GOSSIP User Manual – `gossip-sim`

Marius Vollmer

Table of Contents

1 Introduction

This text documents the '`gossip-sim`' package. It is the core member of the GOSSIP suite of packages that together provide a simulation environment for synchronous, data driven models. Gossip-sim itself is the simulation engine and there are several other packages that contain a graphical editor for simulation nets, for example, or libraries of useful simulation components.

Simulations are described as *nets* of *components*. The components are connected via *signals*. A component has *ports* that can accept the signals. While executing the simulation, a component receives data at its input ports and produces new data at its output ports (which in turn travels thru a signal to the input port of another component). A component must specify how much data it expects at each of its input ports and how much it produces at each output port once enough data has arrived. These sizes must remain constant during the whole simulation. The simulation engine takes care of scheduling all components in the right order.

There is no explicit concept of *time* in the simulation model, i.e., no global clock is made available to the components and you can't specify delays in terms of seconds, for example. Rather, the components can be viewed as connected by continuous streams of data and the data itself is what drives the computation. You, as the programmer of a specific simulation net, might attach timing relationships to these streams, and might interpret them as discrete signals with different sampling frequencies, say, but the simulation engine doesn't really care. All it cares about is that it can find a way to invoke the components that all signals are in balance in the sense that exactly as much data is consumed at the end of a signal as is produced at its start.

Components are created by *instantiating blocks*. A specific block is a blueprint of all components that perform the same kind of function. While instantiating a block you can specify values for the *generics* of the block. The generics are parameters that influence the behaviour of the resulting components. Each component has its own, independent set of values for the generics of its block. Blocks can be organized into *libraries*, but they don't have to be.

There are two general kinds of blocks, and correspondingly two general kinds of components: *primitive* ones and *hierarchical* ones. A primitive component is one that actually takes part in the final simulation. It contains executable code that gets run whenever enough data has arrived at its input ports and there is room in its output buffer. A hierarchical component on the other hand contains code to instantiate sub-components inside itself. Ultimately, there will be a net of primitive components that can be executed in concert. Thus, primitive components will do the real simulation work, while hierarchical components are a tool for organizing the net of primitive components.

The data itself that is transported via the signals consists of just bits, as far as the simulation engine is concerned. It is up to the primitive components to interpret these bits. There is support for checking that connected components will interpret these bits in the same way, however. There is also support for easily writing primitive blocks that work with complex numbers, or integers of up to 32 bits.

Ports can be declared to be *multi ports*. A multi port can accept any number of signals that are then treated individually by the component. It is, for example, possible to implement a '`sum`' block with a multi port as its sole input and a normal port as its output. When instantiated, the resulting component would then query how much signals are actually connected to the input and add them all up.

The simulation system is targeted at simulations that contain relatively few components with relatively rich behaviour. In a typical use, you might have a couple of hundreds components that do things like matrix multiplications or factorizations, implement radio transmission channel models, adaptively filter a digital signal, or something on that scale. There might also be lesser components that do multiplications or additions, but it wont probably be efficient to simulate gate-level designs that consist of hundred thousands of individual nand gates.

Gossip-sim itself is implemented in two coarse layers. The upper layer that is responsible for building the simulation net from user input, checking its validity and finding a suitable schedule

for the primitive components, is written in Guile Scheme. The lower layer that is responsible for actually carrying out the simulation is written in C++.

The consequence of this is that hierarchical blocks are written in Scheme. The top-level user specification of the simulation net is written in Scheme, too, as it can be seen as a special purpose hierarchical block. Primitive blocks are written in C++ and loaded into the simulation as shared libraries.

When describing the layout of your simulation net, you are in effect writing a Scheme program and consequently you can use all features of Scheme. This gives you a considerable flexible, much more than you would get from more special purpose languages like VHDL, for example.

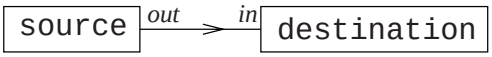
2 Examples

This chapter contains a number of runnable examples that should make you familiar with most of the concepts mentioned in the introduction. It is assumed that you are already familiar with Scheme and C++.

2.1 Example 1

Let us try to warm up with an example that shows a top-level simulation description that uses two primitive blocks.

The simulation will contain a '`source`' component that produces a stream of complex numbers and a '`destination`' component that receives the stream and prints it to the console. After printing 5 numbers, the simulation will stop. The simulation net can be illustrated like this:



Here is the actual simulation description, as you would type it into a file:

```
(use-library example *)

(signals sig)

(source :out sig)
(destination :in sig)
```

Assuming that the file is called '`example-1.sim`', you can execute the contained simulation with this shell command

```
gossip-run example-1.sim
```

You should then see something like this as the output:

```
(1,0)
(1,0)
(1,0)
(1,0)
(1,0)
destination.2: finished
```

For this to work, there must be a library called '`example`' that actually contains '`source`' and '`destination`' blocks. When gossip-sim has been installed correctly at your site, you should have this library.

You can also use '`gossip-run`' as a shell. When you invoke it without any arguments you are put into an interactive dialog where you can start simulations and dop various other things. The advantage of this mode of operation is that you do not have to sit thru the startup time of '`gossip-run`' for every simulation and you can carry state from one simulation run to the next.

When starting '`gossip-run`' as a shell,

```
gossip-run
gossip>
```

you can start simulations with the '`run`' command:

```
gossip> (run "example-1.sim")
(1,0)
(1,0)
(1,0)
(1,0)
```

```
(1,0)
destination.2: finished
```

Let us now turn to the detailed meaning of the example simulation description.

```
(use-library example *)
```

This line makes the blocks in the '`example`' library available. The blocks can be accessed simply by using their names as variable names, as we will see below.[1] The star '`*`' just means that we want to see all available blocks of the '`example`' library. You could also restrict this set by listing each individual block, or you could rename blocks while importing them. Full details on the '`use-library`' statement can be found later on in the reference part of this document. For this example, the effect of '`(use-library example *)`' is that we can make references to variables named '`source`' and '`destination`' and find the corresponding blocks in them.

```
(signals sig)
```

This statement defines a new signal, named '`sig`', that we will use to connect the two components. The `signals` statement behaves like a regular Scheme definition using `define`. Thus, you can use it only in places where a `define` would be valid.

```
(source :out sig)
```

This is a component instantiation. It looks like a regular function call, and, as far as Scheme is concerned, it actually is one. The effect is that a new component is created from the '`source`' block and inserted into the *current net*. When '`gossip-run`' starts to execute a simulation description, it creates an empty net. The executed component instantiation statements (such as the one we are discussing right now) modify this net by inserting new components into it. When the description file has been executed completely, the net is postprocessed into a runnable simulation and then started.

The arguments to the function call are interpreted as *keyword/value* pairs. In the example above, '`:out`' is a keyword (because it starts with a colon), and '`sig`' is its associated value. Together, they specify that the '`out`' port of the new '`source`' component should be connected to the '`sig`' signal.

Let me say again that the expression above is really a function call, and thus, '`source`' is really a function. You can do anything with it that you could do with any other function, like invoking it with '`apply`', passing it around and storing it in data structures.

```
(destination :in sig)
```

This is the second component instantiation. It completes the net by connecting the '`in`' port of the '`destination`' component to the '`sig`' signal.

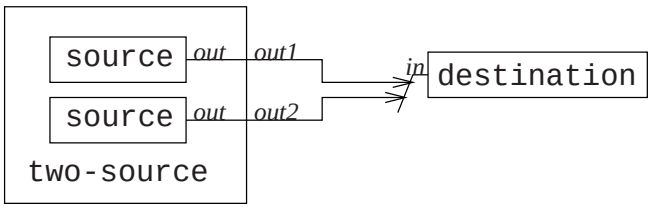
You might be wondering how Gossip knows when to stop the simulation. The default behaviour is to stop as soon as one component has *finished*. Each component has a flag that says whether it has already finished or not. The '`destination`' component raises this flag when it has printed 5 numbers. The '`source`' component never raises its flag. Thus, our sample simulation stops as soon as the '`destination`' component raises its finished flag.

You can control the condition for stopping in a very general way, Section 5.3 [Blocks and Components], page 17.

2.2 Example 2

The next example introduces hierachical components and multi-ports. It also shows how to give values to the generics of components. It extends the basic structure of the first example by connecting two sources to the destination. The two sources are instantiated by a hierarchical block called '`two-source`'. It is not imported from a library. Instead, it is defined along with the simulation description itself.

[1] This works by using the module system of Guile, in case you are interested.



Note how the '`two-source`' component contains two '`source`' components. Each '`source`' component has an output port named '`out`', with a signal connected to it. From the outside of the '`two-source`' block, these two signals are accessible on two output ports named '`out1`' and '`out2`', respectively. Now, both signals are connected to the only input port of the '`destination`' component. This port is a multi-port and thus it can accept more than one signal. This fact is expressed by this funny sloped line in the picture above.

Here is the simulation file '`example-2.sim`':

```
(use-library example *)

(define-block (two-source (> out1) (> out2))
  (source :out out1 :value 1)
  (source :out out2 :value 2))

(signals a b)

(two-source :out1 a :out2 b)
(destination :in (list a b))
```

It uses two new features, which will be explained in turn.

```
(define-block (two-source (> out1) (> out2))
  (source :out out1 :value 1)
  (source :out out2 :value 2))
```

This statement defines a new block called '`two-source`'. This is an alternative to importing blocks via libraries. The sub-form '`(two-source (> out1) (> out2))`' specifies the name and the interface of the block. In our specific case, the interface consists of two *formals*, '`(> out1)`' and '`(> out2)`'. They specify two output ports (indicated by the '`>`' symbol) named '`out1`' and '`out2`', respectively.

The remaining part of the '`define-block`' statement is its *body*. It consists of Scheme expressions that are executed when the block is instantiated. In our example, we have two expressions that each instantiate a '`source`' component. Within the body, the formals of the block are available as correspondingly named variables. For ports, they have the signal as their value that is connected to the port.

Thus, the first component instantiation expression in the body of the '`define-block`' statement creates a '`source`' component with its '`out`' port connected to the signal that is connected to the '`out1`' port of the enclosing '`two-source`' block.

You can also see how to specify values for generics. It is done with a keyword/value pair, analogous to how signals are connected to ports. The first '`source`' component gets 1.0 as the value for its '`value`' generic, the second gets 2.0. What generics exist and what they mean is of course determined by the block. The '`value`' generic in this example simply determines the value that the '`source`' component produces at its output port.

The `define-block` statement behaves like `define` as well. You can use it wherever `define` would be valid.

```
(destination :in (list a b))
```

This component instantiation connects two signals to the '`in`' port of '`destination`'. It does this by constructing a list of the two signals and uses the list as the value in the keyword/value pair. However, we could also have done it by connecting to the '`in`' port twice, like this:

```
(destination :in a :in b)
```

Note that you can only connect multiple signals to ports that are prepared for that. The '`destination`' has been specifically written to accept multiple signals at its '`in`' port. Not every port can be used in this way.

When you run this simulation, you get output like this

```
(1,0) (2,0)
(1,0) (2,0)
(1,0) (2,0)
(1,0) (2,0)
(1,0) (2,0)
destination.4: finished
```

You can see what '`destination`' does with multiple signals on the '`in`' port: it outputs one column for each of them.

2.3 Example 3

This little example shows how to run more than one simulation from one simulation file.

As we have seen, the '`source`' block has one generic named '`value`' that determines what value the source produces. We are now going to start several simulations with different values for this generic.

```
(use-library example *)

(define (simulation value)
  (signals sig)
  (source :out sig :value value)
  (destination :in sig))

(do ((val 0 (1+ val)))
    ((= val 5))
  (sim-run simulation val))
```

As you can see, the instantiation statements have been combined into a function named '`simulation`'. When this function is called, our two well-known components are added to the current net, just as it was the case in the previous examples. However, the function takes a parameter *value* that is used to set the '`value`' generic of the '`source`' component. Thus, we can easily instantiate components with different values for this '`value`' generic.

The function '`simulation`' is then used together with the '`sim-run`' function (provided by Gossip) to create and execute a net. When '`sim-run`' is called like

`(sim-run` *function args*`)`

it creates a new, empty net and makes it current. Then, while the net is still current, it calls *function*, passing it *args*. The function *function* is supposed to add new components to the net, and when it returns, the net is executed in the usual way.

In the example above, we call '`sim-run`' in a loop, thus causing it to execute several simulations.

Remember that blocks can be used like functions? This leads to an interesting variation of this example. Because blocks behave like functions, we can pass a block to '`sim-run`' and use the *args* to carry the keyword/value parameters. It would look like the following:

```
(use-library example *)

(define-block (simulation (= value))
```

```
  (signals sig)
  (source :out sig :value value)
  (destination :in sig))

(do ((val 0 (1+ val)))
    ((= val 5))
  (sim-run simulation :value val))
```

2.4 Example 4

We now get to a slightly more complicated simulation that does something vaguely useful, for a change. We will compute the first few Fibonacci numbers. How many numbers are computed can be determined via an *argument* to the simulation script. The working of this example might be hard to visualize from the code alone, so here is a picture of it:

As you can see, the central piece is an '`adder`' block with two feedback connections. An '`adder`' computes the sum of the values on its input signals and writes this sum to its output port. One of its inputs is connected to an '`impulse-source`', which writes one non-zero value on its output port, and then only zeros. It is realized by a hierarchical block defined in the simulation script. Here now is the whole script.

```
(use-library example *)

(define-block (impulse-source (= value 1.0) (> out))
  (signals a b)
  (source :out a :value value)
  (source :out b :value (- value))
  (adder :in a
 :in (sig b :chunk-delay 1)
 :out out))

(signals a b)
(impulse-source :out a)
(adder :in a
       :in (sig b :chunk-delay 1)
       :in (sig b :chunk-delay 2)
       :out b)
(destination :in b :count (arg 0 20))
```

The '`impulse-source`' block consists of two '`source`' blocks that each produce a constant value. The second '`source`' produces the negated value of the first one, and a delayed version of the second stream is added to the first. This produces the desired impulse. Incidentally, you can think of this strange arrangement as taking the derivative of a step function.

The '`impulse-source`' block already has to delay a signal. It does this by using the '`sig`' function to generate a different view of a signal:

```
(sig b :delay 1)
```

The effect of this function call is to produce a new signal that is a delayed (or otherwise modified) version of an existing signal.[2]

This example shows also how to specify a default value for a generic when defining a hierarchical block. The generic '`value`' of '`impulse-source`' gets a default value of `1.0` when no keyword/value pair has been specified for it during the instantiation.

[2] Gossip-sim still has some problems with delays. They must be a multiple of the input chunk size of the destination port that the delayed signal is connected to.

```
(define-block (impulse-source (= value 1.0) ...
```

How many numbers will be computed is determined by the '`count`' generic of the destination block. It will stop the simulation when it has received and printed '`count`' numbers. The value for '`count`' in turn is determined by calling '`arg`':

```
(arg 0 20)
```

This will return the value of the first argument to the simulation script (the first argument has index 0). When no argument is given, the value 20 is used as a default.

To print the first 25 Fibonacci numbers, you can either invoke '`gossip-run`' like

```
gossip-run example-4.sim 25
```

or use the '`run`' command interactively like

```
gossip> (run "example-4.sim" 25)
```

When arguments come from the command line, they are first passed thru the Scheme function '`read`'. This means that you need to use the regular Scheme read syntax. For example, '`123`' is a number, while '`"foo"`' is a string and '`(a b c)`' is a list of three symbols.

2.5 Example 5

After looking at the Scheme side of things, this example will finally show how to actually implement the '`source`' block in C++.

As we have seen already, the '`source`' block has one output port that produces complex numbers. The data size of this port is always 1, that is, the numbers are produced individually.[3]

The '`source`' block also has one generic of type complex. This generic determines the constant value that is produced at the output. The default value for this generic is 1.

Without further ado, here is the C++ code for '`source`':

```
#include <gossip/sim.h>

sim_generic generics[] = {
  SIM_COMPLEX_GENERIC ("value", 1.0, 0.0),
  NULL
};

sim_port outputs[] = {
  SIM_COMPLEX_PORT ("out", 1),
  NULL
};

struct source : sim_complex_comp {

  sim_complex value;

  void
  init ()
  {
    get (value, "value");
  }

  void
```

[3] You can use data sizes greater than 1, which is for example useful for blocks that do matrix operations.

```
  step (const sim_complex **in, sim_complex **out)
  {
    out[0][0] = value;
  }

};

SIM_DECLARE_BLOCK (source, generics, NULL, NULL, outputs);
```

To bring this into a form that Gossip can use, you have to build a shared library that includes the code above. Right now, you need a separate shared library for each individual block. How to exactly produce such a shared library is very platform dependent, unfortunately. On GNU/Linux, the following works, assuming the code above is contained in '`source.cc`':

```
g++ -shared -fPIC -o source.prim source.cc
```

Using '`g++`' is important so that the right C++ libraries are included. If you want to use '`libtool`' to build these shared libraries (which is actually recommended), you can find examples of its use in the '`gossip-sim`' sources itself.

After you have managed to build a working shared library, you must make it visible to Gossip so that you can use it in simulations. The direct way is to load the block by hand with a suitable '`define-block`' statement like

```
(define-block source :primitive "some/where/source.prim")
```

Another, more elegant way is to copy the shared library file into a directory where it can be automatically found by the '`use-library`' mechanism, Section 5.1 [Libraries], page 15. You can also make use of the special library called '`work`' that corresponds to all blocks in the directory that is current when the `use-library` statement is evaluated.

Now to the source of '`source`' itself.

```
#include <gossip/sim.h>
```

The header file '`gossip/sim.h`' must be included at the top of every primitive Gossip block.

```
sim_generic generics[] = {
  SIM_COMPLEX_GENERIC ("value", 1.0, 0.0),
  NULL
};
```

This array of `sim_generic` structures describes the generics of the '`source`' block. The initialization of each member should be done via pre-defined macros like `SIM_COMPLEX_GENERIC`. The concrete usage of `SIM_COMPLEX_GENERIC` above means: the type of the generic is `complex`, the name of it is `"value"`, the real part of its default value is 1.0, and the imaginary part is 0.0.

The end of the array is marked with a `NULL`. The array itself is brought into action by the `SIM_DECLARE_BLOCK` macro at the end of the code.

```
sim_port outputs[] = {
  SIM_COMPLEX_PORT ("out", 1),
  NULL
};
```

This array of `sim_port` structures describes the output port of the '`source`' block. It follows the same general principles as the `sim_generic` array above, i.e., you should initialize it by using macros and its end is marked with a `NULL`.

The `SIM_COMPLEX_PORT` macro as used above means: the type of the port is `complex`, its name is `"out"` and its default chunk size is 1.

```
struct source : sim_complex_comp {
```

Each block is represented by a new class that inherits from one of several base classes that are provided by '`gossip/sim.h`'. Which base class to choose depends on the types of the input and

output ports. For ‘source’, the right class is sim_complex_comp, because it exchanges complex values on its ports.

```
void
init ()
{
  get (value, "value");
}
```

The sim_complex_comp class provides a virtual function init that you should override to initialize new objects of your class. The ‘source’ block just uses the get function to copy the value of the generic named ‘"generic"’ into the member field value.

```
void
step (const sim_complex **in, sim_complex **out)
{
  out[0][0] = value;
}
```

This is another virtual function provided by sim_complex_comp that you should override. The step function is called whenever enough data has arrived at the input ports and the output ports can accept new data. The parameters in and out point to the input and output buffers, respectively. The length of these buffers is determined by the chunk sizes associated with the corresponding ports. The ‘source’ block has no inputs, so we can’t use the in parameter. The buffer for the only, zeroth output port can be accessed as out[0]. It has room for exactly one complex number, because the chunk size of its port has been set to 1. Thus, we just store value in out[0][0].

```
SIM_DECLARE_BLOCK (source, generics, NULL, NULL, outputs);
```

The SIM_DECLARE_BLOCK macro actually makes the new source class available to Gossip. It takes the name of the new class, and pointers to the generic array, the results array, the input port array, and the output port array.

3 The World according to Gossip

This chapter describes the simulation model of Gossip, its various building blocks, their relationships, and the technical terms used to talk about it all.

Gossip can simulate *syncronous data flow programs*. This means that it can deal with programs that can be divided into pieces that run concurrently and these pieces communicate only by exchanging streams of data at a constant rate and with a fixed pattern of connections.

In Gossip, the independent pieces of a program are termed *processes*. The entities that carry the data streams are called *signals*. These signals connect to *input ports* and *output ports* of processes. Whenever enough data is avilable at the input ports of a process, it performs its assigned computation and produces data at its output ports. These new data elements appear on the input ports of other processes (via the connected signals) and may enable these other processes to run in turn.

Exactly how much data must appear on the input ports of a process is determined by the *chunk* size of the individual input ports. The chunk size of a input port specifies the number of data elements that the process expects at that port. When a process has more than one input port, there must be enough data on all its input ports before it will run.

Likewise, the chunk size of an output port specifies how data will appear on this port when the process is run once. As an example, consider a process with only one input port and only one output port. When the input port has a chunk size of 1 and the output port has a chunk size of 2, then the process will produce two elements at its output port for every single element at its input port. You can also view the chunk sizes as indicators of the (relative) data rate at a port. The example process would have a output data rate twice that of its input data rate.

The signal connections must be made before the simulation is started and they cannot be changed during the run of a simulation. All signals must be connected to exactly one output port and at least one input port. When a signal is connected to more than one input port, all input ports see the same, effectively duplicated data stream.

Furthermore, the chunk sizes of all ports must be fixed at run-time. and the chunk sizes and connections must be chosen so that all data rates are consistent. In effect, the simulation must be able to run forever and not produce more data elements at output ports than are consumed at input ports. This does not mean that the chunk size of an output port must be equal to the chunk sizes of the input ports it is connected to, however.

You can specify a *delay* individually for each input port that a given signal is connected to. The effect of a delay is that the first data element that comes out of the output port of the signal is not the first to reach the input port. Instead, it is preceeded by a number of data elements that have been placed on the signal before the simulation is started.

You can form loops with your signal connections. However, there must be sufficient delay in the loop to allow the simulation to start. How much is sufficient depends on the chunk sizes of the processes on the loop. Gossip will tell you how much is needed when you specify too little. Of course, you can also have delay in signals that don't form loops.

Processes are created as a side effect of *instantiating blocks*. When you instantiate a block, a *component* is created. A component can contain subordinate components (in a hierachical fashion) and processes. A component that contains other components is also called a *hierarchical* componenent, while a component that only contains processes is called a primitive component.

Like processes, components have input and output ports.[1] Exactly what ports are created is determined by the *interface* of the block that is being instantiated. This interface specifies the *name* of each port, its *type*, and whether it is a *multiport*. The name is used to identify the port during instantiation and the type is used to make sure that only compatible ports are connected via signals. When instantiating a block, you must specify exactly one signal for each

[1] Actually, only components have ports. Processes just use the ports of their immediatly enclosing component.

port, unless the port is a multiport. A multiport can accept any number of signals, including zero. For each signal that is connected to a multiport, an individual, normal port is created and the signal is connected to it.

In addition to ports, the interface of a block can also contain *generics*. A generic is a parameter that influences the resulting component of instantiating the block. For example, it can influence the computations that are performed by the processes, or it can control how the sub-components of a hierarchical component are created. What exactly a generic does is part of the definition of the block. The interface of a generic can specify a default value that is used when no value is specified during instantiation.

Each component has an *exit predicate* that is used to determine when a simulation should be stopped. The exit predicate is specified during the instantiation of a block. A simulation is always constructed in such a way that there is a single top-level component. The simulation stops when the exit predicate of this top-level component becomes true. Typically, an exit predicate of a component makes use of the exit predicates of its sub-components or some other means to check whether one of its processes has finished. The evaluation of the exit predicate has to be triggered explicitly by a process.

Each component can have an *epilog*. The epilog is a piece of code that is executed when the simulation has finished. Typically, it computes *results* for the component. A result is a named value that can be retrieved from a component once the simulation has finished.

3.1 Notes

It is probably interesting to relax the requirement that chunk sizes must be constant at run-time. This will force us to support dynamic scheduling, which is probably harder to do efficiently.

As of now, you can only have non-negative amounts of delay. Maybe Gossip will be extended to allow negative delays. Such negative delays will make a number of data elements disappear at the start of a data stream before they reach their destination.

Right now, a component can either be a strictly hierarchical component without any processes, or be a primitive component with exactly one process. A primitive component is completely specified in C++, including generics, results, the process, and the epilog.

There are some plans for ‘meta simulations’. In such a simulation, the generics of a block and the results of its components are turned into input and output ports respectively. The process of the meta-component would consist of a complete sub-simulation.

4 Invoking gossip-run

The program '`gossip-run`' that you have been using in the examples is only one of several ways to start a Gossip simulation. This chapter explains these different ways in detail.

The simulator is actually an extension to *Guile*, it is not a separate program. Guile in turn is an implementation of an extended version of the programming language Scheme. That is, after '`gossip-sim`' has been added to the Guile environment, the language that you can use in that environment has been extended with Gossip specific features and you can use these features to run simulations.

4.1 From the command like

When you run the program '`gossip-run`', it does not much more than to start the Guile environment, load the Gossip extensions into it, and use the function `command-line-run` (see below) to evaluate the arguments passed on the command line.

You can use '`gossip-run`' in two modes: when you specify a script on the command line, it will run this script and then exit. When you don't specify a script, it will launch into a regular Scheme read-eval-print loop, where the Gossip extensions have been made available already.

The general form of the first mode is

`gossip-run` *script script-args...*

The first argument, *script*, is the name of a file that contains Scheme code to be executed. This code is executed in a fresh module where the Gossip functions and variables that are described in the next chapter are visible in addition to the usual Scheme features. While the code is executed, blocks are instantiated into the current net. After the script has finished, that net is simulated. The *script-args* are available to *script* via the '`args`' and '`arg`' functions, as explained below. The shell passes the *script-args* as strings to Gossip, but for '`args`' and '`arg`', these strings are filtered thru the Scheme function '`read`'. XXX - give some examples.

The program '`gossip-run`' does not do anything special or complicated, because it is not meant to be used really that much. Whenever you find yourself wanting to write shell script to invoke '`gossip-run`' repeatly with complicated arguments, you should probably just write a Scheme program that does all the complicated things and that uses the Gossip features directly.

Likewise, you should probably use '`gossip-run`' mostly in interactive mode and use the function '`run`' to load and execute your simulation scripts.

4.2 From within Guile

When you are already 'in Guile', that is, when you are writing a Scheme program or typing into a regular Scheme read-eval-print loop, you can load Gossip-sim by loading the module '`(gossip sim)`' with '`use-modules`', or any other mechanism you use for loading modules. After that, for example, you can use the Gossip function '`run`' to load and execute simulations.

run *script args...* Function

Load and execute the program in the file named by *script*, passing it *args*. *script* should be a string, while *args* can be any Scheme values. From within script *script* they can be retrieved unchanged via the functions '`arg`' and '`args`'. The function '`run`' uses '`with-args`' to arrange for this.

The code in *script* is loaded into a fresh, unnamed module where the module '`(gossip sim)`' is visible in addition to the normal Scheme definitions. Thus, all top-level definitions done by *script* are local to *script*. If you don't want this, you should be using more primitive means than '`run`' to load your Scheme code, for example plain '`load`', maybe in combination with '`with-args`' to set the arguments.

While *script* is executed, a fresh and empty net is made current. Thus, all block instantiations performed by *script* are collected into this net. When *script* has been executed completely, the net is executed as a simulation. In effect, *script* is loaded in an environment as setup by the function '`sim-run`'.

Of course, *script* can ignore the current net provided by '`run`' and use '`sim-run`' directly to have more control.

arg *index* [*default*] Function

Return the script argument indicated by *index*, which should be a non-negative integer. The first argument has index 0. When there are not enough arguments to satisfy the request, '`arg`' returns *default*. When *default* has not been specified, an error is signalled instead.

args Function

Return a list of all script arguments as specified by the most recent call to '`with-args`' or '`call-with-args`'. You should not modify this list.

with-args *args body...* Macro

Install *args* as the current list of script arguments and execute *body*. After *body* has finished, reinstall the previously active list of script arguments. *args* should evaluate to a list. The whole '`with-args`' form evaluates to the value that *body* evaluates to.

call-with-args *args proc* Function

Call *proc*, a procedure of no arguments, while *args* have been installed as the current list of script arguments. After *proc* has finished, reinstall the previously active list of script arguments and return the value returned by '`proc`'. *args* should be a list.

sim-repl Function

Launch into an interactive read-eval-print loop. The module '`guile-user`' is made the current module, the '`(gossip sim)`' module is made visible inside it and the file '`~/.gossip/simrc`' is loaded if it exists.

command-line-run *progname* [*script args...*] Function

When *script* and *args* are not specified, call '`sim-repl`'. Else, filter each *arg* thru '`read`' and call '`run`' with the result. *progname* is ignored. This function is mainly intended to be sued by the program '`gossip-run`', as explained above.

5 Constructing Simulations with Scheme

As you have seen in the examples, simulations are constructed by running a Scheme program. The program can use several features that are provided by Gossip. This chapter describes these features in detail.

5.1 Libraries

Blocks can be put into libraries and these libraries can be conveniently accessed with the '`use-library`' macro. In effect, libraries are an alternative to the normal module mechanism of Guile. Libraries are better adapted to the typical use of blocks and especially to the fact that some blocks are implemented in C++, but being in a library is not essential for a block. Thus, if you don't want to use the library mechanism for your blocks, you don't need to. Nevertheless, it is recommended that you do.

A library is simply a collection of files in the filesystem. Each file represents one block. The file contains either Scheme code that will define the block when executed (probably by using '`define-block`', below), or it will be a shared object that can be dynamically linked into the running Gossip process. These different kinds of files are distinguished by the extensions of their names. Files with an extension of '`.block`' are executed as Scheme code, and files with an extension of '`.prim`' are dynamically linked as object code. More extension might be defined in the future. The name of a block is determined by stripping the extension from its filename.

The files representing the blocks are found by looking into a sequence of directories. Blocks of a certain name will shadow blocks of the same name that are found in directories later on.

The sequence of directories that make up a library is determined by looking into all directories specified by the *library path* for directories with a name that matches the name of the library. Because blocks are either implemented in an architecture independent (Scheme) or architecture dependent way (shared objects), it is customary to have both architecture dependent and architecture independent directories on the library path.

An exception is the library with the name '`work`'. Instead of searching the library path for suitable directories, it always uses the current working directory.

sim-library-path Settable Function

Return the currently active library path as a list of strings. Each string in the list names a directory that is consulted in the way eplained above.

You can use '`set!`' to specify a new library path like so

```
(set! (sim-library-path) (cons "mydir" (sim-library-path)))
```

use-library *library import-spec...* Macro

Make the blocks in *library* available in the current module according *import-spec*.

The first argument *library* must be a symbol and names the library that is to be used. The library path as determined by '`sim-library-path`' is searched for directories matching *library*. The blocks that are found in these directories are imported as top-level bindings into the current module, as requested by the '`import-spec`'s.

An *import-spec* can be:

'`*`' All bindings of the library without renaming.

'*sym*' Only the block named *sym*, without renaming.

'(*local-name sym*)'
Only the block named *sym* but renamed to *local-name*.

‘(*prefix sym sym...*)’
: All blocks with the indicated *sym*s, prefixed with *prefix*. Note that this case is only distinguished from the previous one by having more than one *sym*.

‘(*prefix* `*`)’
: All blocks, prefixed with *prefix*.

The library is checked for new and changed blocks every time a ‘`use-library`’ statement is executed. The ‘`use-library`’ that triggered this checking will see the changes, but no previously executed ‘`use-library`’ will be affected. That is, once you have done an ‘`use-library`’ in a certain module, you don’t get to see new blocks in that module as they appear on disk, or new versions as they are updated. You need to issue the ‘`use-library`’ in the module again. This is considered a feature so that you have some control over when old blocks get replaced with new ones (and also to cut down on the amount of expensive checking that needs to be done). You can have multiple versions of a block loaded into one Gossip session at the same time without confusing it.

5.2 Signals

Signals are objects that can be used to connect two or more ports. You can construct signals with the ‘`make-signal`’ function or, more conveniently, with the ‘`signals`’ macro. You can also construct different *views* of a signal, for example a delayed version of it. These views are produced by the function ‘`sig`’ and can also be connected to ports.

signals *signal-spec...* Macro

Construct signals according to the *signal-spec*s and assign them to new Scheme variables.

The general form of a *signal-spec* is

```
(name option...)
```

It is equivalent to

```
(define name (make-signal 'name option...))
```

The abbreviated form ‘`name`’ of a *signal-spec* is equivalent to ‘`(name)`’, that is, a name without any options.

make-signal *name :bus width* Function

Construct a new signal or a list of signals with name *name* and return it. When *name* is omitted, the signal has no name. When *width* is specified, it must be a non-negative integer, and a list of *width* single signals is constructed. When *width* is omitted, a lone signal will be returned.

sig *signal :delay delay :chunk-delay chunk-delay* Function

Construct a delayed version of *signal*, which must be a signal or a list of signals.

The actual delay is determined by the sum of the delays specified by *delay* and *chunk-delay*. The first, *delay*, specifies the delay in terms of single elements that are transported over the signals. The other one, *chunk-delay*, works in terms of the chunk size of the input port that the delayed signal will be connected to. For example, if a signal transports complex values and the delayed version is connected to an input port that consumes three complex values in each cycle, specifying ‘`:delay 1`’ will delay by one complex value, while ‘`:chunk-delay 1`’ will delay by three complex values.

5.3 Blocks and Components

You can use the function '`make-block`' to create either a strictly *hierarchical block* or a *primitive block*. You can make the block definition more convenient by using one of the macros '`block-lambda`' and '`define-block`'.

Once a block object has been created by one of the methods above, it can be treated like a procedure object and applied to arguments. Such an application of a block object will instantiate a new component into the current net.

For primitive blocks, the instantiation will carry out all the magic needed to make the process defined by the C++ code appear in the current net. For hierarchical blocks, the instantiation procedure has been specified when the block was defined. The procedure will receive the filtered arguments that the block object has been applied to to initiate the instantiation.

This instantiation procedure can instantiate sub-components, and it can set the exit predicate and epilog procedure of the created component.

The instantiation makes use of the concept of a *current component*. Block instantiations always add the resulting components to the current component and functions like '`set-exit-predicate`' affect the current component. The top-most component is constructed by '`sim-run`' and is current while the setup function of '`sim-run`' is called.

make-block *:name name :primitive shared-object* Function
make-block *:name name :interface interface :instfunc instfunc* Function

Create a new block with name *name* (a string) and return it.

When *shared-object* is specified, it will be dynamically linked into the running Guile process and the interface and instantiation function of the block will be automatically derived from it. Chapter 6 [Writing Primitive Blocks in C++], page 21 for how to create such shared objects.

When *interface* and *instfunc* are specified, they define the interface and instantiation function of the block directly. The *interface* should be a list of *formals* and each formal is a list of the form

(*kind name* [*default*])

where *kind* (a symbol) determines whether the formal describes what kind formal it is (i.e. a port or a generic), *name* (a symbol) is the name of the formal. The following table shows the possible values for *kind*.

`<`	input port
`<<`	input multiport
`>`	output port
`>>`	output multiport
`=`	generic

For formals that describe a generic, *default* is the default value for the generic when it is not given a value during instantiation.

The function *instfunc* will be called when the block is instantiated. it will receive one argument for each formal in *interface*, in the same order as the formals appear in *interface*. For formals that are ports, the argument will be a list of signals that should be connected to the port. When the port is not a multiport, the list is guaranteed to contain exactly one signal. For formals that are generics, the argument will be the value given to that generic, or the default value from the formal.

block-lambda *name :primitive shared-object* Macro
block-lambda *name :interface interface body...* Macro
block-lambda (*name formal ...*) *body...* Macro

This macro expands into a call to '`make-block`'.

The first form is equivalent to

```
(make-block :name "name" :primitive shared-object)
```

Note that *name* is converted to a string for make-block.

The second form is equivalent to

```
(make-block :name "name" :interface 'interface
            :instfunc (lambda instargs body...))
```

where *instargs* is a list of the names of all formals of *interface*, i.e., the result of '`(map cadr 'interface)`'. Note that *interface* is not evaluated.

The third form is equivalent to

```
(block-lambda name :interface (formal ...) body...)
```

that is, it is just a more compact version of the second form.

define-block *name :primitive shared-object* Macro
define-block *name :interface interface body...* Macro
define-block (*name formal...*) *body...* Macro

The *define-block* macro is a combination of *define* and *block-lambda*, in the following hopefully obvious ways

```
(define name (block-lambda name :primitive shared-object))
(define name (block-lambda name :interface interface
                                body))
(define name (block-lambda (name formal ...) body...))
```

block :key value ... Function

When an object that has been returned from '`make-block`' is applied to arguments, it will create a new component.

The new component is added as a child of the *current component*, is in turn made the current component and the instantiation function of *block* is called. After the instantiation function has returned, the current component is reverted to its previous value.

The *key*, *value* pairs determine the arguments to the instantiation function. Each *key* must correspond to a name of a formal of the block and *value* will be passed to the instantiation function.

In addition to keys that correspond to the formals of the interface of *block*, you can also use the key '`:name`' to give a name to the newly created component. The name must be a string and can be used with '`find-comp`'. When no name is given, a default name is constructed from the name of the block and a sequence number. Names must be unique among all children of a given component.

The newly created component is returned as the value of the function call.

set-exit-predicate *function* Function

Set the exit predicate of the current component to *function*, a function with no arguments. The simulation stops when the exit predicate of the top-most component returns true. The exit predicates are only evaluated when a process explicitely requests it.

The default exit predicate of a component returns true when any of the exit predicates of the children of the component returns true.

While *function* is running, its component is again the current component. This is useful when you want to use '`find-comp`', for example.

finished? *comp* Function
Evaluate the exit predicate of *comp*.

find-comp *name* Function
Search the children of the current component for a component named *name*, a string. Signal an error when it is not found.

exit-expression *exit-expr* Macro
Set the exit predicate of the current component by translating *exit-expr* into a function and passing it to '`set-exit-predicate`'. The form *exit-expr* is translated into a Scheme expression *scheme-expr* according to the following rules

```
(and exit-expr ...)  =>  (and scheme-expr ...)
(or exit-expr ...)   =>  (or scheme-expr ...)
symbol                =>  (finished? symbol)
string                =>  (finished? (find-comp string))
```

and the whole '`exit-expression`' form is translated as

```
(set-exit-predicate (lambda () scheme-expr))
```

set-epilog *function* Function
Set the epilog function of the current component to *function*, a function of no arguments. The epilog function of all components is run when the simulation has stopped. The epilog functions of all children of a component are run before the epilog function of the component itself. While an epilog function runs, its component is the current component. The return value of the epilog function of the top-most component is the result of the corrsponding '`sim-run`' call, but all other return values are ignored.

epilog *body...* Macro
Set the epilog function of the current component like so

```
(set-epilog (lambda () body...))
```

results *:key value ...* Macro
Set the results of the current component according the the *key*, *value* pairs. For each pair, the result named by *key* is set to *value*. These results can be retrieved with the function '`result`'.

result *comp res* Function
Retrieve the result named by *res*, a string, from the component *comp*. When *comp* is a string, '`find-comp`' is used to find the corresponding component.

5.4 Simulation control

A simulation consists of a call to '`sim-run`'. The simulation is constructed, executed and the results are gathered under the control of '`sim-run`'.

sim-run *setup args ...* Function
Run a simulation. A new empty component is created and made current. The component is of a special type and is also installed as the current net. The function *setup* is called, passing it the *args*. It is expected that *setup* will instantiate new components into the top-level component and that it will set the exit predicate and epilog functions, if it wants to. After *setup* has returned, the instantiated components are post processed into a runnable simulation, and this simulation is then executed. When the exit predicate of the top-level component becomes true, the simulation is stopped and all epilog functions of the components are called bottom up. The return value of the epilog function of the top-level component is returned as the value of '`sim-run`'.

set-verbose *bool* Function

Set the verboseness of the current simulation to *bool*. When *bool* is true, a lot of information about the constructed simulation net and the schedule is printed prior to executing the simulation.

set-schedule-combiner *symbol* Function

Specify the combiner function of the current net. The combiner function determines in what way the static schedule will be found. For a given simulation, one combiner might produce a better schedule than others (in terms of buffer memory required and activity distribution), but all will yield correct schedules. Normally and ideally, you should not have a need for this option.

The available values for *symbol* are

'`simple`' The default combiner, which combines from the start.

6 Writing Primitive Blocks in C++

Writing blocks in C++ is right now the only way to create processes. The interface of the block is specified by filling out some information structures, and the executable code of the block is defined by deriving a new class from a predefined class and overwriting some virtual functions of that base class.

You need to include the header file '`<gossip/sim.h>`' in your programs to access the discussed features. The final block program must be a shared library that can be dynamically linked on your platform. As of now, we leave the details of this to you but the goal is of course to eventually use libtool or some derived mechanism.

The typical layout of a C++ source file that defines a block is

```
#include <gossip/sim.h>

sim_generic generics[] = {
  generic declarations,
  NULL
};

sim_result results[] = {
  result declarations,
  NULL
};

sim_port inputs[] = {
  input port declaration,
  NULL
};

sim_port outputs[] = {
  output port declarations,
  NULL
};

struct component : sim_comp
{

  void
  init ()
  {
    initialization code
  }

  void
  step (const sim_data **in, sim_data **out)
  {
    process code
  }

  void
  epilog ()
  {
```

```
    termination code
  }
};

SIM_DECLARE_BLOCK (component, generics, results, inputs, outputs);
```

You can leave out the things that you don't need.

6.1 Interfaces in C++

The interface definition of a C++ block consists of declarations for generics, results, input ports, and output ports. You have to provide a table for any of these categories that you want to use. The table for the generics consists of entries of the type '`sim_generic`', the result table is a vector of '`sim_result`' objects and the input and output port tables have entries of type '`sim_port`'. Each of '`sim_egneric`', '`sim_result`' and '`sim_port`' is a simple struct that you need to initialize.

The initialization is done by using macros that will expand into an initializer for such a struct. For example,

```
SIM_GENERIC ("foo")   =>   { "foo" }
```

but you are not supposed to know anything about the precise layout of any of the '`sim_generic`', '`sim_result`' and '`sim_port`' structures.

The tables are brought together by the '`SIM_DECLARE_BLOCK`' macro.

SIM_DECLARE_BLOCK (*component, sim_generic *generics, sim_result *results, sim_port *inputs, sim_port *outputs*) Macro

This macro expands into the necessary magic code to register *component* with Gossip. The first argument, *component*, is the name of the new type that implements the block and its only process, Section 6.2 [Processes in C++], page 23. The rest are the tables for the interface. You can use '`NULL`' for an empty table.

6.1.1 Declaration Macros for Generics

There is only one macro for declaring a generic. Generics itself do not specify a type or whether they have a default value. These issues are handled by the '`get`' functions, below.

SIM_GENERIC (*char *name*) Macro

Declare a generic of name *name*. You can retrieve the value of this generic with one of the '`get`' functions in your '`init`' function.

6.1.2 Declaration Macros for Results

SIM_RESULT (*char *name*) Macro

Declare a result of name *name*. You can set results with one of the '`set`' functions in your '`epilog`' function.

6.1.3 Declaration Macros for Ports

In addition to the general port declaration macros '`SIM_PORT`' and '`SIM_MULTI_PORT`', there are also macros to declare types of a specific type.

SIM_PORT (*char *name, type, size_t chunk*) Macro
Declare a port of name *name*, of type *type*, with an initial chunk size of *chunk*. The *type* should be a C++ identifier that names a valid type, for example '`int`' or some typedefed name. Data will be exchanged on this port in units that are a multiple of `sizeof(`*type*`)`.

SIM_MULTI_PORT (*char *name, type, size_t chunk*) Macro
Analogous to '`SIM_PORT`', but declare a multiport. Only the last port in a table can be a multiport.

SIM_INT_PORT (*char *name, size_t chunk*) Macro
Declare a port with name *name*, type '`"sim_int"`' and initial chunk size *chunk*.

SIM_MULTI_INT_PORT (*char *name, size_t chunk*) Macro
Declare a multiport with name *name*, type '`"sim_int"`' and initial chunk size *chunk*.

SIM_COMPLEX_PORT (*char *name, size_t chunk*) Macro
Declare a port with name *name*, type '`"sim_complex"`' and initial chunk size *chunk*.

SIM_MULTI_COMPLEX_PORT (*char *name, size_t chunk*) Macro
Declare a multiport with name *name*, type '`"sim_complex"`' and initial chunk size *chunk*.

6.2 Processes in C++

Blocks that are written to the C++ interface of '`gossip-sim`' always instantiate exactly one process and cannot instantiate any sub-components. To define this process, you derive a new class from the existing class '`sim_comp`' and overwrite some virtual functions of '`sim_comp`'. Inside your versions of these virtual functions, you can use other functions provided by '`sim_comp`' to retrieve the values of generics, for example, or set the chunk sizes of the ports of the component.

You can overwrite '`sim_comp::init`' and '`sim_comp::epilog`' to define the initialization and termination actions of your block and you must overwrite '`sim_comp::step`' to define the process code.

`virtual void` **init** () Overwritable Method on `sim_comp`
This virtual function is called when the block defined by the '`this`' object is instantiated. Typically, you will use one of the '`get`' functions to retrieve the values of the generics of your block and maybe call '`set_in_chunk`' or '`set_out_chunk`' to adjust the chunk sizes of some ports.

`virtual void` **epilog** () Overwritable Method on `sim_comp`
This virtual function is called when the simulation has stopped. Referring to the general description of the simulation model, this is the epilog function of the component. Typically, you will use one of the '`set`' functions or some other means to record the results of the component, if there are any.

`virtual void` **step** (`const sim_data **in, sim_data **out`) Overwritable Method on `sim_comp`
This virtual function defines the process of the component. It is called whenever enough data is available at the input ports and the output ports can accept enough data.

The data at the input ports is delivered to you as a vector of pointers to '`sim_data`' objects, in the parameter *in*. There is one pointer per connected signal. As long as there

is no input port that is a multiport, there is exactly one signal per port. Then the vector of pointers corresponds directly to the vector '`sim_port`' structures used to declare the interface of this block. That is, '`in[0]`' points to the data for the first port declared, '`in[1]`' points to the data for the second port, and so on.

When there is a multiport among the input ports, it must be the last one (as a requirement). Say, the last port has index *n*. Then the ports 0 to *n*-1 are handled as described above, that is, for $i = 0$ to *n-1*, '`in[`*i*`]`' corresponds to '`inputs[`*i*`]`'. The signals connected to the multiport correspond to entries in *in* with indices greater than or equal to *n*. When there are *m* signals connected to the multiport, their data can be found in '`in[n]`' to '`in[n+m-1]`'.

The length of the vector *in* can be determined by calling '`get_n_in`'. This is also the only way to find out how many signals have been connected to the multiport, if there is one.

Each pointer in *in* points to objects of type '`sim_data`'. This type is of no interest, because the first thing you need to do is to cast the pointer to the right type for the port. The right type is a pointer to the type as used in the corresponding '`SIM_PORT`' declaration. For example, if you declared port 0 to be of type '`int`', you would typically do something like this at the start of '`step`':

```
const int *in0 = (const int *)in[0];
```

Each pointer in *in* points to as many objects of the right type as specified by the chunk size of the corresponding port. The chunk size has either been set by calling '`set_in_chunk`' in '`init`', or, if no call to '`set_in_chunk`' has occured, it defaults to the value in the corresponding '`SIM_PORT`' declaration.

Output ports are handled in exactly the same way as input ports, substituting *out* for *in*, '`get_n_out`' for '`get_n_in`', and '`set_out_chunk`' for '`set_in_chunk`'. Each pointer in *out* points to enough memory to hold as many objects of the right type as specified by the chunk size of the corresponding port. You are supposed to write the computed data for the output ports into this memory.

You cannot change the chunk size from within '`step`', only from '`init`'.

`virtual void` **step** (`int steps, const sim_data **in, sim_data **out`) Overwritable Method on `sim_comp`

This variant of the '`step`' function has essentially the same function, but for very simple computations that only require as many time as a few function call overheads, it might speed up the simulation significantly to use this version of '`step`' instead.

The additional parameter *steps* specifies how many times the basic computation of the process should be performed. That is, the input ports have *STEPS* times as much data outstanding, and the process should produce *STEPS* times as much data for the output ports, as for the normal variant of the '`step`' function.

The following functions provide services that you can use in your '`init`', '`epilog`', or '`step`' functions.

`const char *` **get_name** () Method on `sim_comp`

Return the name of the component. You should prefix all your messages to '`stdout`' or '`stderr`' with this name to make it easy to distinguish messages from different components. Using the name of the block is not sufficient for this, because there can be many components that have been instantiated by the same block.

`int` **get_n_in** () Method on `sim_comp`

`int` **get_n_out** () Method on `sim_comp`

Return the number of connected input or output signals, respectively. This is the size of the *in* (or '`out`') vector of the '`step`' functions.

`bool` **get** (`int &var, const char *name, int def`) Method on `sim_comp`
`bool` **get** (`double &var, const char *name, double def`) Method on `sim_comp`
`bool` **get** (`sim_complex &var, const char *name, sim_complex def`) Method on `sim_comp`
`bool` **get** (`sim_complex *&var, int &len, const char *name, sim_complex *def, int def_len`) Method on `sim_comp`

Store the value of the generic named *name* in *var*. When no value for the generic *name* has been specified during instantiation, use *def* instead and return '`false`'. When *def* has not been used to provide a default value, return *true*.

The variants differ only in the type of *var*. The type '`sim_complex`' is for complex values where real and imaginary part are each of type '`double`'. The variants where *var* is a pointer and an additional *len* parameter is present, convert vectors and return a pointer to a new array in *var* and the length of the new array in *len*. You should not free the memory of this array, it is freed automatically when the component is destroyed.

The list of '`get`' methods will be extended in the future.

`void` **set** (`const char *name, int val`) Method on `sim_comp`
`void` **set** (`const char *name, double val`) Method on `sim_comp`
`void` **set** (`const char *name, sim_complex val`) Method on `sim_comp`

Store *val* as the result named *name*.

`void` **set_in_chunk** (`int input_id, size_t chunk`) Method on `sim_comp`

Set the chunk size of the input port with index *input_id* to *chunk*. This can only be called from '`init`'.

`void` **set_out_chunk** (`int output_id, size_t chunk`) Method on `sim_comp`

Set the chunk size of the input port with index *output_id* to *chunk*. This can only be called from '`init`'.

`void` **finish** () Method on `sim_comp`

Signal that this component would like the simulation to stop. The finished flag of the component is raised and the exit predicate of the simulation is evaluated. The simulation is only stopped when the exit predicate returns true, so you should be prepared to carry on with the simulation.

You can call '`finish`' from '`init`'. In that case, the simulation will not start. When you call '`finish`' from '`step`' you must nevertheless compute the right values for the output ports.

The function '`finish`' will only evaluate the exit predicate the first time it is called for a given component.

6.3 More Convenience

You might have noticed the need for the awkward casts in '`step`' to get the types right. For some restricted cases, there is also a more convenient way. When all of your input ports have the same type, and all of your output ports have the same type (but not necessarily the same as the input ports), then you can get away with more elegant code.

Currently this only works when both of the input and output types are either '`sim_complex`' or '`sim_int`'. The type '`sim_complex`' is a complex value where both real and imaginary part are of type '`double`'. The type '`sim_int`' is a signed integer of at least 32 bits. In the future, we will also provide a method to extend this support to other types.

When your block meets the above conditions, you can derive your new class from some other predefined class than '`sim_comp`'. This class works just like '`sim_comp`', but has '`step`'

functions with the right parameter types. For example, the class '`sim_complex_comp`' works for components that have only '`sim_complex`' ports and defines '`step`' function with the following signatures:

```
void step (const sim_complex **in, sim_complex **out)
void step (int steps, const sim_complex **in, sim_complex **out)
```

Refer to the following table to find out which class to use instead if '`sim_comp`'.

Input port type	Output port type	Class
`sim_int`	`sim_int`	`sim_int_comp`
`sim_complex`	`sim_complex`	`sim_complex_comp`
`sim_int`	`sim_complex`	`sim_int_complex_comp`
`sim_complex`	`sim_int`	`sim_complex_int_comp`

Index