl/~dimitri/doxygen/, and optional graphviz from http://www.graphviz.org/). The source distribution however already includes a pre-build documentation, see docu/index.html.

Simplus is distributed in source, as a compressed tar archive. If not already done, unpack it into an arbitrary directory. If the "*make*" tool is available (e.g. on Linux or Windows/Cygwin), just type make in the top level directory. It should compile without warnings. The compiler creates the library *simplib.a* which later on can be linked against your simulations.

Building simplus with a IDE is also possible. The following applies to Microsoft Visual C++ (but may work similar in other IDEs):

- create a new "Project from existing code"
- choose the *src* and *incl* directories as source to import (do not choose the example or the test directories!)
- choose "static library (LIB) project" as project type
- eventually add the *incl* directory to the list of include directories
- build

Now you can start to develop your code which then is linked against the simplus library.

Directories *incl* and *src* hold the *simplus* sources. Directory *docs* contains this text and an API reference in html. The directory *test* contains unit tests.

A bunch of examples is in directory *example*. All files referenced in this tutorial are also located there.

Simplus is still under development and new versions may have API changes. To port code from older versions, see the Changelog and Readme.txt for hints to do this.

The source directory contains a Makefile which can be customized e.g. by adding compiler options, see inline documentations.

3 Basic Concepts

Central concept in Simplus is a processes communication using so-called 'events'. Events are time-dependent, that is, they will be delivered at a certain virtual time. Virtual time - further on simply called time - is the simulation time. This can be used to implement time dependent state machines. There are two types of events, signals and messages. A signal is an event which holds just a simple integer, a message is an event which may transport any user defined structures.

Note, all classes and global functions needed for normal use are in the *namespace simpl*. All internal things use the *namespace SimKern*, you will only need it if you intend to enhance the library itself.

Modeling

As an example, let's assume we have a process A sending signals to process B. A signal shall take t1 seconds until it reaches B. A will repeat sending in intervals of ti seconds. In Simplus, both A and B will be instances of classes derived from the *class process*. When sending events, a delivery time must be specified. The scheduler of the library takes care of proper in-time delivery of messages. To be able to receive events special call-back methods must be implemented. A *process* is a normal class which is called by the library using those signal handlers.

The source of this example is example/tutorial_1.cc. The process B is modeled using *class Receiver* which inherits from *process*:

```
1. class Receiver : public process
2. {
3.   public:
4.     void Handle_Signal( process* src, int signal ) {
5.          cout << sim_time() << endl;
6.     }
7. };
```

Important is the method *Handle_Signal()*. It is one of two possible methods for receiving events and is called by the scheduler. It receives the content of the event, which is always an integer in *Handle_Signal()*, and a pointer to the sender of the event. Both values are ignored in this example.

The sender of the message, *process A* looks like:

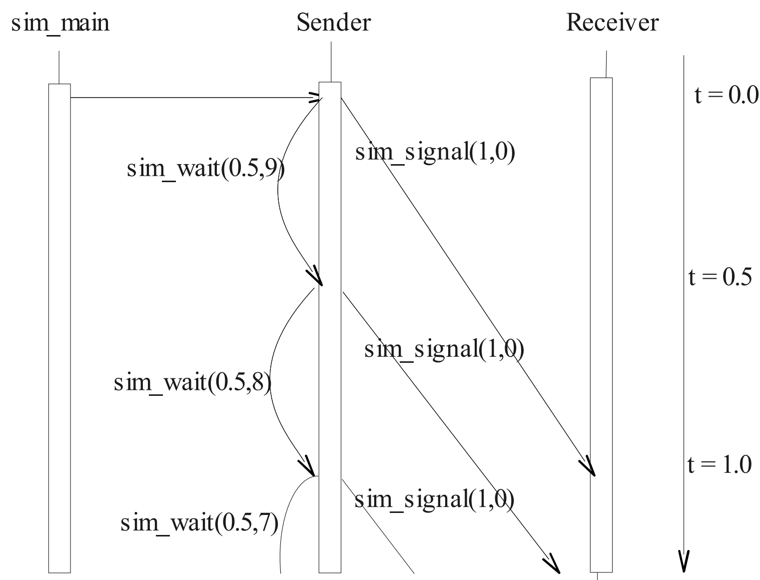
```
1. class Sender : public process
2. {
3.   public:
4.     Receiver* receiver; // holds address of the receiving process
5.     double t1; // the message delay
6.     double ti; // interval
7.
8.     void Handle_Signal( process* src, int sig ) {
9.          if ( sig ) // count-down to zero
10.              sim_wait( ti, --sig );
11.
12.         // send message to B
13.         sim_signal( receiver, t1 );
14.    }
15.};
```

We have also have a *Handle_Signal()* here. Basically A sends an event to itself to be called periodically. This is done using the *sim_wait()* method in line 10. It holds the time for delivering the event and the message itself. We decrement the message before sending and check this in line 9 to avoid an infinite simulation. Line 13 the actually sends the message to process B using t1 as a delay. Because we send an integer, we must use *sim_signal()* to send the event.

No all we need is a main program to create instances and to start things.

```
1.  int sim_main( int argc, char** argv )
2.  {
3.      Sender* A = new Sender(); // message source
4.
5.      A->receiver = new Receiver(); // the receiver
6.      A->t1 = 1; // one second message delay
7.      A->ti = 0.5; // two messages per second
8.
9.      // start A at time=0 with initial value 10
10.     sim_signal( A, 0, 10 );
11.
12.     // here we go, start the simulation
13.     sim_schedule();
14.
15.     return 0;
16. }
```

First important thing to note is, instead of using the standard main(), a *sim_main()* is used. This is because the library initializes itself before we can use it[1]. Second important thing is, you need some initial event to start the simulation. In this case it is a sgnal send in line 10. Next, *sim_schedule()* starts the message scheduler and the simulation runs until you press Control-C or until there are no more events in the system. The sequence chart below illustrates what's going on.



3.1 Building and Running

You have to compile the simulation and link it against the *simplus* library. For the example tutorial_1.cc this can be done e.g. from command line ("$>" is the prompt):

```
$>c++ tutorial_1.cc -I../incl ../simplib.a -o tutorial_1
```

The example directory contains a Makefile, so you can as well type

```
$>make tutorial_1
```

1 This behavior can be changed at build time. See ch. 4.8

The compiler now creates the executable simulation, tutorial_1 in this example. Simplest way to run this is to type:

```
$>tutorial_1
 ----- SIMPL++ Version 0.7 Nov 19 2005 12:57:47 (c) by HPH -----
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
$>
```

from command line. It starts, shows the library version and then runs the simulation. The output here shows the times when B got a message. By default a simulation runs until it is interrupted (by pressing control C), until it is terminated explicitly from the program logic or until there's no further event to process. Using Control-C is a safe operation, it will close all files. In the example above there's no new event after the 11th message, so the program terminates automatically.

However simplus provides several command line options to control the execution. There is a brief online help by adding "-h" as an option:

```
$>tutorial_1 -h
Simplus 0.91 (c) by HPH

<simulation name> [-optionen] [- args]
-cf <file> read configuration from <file>
-d         debug mode
-l <logfile>    set verbose output and use <logfile> for it.
           Use "-l stderr" to redirect to stderr.
-stat <t> [sec|min|h]     print statistics every <time> seconds or minutes or
hours.
-stop_time <t>  stop simulation after <t> seconds.
-seed <s>  initizialise random generator with <s>
-q         quit mode - surpress startup message
-v         verbose
-h         this help message.
Everything after an '-' will be given to the simp_main().
$>
```

Important options are the stop_time which forces termination at specified time, e.g.:

```
$>tutorial_1 -stop_time 2
 ----- SIMPL++ Version 0.7 Nov 19 2005 12:57:47 (c) by HPH -----
1
1.5
2
 -----  Simulation Terminated
 -----  Sim-Time: 2.5
 -----  -----  -----  -----
$>
```

A nice thing for longer simulation runs is the stat option. It will print the simulation time in the specified interval:

```
$>tutorial_1 -stop_time 2 -stat 2
 ----- SIMPL++ Version 0.7 Nov 19 2005 12:57:47 (c) by HPH -----
1
1.5
 ----- Time 2 sec -----
2
 -----  Simulation Terminated
 -----  Sim-Time: 2.5
 -----  -----  -----  -----
$>
```

Note: the stat option works by using an internal process which sends messages to itself (similar to process A in our example). So the automatic termination we used does not work and we must terminate the simulation explicitly.

Another important command line option is the "seed" option. If not given (or set to 0), *simplus* will initialize the C random generator with the current time. Thus each start will create a new sequence of random numbers. Using the seed option, you can enforce a certain initialization. Restarting an executable with the same seed will therefor create exactly the same sequence of random numbers.

3.2 Debugging

Using source level debuggers with simplus is possible, but it is difficult to follow the program logic. The simplus library however has some basic debugging support build-in.

One feature to use here is the verbose mode enabled by "-v". It prints many messages tracking all events. For example the first events look like:

```
$>tutorial_1 -v
 ----- SIMPL++ Version 0.7 Nov 19 2005 12:57:47 (c) by HPH -----
 ----- Running Simulation for 3.40282e+38 sec -----
 ----- New Process (2)  -----
 ----- New Process (3)  -----
 ----- New Process SimKernNull(4)  -----
 ----- Creating Event(1), scheduled time = 0, for Process (2) , source Process
SimKernNull(0) , Value= Int: 10
Starting Simulation
*****  Delivering  Event(1),  scheduled  time  =  0,  for  Process  (2)  ,  source
Process SimKernNull(0) , Value= Int: 10 *****
  -----  Creating  Event(2),  scheduled  time  =  0.5,  for  Process  (2)  ,  source
Process (2) , Value= Int: 9
 ----- Creating Event(3), scheduled time = 1, for Process (3) , source Process
(2) , Value= Int: 0
```

We see, three process are created. Process 2 and 3 are our Sender and Receiver, the third one is a library internal process. Internal process can be recognized by a name starting with "SimKern". Next, a event is created targeted at process 2 (which is our sender) which shall be delivered at time 0 with a value of 10. This corresponds to the *sim_signal(A, 0, 10)* statement from the *sim_main()* routine. Next, the simulation is started (from *sim_schedule()*), and the first event is delivered. The verbose output is sometimes useful, but tends to get to large in complex simulations. Using the command line option “-l <logfile>, it can be redirected to some file.

As a better method, simplus provides a simple debug mode which allows to trace messages and internal states and allows you to set breakpoints to events, messages or processes. It is enabled from command line using the "-d" option:

```
$>tutorial_1 -d
 ----- SIMPL++ Version 0.7 Nov 19 2005 12:57:47 (c) by HPH -----
 ----- Running Simulation for 3.40282e+38 sec -----
 ----- New Process (2)  -----
 ----- New Process (3)  -----
 ----- New Process SimKernNull(4)  -----
```

```
 ----- Creating Event(1), scheduled time = 0, for Process (2) , source Process
SimKernNull(0) , Value= Int: 10
Starting Simulation
1>
```

As we can see, it looks like the verbose mode, but after the start it stops in a prompt. Type "?" to get an online help. The following listing is an example on how to use this feature.

```
$>tutorial_1 -d
 ----- SIMPL++ Version 0.7 Nov 19 2005 12:57:47 (c) by HPH -----
 ----- Running Simulation for 3.40282e+38 sec -----
 ----- New Process (2)  -----
 ----- New Process (3)  -----
 ----- New Process SimKernNull(4)  -----
 ----- Creating Event(1), scheduled time = 0, for Process (2) , source Process
SimKernNull(0) , Value= Int: 10
Starting Simulation
1> st 1
 ----- Creating Event(2), scheduled time = 1, for Process SimTStop(1) with
aktive breakpoint( 1), source Process SimKernNull(0) , Value= Com: END
2> c
***** Delivering Event(1), scheduled time = 0, for Process (2) , source
Process SimKernNull(0) , Value= Int: 10 *****
 ----- Creating Event(3), scheduled time = 0.5, for Process (2) , source
Process (2) , Value= Int: 9
 ----- Creating Event(4), scheduled time = 1, for Process (3) , source Process
(2) , Value= Int: 0
***** Delivering Event(3), scheduled time = 0.5, for Process (2) , source
Process (2) , Value= Int: 9 *****
 ----- Creating Event(5), scheduled time = 1, for Process (2) , source Process
(2) , Value= Int: 8
 ----- Creating Event(6), scheduled time = 1.5, for Process (3) , source
Process (2) , Value= Int: 0
***** Delivering Event(2), scheduled time = 1, for Process SimTStop(1) with
aktive breakpoint( 1), source Process SimKernNull(0) , Value= Com: END   *****
Process Breakpoint reached.
current time 1  Process SimTStop(1)
 Breakpoint reached.
2> te
3 pending events.
Event(4), scheduled time = 1, for Process (3) , source Process (2) , Value=
Int: 0
Event(5), scheduled time = 1, for Process (2) , source Process (2) , Value=
Int: 8
Event(6), scheduled time = 1.5, for Process (3) , source Process (2) , Value=
Int: 0
3> tp
Process SimTStop(1)
Process SimKernNull(0)
Process (2)
Process (3)
4> q
Quit Simulation? (y/n)y
 ----- Simulation Ended with Code 0 at 1 sec. -----
$>
```

You can include logging statements in your code by means of the logger facility defined in *logger.h*. It gives you special streams which can be used like *cout* but will be enabled or disabled according to the flags from the program startup.

There's also a method *Info()* in the process class. Overwrite it with your debugging code, it can be called from the debugger console using the “info process” command ip <process id>

3.3 Summary

Identify active functions and actors of your system. Model them by means of a class which inherits

from *class process*. Use messages with time stamps to activate processes.

4 Advanced Techniques

4.1 Custom messages

Often the content of a message is not important or a simple integer is sufficient. For this special case, simplus calls sending integers "signal". These are the sim_signal() and Handle_Signal() methods. Signals are easy to use and have a good performance.

If the message should contain more information, you can use what is called "Message" in simplus. The message class includes a instance variable of type message_content which can be used to define custom classes encapsulating your information of arbitrary type. Sending of messages uses the *send_message()* family of functions and methods, reception of messages requires the *Handle_Message()* method of a process. There's is no automatic memory management, you must explicitly delete both message and *message_content* at appropriate places. You should overwrite the destructor. You also have to define the *Copy()* method which is called when the library has to copy messages internally, for example if you use groups.

As an example, we extend the tutorial_1.cc by sending text notes between A and B. For the full source see tutorial_2.cc .

4.2 Groups

Sending of events to many recipients can be done using a loop and an array holding the recipients. A slightly more efficient and more convenient method is to use the *class process_group* to do the job. Basically an instance of *process_group* is a process which will be addressed instead of the true receivers. It will then copy and forward the event to its members. When copying the event, the *Copy()* method of the user specified content (derived from *class message_content*) will be called. It is safe to change the members of a *process_group* while events to that group are delivered, however the change will be applied only in the next time step and performance of removing processes from a group is poor (O(N) complexity).

The behavior of group mechanism is configurable:

- filter events: if the source of an event is in the group, the *filter_self()* method can be used to specify whether the event should be delivered to the source as well or not.
- shuffle recipients: normally, each time events are delivered the list of recipients is traversed in the same order. The method *shuffle()* can enable a shuffling of the list before sending. This results in a nicer, more random behavior, however is time consuming.

Example tutorial_3.cc extends the tutorial_2.cc by sending a message to many recipients.

4.3 Connectors

If in our examples the receiving process terminates, process A will still happily send messages. This may result in crashing the program - or even worse - the simulation continuous to run but with wrong results. Another interesting situation occurs if another process - lets call it C - should replace the original receiver, but without telling the sender.

Connectors solve those problems. They are much like cables: send something into a cable without knowing who receives it at the other end. You can plug cables into a receiver and a sender. You can unplug a cable without harming the other end, and re-plug it again. Technically, it is a combination of a moderator pattern with a reference counter and an observer. Performance of the *connector* method is the about the same as the *sim_signal()* or *sim_message()* family at the cost of slightly higher memory consumption. Connectors however are much saver to use, so it recommended to use

them where ever possible. In the current version, connectors are always a 1:1 association, they can not be used with process groups.

To illustrate how to use it, see example 4, *ring.cc* which connects three processes to form a ring (fig. 4.1).

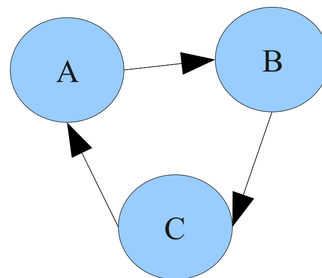


fig. 4.1: ring of processes

First we define a *class Connected_process* which includes a "plug" for sending (of type *class transmitter*) and one for receiving (of *class receiver*). The transmitter - instance variable line_out - is the end of a connector used to send messages or signals. Sending a signal or a message is done by the *send()* method of a transmitter.

The *receiver* - variable line_in - associates a *connector* with a process for receiving. Events a will still be caught by Handle_Message() respective Handle_Signal() in the receiving process.

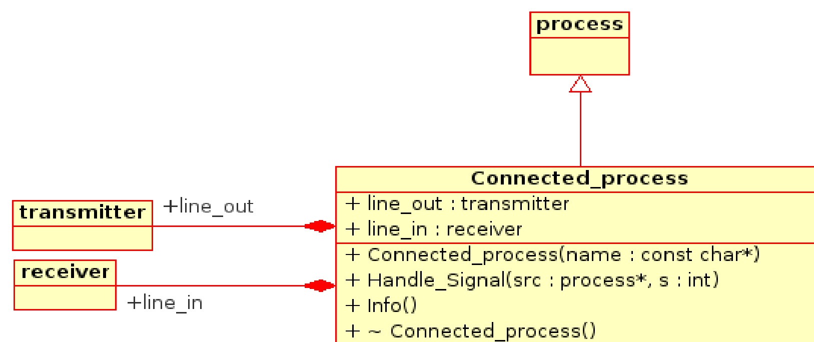


fig. 4.2: simplified UML class diagram of the ring example

Forming the ring is done by connecting a transceiver to a *receiver*. In the example this looks like:

```
Connected_process* p1 = new Connected_process("A");
Connected_process* p2 = new Connected_process("B");
Connected_process* p3 = new Connected_process("C");

// make a ring
p1->line_out.connect( p2, p2->line_in );
p2->line_out.connect( p3, p3->line_in );
p3->line_out.connect( p1, p1->line_in );
```

Deleting a *receiver* will reroute all events of the *connector* to a default sink. Deleting a *transmitter* won't affect the *connector* (there might still be old events on the fly). However connectors have smart pointer-like automatic memory management, that is, if neither *receiver* or *transmitter* is alive and if there are no more events or signals to deliver, the connector will be released automatically.

In some cases you may want to exchange events between two processes. In this case you could use two connectors, on for each direction. That's exactly what the *class transceiver* does: a bidirectional connector. A *transceiver* is a combination of a *receiver* and a *transmitter*. It has all advantages of the unidirectional connectors, but is slightly simpler to handle than two connectors. See *transceiverdemo.cc* as an example on how to use it.

4.4 The configuration database

It is possible to write a configuration file for running simulations with varying parameters. In the code you can then access the parameters from a simple database API, the *class SimConfDB*.

The configuration file format is simple:

- lines starting with '#' are comments
- all other lines must contain a pair of <key> <value> strings. Empty lines are allowed.
- key/value pairs are either separated by spaces or by a "=". In the later case the value may contain spaces as well.

The two functions *setFromConfDB()* and *getFromConfDB()* or the Find() method can be used to look up the configuration data base.

The file *confdb.cc* in the examples directory illustrates the use. To read it, start the simulation with the command line option "-cf <configfile>, in this example try *confdb -cf conf.db*

4.5 Probes

When running simulations, gathering statistics is an important job. In simplus, the so-called probes are used to gather the requested informations. The probe subsystem is build around a decorator pattern for maximum flexibility and extensibility. For simplicity and easier adaptation of older code, the *SimProbe* and *SimProbeQuick* classes are provided (which are also an example of how to use the probe subsystem to define custom probes).

4.5.1 The SimProbe classes

There are two types of probes:

- quick probes will calculate basic statistics like minimum, average, maximum on the fly. They do not write to a file and thus are very fast. They use the class *SimProbeQuick*.
- normal probes do not calculate any numbers, they just produce log files which then can be post processed, e.g. to produce curves. Use the class *SimProbe* for them. Configuration includes the file name, formats or with/without timestamp.

Both types can use data of type double. They provide behaviors when multiple values should be handled for the same time. Consider for example two processes producing two values (e.g. 2 and 4) at the same simulation time. Or consider a buffer having 5 elements at time t_0. Assume a process taking one element from this buffer at time t_1 while at the same time another processes puts an element on the buffer. When plotting the buffer size, which value to take? 5 or 6? Normal logging would write both values to the log file. The simplus probes can be configured to either process all values using the *SetAll()* method or to just take the last value. Default behavior is to use all values.

Default behavior of a probe is to do nothing, so you can include it in your classes with (almost) no performance decrease. There are two ways to enable probes:

- explicitly switch them on in the simulation using the *On()* method.
- enter the probe name in a configuration file. The value in the conf. file will be used as a file name for logging the probe or it will be ignored for Quick Probes.

A probe must have a name which identifies it in a configuration file.

Example tutorial_4.cc extends the tutorial_1.cc using probes. When started without parameters it will output nothing. Starting it with "tutorial_4 -cf tutorial_4.cfg" will read the configuration and the probes are switched on as configured.

4.5.2 The Probe subsystem

The basic idea here is to construct custom probes by means of the class *Probe* template which contains a chain of filters an an output object. A value which should be probed must be added (method Add()) to the probe object which will forward the value to the filters and eventually to the output object. The filter may alter, delay or delete the value. The output object may do a final formating and then outputs the value, e.g. to a file or GUI.

File *msgtest.cc* in the example directory demonstrates its use. The probe system is build around a decorator pattern which allows easy customization. For detailed documentation see the generated reference.

The main classes used here are:

- class *Probe*: an adapter which can be integrated into the simulation modules.
- class *ProbeFilter*, the abstract base class of filters.
- class *ProbeDecorator*, the base class of the filter decorators. There are already a number of predefined decorators:
    - *ProbeFilterCompress* which will only forward the first of potentionally many equal values
    - *ProbeLogger* which makes the probing verbose for debugging and testing
    - *ProbeFilterOnePerTime* only forwards the last of potentionally many values which are added at the same time
- class *ProbeBackend*, the base class of concrete filters to do the real output. Predefined classes are:
    - *ProbeFileBackend* is yet another abstract class. It can be used as a parent for deriving probes which use a file as output media. The file *histogram.cc* in the examples illustrates its use.
    - *ProbeOutputFile* which logs all values to a file. It uses a template-based strategy pattern which allows to specify a formating functionality and to add pre/post ambles, i.e. to write the values in a special way. As an example, the class *ProbeFormatterWithTime* is provided. It writes a time/value pair.
    - *ProbeQuick* calculates minimum, average, maximum, and sum on the fly.
- struct *ProbeValue* is the interface between Probe() and the filters.

Note, class Probe does not delete filters automatically. This allows it to use the same filter in many probes.

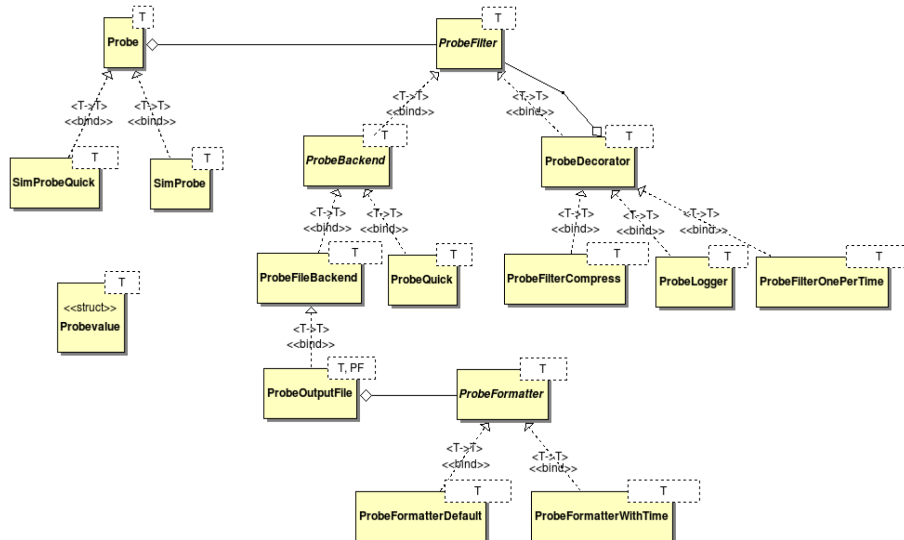


Fig. 4.3: probe class diagram

4.6 Dynamic Processes

In simplus, a process can be created dynamically and can be "killed" at any time. Example *simtest.cc* illustrates this. Creating a process is as simple. Derive your custom process class from *class process*, then use the new operator to create an instance or simply declare a member variable of that type in another class. The latter method however may cause troubles if the surrounding object gets deleted, also if it is at a global scope. Thus, using operator *new* for process creation is recommended.

Terminating a process has some pitfalls. It is possible to use the delete operator for processes, but the recommended method is calling *sim_terminate()* to kill a process. The *sim_terminate()* function guarantees that a process is deleted after all other events scheduled for the same time have been delivered. Consider for example a process C which receives a message at time T1. If another process "deletes" C at time T1, the order of those two actions can not easily be predicted. Using *sim_terminate()* to terminate C will result in delivering the message first, and then the process is deleted.

It is save to kill a non-existing process with *sim_terminate()*, i.e. because it has been deleted before.

In some cases you want to avoid that a process can be terminated by *sim_terminate()*. To do so, the method *process::Make_deletable()* sets a flag which is honored by *sim_terminate()* which does not terminate those processes.

4.7 Tools

4.7.1 The FIFO

A frequently used object in simulations is a first-in first-out buffer. *class FIFO* encapsulates a list with first-in first-out behaviour and provides additional features such as built-in probes. The program *msgtest.cc* from the examples shows how to use a fifo.

4.7.2 The Alarmclock

In many cases it is useful to implement a watchdog which triggers a call back function or functor at a specified time. Class *alarmclock* can be used for this purpose. It has a constant time for cancellation of pending events. See examples *alarm.cc* or *alarmdemo.cc*.

4.7.3 Random Generation

Unlike other simulation frameworks, *simplus* does not include its own random generators. There are several sophisticated random libraries out there, i.e. the boost library. However, *simplus* has simple wrapper functions around the standard rand() function which may be useful in some cases. See the *simplrand.h* file for more.

Note, the library will initialize the C random generator via srand() upon startup, this can be controlled from the command line (see ch. 3.1).

5 Customizing

5.1 Overview

To embed *simplus* into a larger project, you may wish to change several aspects of the default behavior. For example if you want to develop a graphical user interface for your simulator, it is possible completely avoid console I/O by specifying your own front-ends for probe output (see 4.5.2) or for the debugger.

5.2 User provided Main Routine

As we have seen before, *simplus* provides its own *main()* routine called *sim_main()*. If you want to use it as a framework for your own simulator or if you want to use other components which re-write the main routine (i.e. unit test frameworks), you can change this behavior at build time. Remove (comment) the SIM_MAIN macro in *simconfig.h*, so its NOT defined. Rebuild the whole library (use 'make lib'). In your code (i.e. in your main() routine), you now have to initialize the simulation kernel explicitly by calling sim*_init(argc,argv)*. Set *argc* and *argv* to the number of command line options and the options. After your simulation has completed, call the sim*_shutdown()* routine to clean up the simulation. See also the *simtest_main.cc* and *runagain.cc* in the example directory.

5.3 Custom Debugger

The default debugger uses a console for its user interface. It is however possible to provide a custom user interface thanks to a strategy pattern, i.e. to create a graphical interface. To do so, derive from *class SimDebuggerStrategy* and tell *simplus* to use it by calling *SimDebugger(). setStrategy(&YourDebuggerStrategy)*. See test/debugtest.cc for an example.