Getting Started with FAMEbuilder

The BARDD Research Group

Getting Started with FAMEbuilder

by The BARDD Research Group

Version 1.0 Edition

Table of Contents

List of Figures

Preface

This book provides an entry point into FAMEbuilder, guiding users through installation and some primitive use of the system.

Typographical conventions used in this book

The following typefaces are used in this book:

- Normal text
- *Emphasized text*
- `The name of a program variable`
- `The name of a constant`
- **The name of a SystemC module**
- **The name of a package**
- *The name of a module parameter*
- **The name of a module port**
- `Literal text`
- *`Text the user replaces`*
- `The name of a file`
- The name of an environment variable
- *The first occurrence of a term*

Chapter 1. Installation

This chapter will walk you through the setup and installation of FAMEbuilder. FAMEbuilder requires that you install some packages that may not currently exist on your machine. This section contains details on what must be done.

Getting FAMEbuilder

FAMEbuilder is available for download from the BARDD Research Group website. Specifically, information about FAMEbuilder can be found at http://bardd.ee.byu.edu/Software/FAMEbuilder. Follow links on the website to obtain a `.tar.gz` archive of the distribution. The file will be named `FAMEbuilder_bundle-`*`version`*`.tar.gz`. At the time this guide was written, the latest available version is 1.0.

Software prerequisites

In order to build and run FAMEbuilder from the source distribution provided, your system must have certain basic programs. In addition to a working C++ compiler and associated libraries, you will need a working installation of Perl, Python, and Xilinx ISE. The distribution was tested using RHEL 6.6, however it should work with most Linux distributions. The following sections summarize what packages are known to be needed, which versions have been tested, where they are available, and operating system-specific notes.

g++

Version. FAMEbuilder has been tested with g++ version 4.4.7, however the exact version of g++ is unlikely to be necessary. Other C++98 compliant compilers may be acceptable as well.

Availability. gcc/g++ is available from the GNU website (http://www.gnu.org). gcc/g++ is available in any Linux distribution, but may not necessarily be installed by default.

Perl

Version. To use FAMEbuilder, Perl version $\geqslant 5$ is necessary. FAMEbuilder has been tested using Perl 5.10.1.

Availability. Perl is available from the Perl website (http://www.perl.com). Perl is also part of most Linux distributions.

Python

Version. To use FAMEbuilder, Python version $\geqslant 2.6$ is necessary. FAMEbuilder has been tested using Python 2.6.6.

Availability. Python is available from the Python website (http://www.python.org). Python is part of most Linux distributions.

Xilinx ISE

Version. FAMEbuilder has been tested using ISE 14.7. Earlier versions are unlikely to work correctly, though with some modifications they may be made to work. Please report any ISE version issues to `famebuilder-users@googlegroups.com`

Availability. ISE is available from the Xilinx website (http://www.xilinx.com/support/download.html).

Operating system notes

Red Hat Enterprise Linux

FAMEbuilder has been installed successfully on RHEL systems.

Installing the FAMEbuilder bundle

If you obtained FAMEbuilder in bundle form, the bundle contains FAMEbuilder and associated packages from the BARDD Research Group necessary to use FAMEbuilder. In addition it provides a script for easy installation. To install FAMEbuilder from the bundle, follow these steps:

1. Unpack the archive using `tar`. Type **`tar xvzf FAMEbuilder_bundle-`*`version`*`.tar.gz`**.
2. Change into the `famebuilder` directory by typing **`cd famebuilder`**
3. Install FAMEbuilder and associated packages by typing **`./install-FAMEbuilder`**. This will take several minutes, as the `llvm` package is quite large.

By default, the install script, `install-FAMEbuilder` will install FAMEbuilder in the `famebuilder` directory where the bundle was unpacked. If you want to install it elsewhere, replace the command in step 3 with **`./install-FAMEbuilder --prefix=`*`install-prefix`***.

After you have completed the above steps, FAMEbuilder will be ready to use.

Preparing your environment

Once you have installed FAMEbuilder, each time you wish to use FAMEbuilder you must make sure your environment is appropriately set up. This is fairly simple; you need only ensure that *`install-prefix`*`/bin` is in your PATH environment variable. Your PATH environment variable should also include the path to the Xilinx ISE tools. Lastly, you must add *`install-prefix`*`/lib` to your LD_LIBRARY_PATH.

Once your environment is set up, you are ready to build platform support for FAMEbuilder.

Building platform support

After installing FAMEbuilder, you must build the platform support hardware libraries and drivers for the reconfigurable platform which you will use. Supported platforms are:

- Pico Computing Platform

Specific instructions are given below for each platform.

Pico Computing Platform

The Pico Computing's M-505 is a high-end FPGA card featuring the Xilinx Kintex-7 and an 8x PCI express Gen 2 interface. Use of this platform requires building of both a hardware library and compiling and installation of a device driver. To do this:

1. Set up your environment as in the previous section.
2. `famebuilder buildplatform --platformlib=platform-support-directory pico`

   This step generates the hardware support for the platform and will take several minutes to run.
3. `famebuilder builddriver pico`

   This step builds the driver.

Getting help

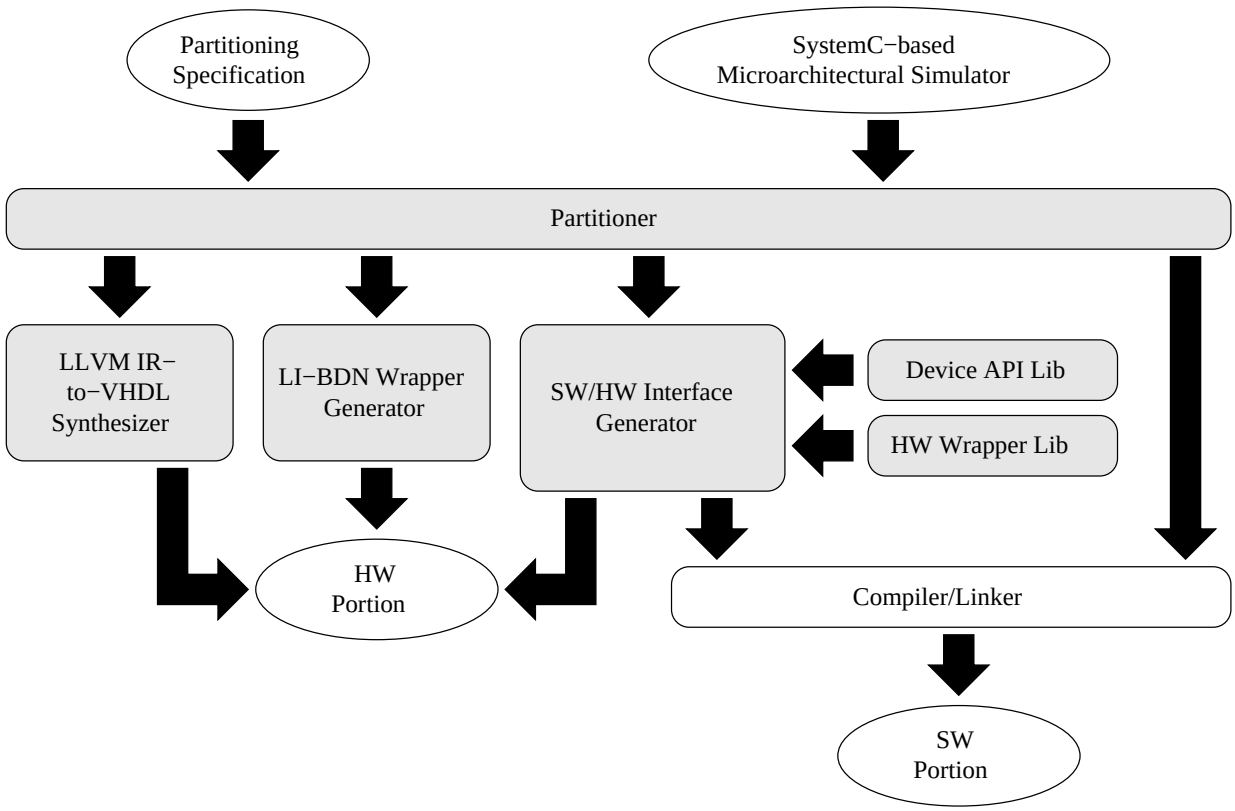
If you encounter problems with installation, please report them to the FAMEbuilder user group forum (https://groups.google.com/d/forum/famebuilder-users). When making such a report, please include complete details of the problem, including output from the installation scripts and version numbers for Python, g++, Perl, and ISE.

Chapter 2. A First Simulator

FAMEbuilder is a tool that facilitates the creation of FPGA Architectural Model Execution (FAME) simulators. FAME simulators offer much higher simulation speed than pure software simulators, allowing architects to better explore the architectural design space.

FAMEbuilder takes as input a simulator written in a structural style in SystemC and a partitioning specification and outputs the hardware and software necessary for a FAME simulator. This chapter walks through the construction of a FAME simulator from a simple SystemC specification.

Figure 2-1. FAMEbuilder Generation Overview



The Initial Specification

Example 2-1 shows a small SystemC specification that contains two modules: a **wire** module which simply echoes its input to its output and a **testbench** module which instantiates the wire, drives inputs to the wire, and checks the outputs. Notice that this is normal SystemC code and that no special constructs are needed. This SystemC model can be found in the file `Examples/Wire/Wire.cpp`.

Example 2-1. A SystemC specification

```
1 #include <systemc.h>
2 #include <iostream>
3
4 SC_MODULE(Wire) {
```

```
 5  public:
 6   sc_in<int> inPort;
 7   sc_out<int> outPort;
 8
 9   void Comb() {
10     outPort = inPort;
11   }
12
13   SC_CTOR(Wire) : inPort("inPort"), outPort("outPort") {
14     SC_METHOD(Comb);
15     sensitive << inPort;
16   }
17 };
18
19 SC_MODULE(testbench) {
20   sc_in<bool> clk;
21
22   Wire UUT;
23   sc_signal<int> toUUT;
24   sc_signal<int> fromUUT;
25
26   int cnt;
27
28   void TestIt() {
29     std::cerr << sc_time_stamp() << ":" << name() << " " << fromUUT << "\n";
30     if (fromUUT != cnt) {
31       std::cerr << sc_time_stamp() << ":" << name() << " ";
32       std::cerr << "Mismatch " << fromUUT << " instead of " << cnt << "\n";
33     }
34     toUUT = ++cnt;
35   }
36
37   SC_HAS_PROCESS(testbench);
38   testbench(sc_module_name name_) : sc_module(name_),
39     clk("clk"), UUT("UUT"), toUUT("toUUT"), fromUUT("fromUUT"), cnt(0) {
40     SC_METHOD(TestIt);
41     sensitive << clk.pos();
42     dont_initialize();
43     UUT.inPort(toUUT);
44     UUT.outPort(fromUUT);
45   }
46 };
```

Preparing the SystemC model for FAMEbuilder

To prepare to create a FAME simulator from a SystemC model, you must make two changes to the main SystemC simulation routine (`sc_main`) in your model. Example 2-2 illustrates these changes, highlighting them in italics. Within **sc_main** calls are made to `bardd::simopti::initialize` to start up the optimizer and perform other initialization steps. The call should include the command-line arguments passed to `sc_main`. After this call, the SystemC modules should be instantiated. Finally, after simulation has finished, `bardd::simopti::finalize` must be called to clean up the simulation. This code is included in `Examples/Wire/Wire.cpp`.

Example 2-2. Main Method Example

```
 1 #include <bardd/simopti/simopti.h>
 2 int sc_main(int argc, char *argv[])
 3 {
 4   bardd::simopti::initialize(&argc, &argv);
 5
 6   if (argc < 2) {
 7     std::cerr << "Usage: " << argv[0] << "<number of cycles>\n";
 8     exit(1);
 9   }
10   sc_clock clock("clock");
11   testbench *m = new testbench("test");
12   m->clk(clock);
13
14   sc_start(atoi(argv[1]), SC_NS);
15
16   bardd::simopti::finalize();
17   delete m;
18   return 0;
19 }
```

Preparing a partitioning file

FAMEbuilder uses partitioning files to learn the user's wishes about what portions of the SystemC simulation model should go in the hardware. A domain-specific language, described later, is used within these files. For this example, we use a file named `wire.part` with the following single line of contents:

```
tohw 0 [ test.UUT ];
```

This line tells FAMEbuilder to put the SystemC object named `test.UUT` into the hardware.

Building a FAME Simulator

Once you ahve a simulation model written in SystemC and the appropriate thre calls to FAMEbuilder , you will be able to use FAMEbuilder to build a FAME Simulator. To do so run the following commands in the command shell:

1. **famebuilder compile**. This command will take the specified SystemC source code file and pass it to the LLVM compiler with arguments to force the output to LLVM compiler bitcode and to locate the FAMEbuilder version of SystemC. Additional arguments for the LLVM compiler may be appended to this command. At the completion of this step, you should have a number of model bitcode files.
2. **famebuilder bitlink**. This command will link model bitcode files from the previous step into a single file named `sim.llvm.bc`. Simply list the names of the bitcode files after the command.
3. **famebuilder synthlink**. This command will link the synthesis executable. It will be named `a.out` by default but it can be named with the -o g++ linker argument. Other possible arguments are the same as those in g++.
4. **famebuilder synth**. This command will run the synthesis executable created in the previous step by **famebuilder synthlink**. This executable is the original SystemC simulator modified to stop execution shortly after elaboration and then synthesize the FAME simulator. The runtime arguments are those that could be

supplied to the SW-only SystemC simulator; these arguments are supplied to the synthesis executables and may be used to control how the model is instantiated. The synthesis options control the synthesis process. The partitioning specification is read from stdin by default, but can be read from a file instead.

5. **famebuilder implementsw**. This command will compile the interface source and bitcode files generated by **famebuilder synth** (using `gcc` and LLVM's `llc`) and links them with the original simulator and with runtime FAME libraries. The interface source and bitcode files must be in the current directory. The linker arguments should included any simulation libraries needed for the simulator. Most of the possible arguments are the same as those in g++.
6. **famebuilder implementhw**. This command will compile the HDL for the FAME simulator generated by **famebuilder synth**, link it to the platform HDL, map, place and route, and generate the final bitstream named fame.bit. It is important that the synthesis toolchain for your platform be in your PATH variable when this command is executed.

The commands you need to run for the provided Wire example are given in Example 2-3. These same commands can be found in `Examples/Wire/Makefile`.

Example 2-3. Sample commands to build the simulator using the provided Wire example.

1. **famebuilder compile -c Wire.cpp -o Wire.bc**
2. **famebuilder bitlink Wire.bc**
3. **famebuilder synthlink -o tosplitSim**
4. **famebuilder synth --pspec=wire.part --dir=generatedFiles pico ./tosplitSim 10**
5. **famebuilder implementsw --dir=generatedFiles -o hybridSim**
6. **famebuilder implementhw generatedFiles**

Running a FAME Simulator

Pico Computing Platform

To run a simulator on the Pico computing platfore, navigate to the directory containing the `fame.bit` file generated by **famebuilder implementhw**. From this directory on the simulation system, run the executable produced by **famebuilder implementsw** (named `a.out` by default). The Pico board will automatically load the `fame.bit` file in the current directory upon start up of the simulation.

Chapter 3. Writing Synthesizable Simulators

This chapter describes what you should do to prepare your SystemC simulators to be analyzed and synthesized. It also lists language features that cannot be synthesized nor analyzed (and thus must be avoided) and language features that cannot be synthesized and thus must remain in instances which are not moved to the hardware.

> **Note:** The language features which are listed are those which the developers have run into during testing or which are obvious to the developers. This list is not necessarily complete, and the developers would appreciate bug reports of language features not yet supported.

Broad limitations on models

There are some very broad limitations on the kinds of models which can be synthesized:

- There must be exactly one clock in the model.
- Processes must be sensitive to either the positive edge of the clock or to *all* of the signals they read, thus making them either fully sequential or combinational. Processes may be made sensitive to additional events.
- Static combinational loops of dependence between signals are not permitted. (For example, where signal A depends upon signal B and signal B depends upon signal A through combinational processes. This would be fine through sequential processes.)
- All processes of the entire SystemC model must be compiled with **famebuilder compile**. It is possible for those processes to call additional functions (e.g. in libraries), but the processes themselves must be made available to the synthesis tool.
- All processes of the entire SystemC model must correspond to functions/methods with external linkage. This limitation arises because the `dlsym` cannot find symbols with internal linkage.
- Signals and their derivatives (such as clocks) are the only permitted primitive channel types.

Preparing for synthesis

Preparing a model for synthesis is a very simple task: only two API calls need to be added. The first call, to `bardd::simopti::initialize` must be made with the argument count and vector passed into `sc_main` and should happen before any SystemC objects are created. (It may be possible to have some objects already created, but this behavior has not been tested.) This call both initializes the analysis system at synthesis time and initializes the runtime system at run time of the hybrid simulator. The argument count and vector will be modified by the call to remove synthesis- and/or runtime-specific arguments from the argument vector.

The second call, to `bardd::simopti::finalize` takes no arguments and should be made after simulation has terminated but before any simulation results are output. Besides cleaning up the analysis and runtime systems, the call will also initiate hardware transfer of data back into the software at runtime, allowing the model to report statistics and other information collected in hardware.

Additional things to consider

While FAMEbuilder can run with only the main loop changes, you may wish to make a few other alterations to your SystemC code. First, you may find it necessary to increase the precision of FAMEbuilder's analysis. This process is described in the Section called *Precision annotations* in Chapter 4. Second, you may find it helpful for debugging to give names to all your signals and avoid the default naming conventions which SystemC will use for unnamed signals. Third, you may need to add annotations for memory locations to be initialized or finalized, as described in the Section called *Memory annotations* in Chapter 4.

Restrictions on what can be analyzed

The following constructs are known to cause issues during analysis and must not be found in any SystemC processes.

- Calls to `default_event` made in a process. They are not a problem if made during elaboration.
- Floating-point operations
- Implied latches. You must never have a path through a non-clock-edge-sensitive process which does not write to all of the output signals of the process.
- Processes sensitive to the negative edge of a clock

Restrictions on what can be synthesized

The following constructs are known to cause issues during creation of VHDL code and must not be found in any SystemC processes which are being moved to hardware. They may be found in SystemC processes which remain in software.

- Very large signals. The problem here isn't generating the code, it's that the Xilinx toolchain can get stuck for many hours dealing with a signal of multiple kilobytes.
- Converting pointers to integers and back, unless the pointer can be successfully proven by the analysis to point to a single location. It's the conversion back that generally creates the problem..
- Calls to functions which haven't been compiled using **famebuilder compile**. While analysis can simply make assumptions about "missing" functions, synthesis cannot.

Chapter 4. Controlling FAME simulator synthesis

Partitioning specifications

FAMEbuilder assumes that the user will guide partitioning by providing a partitioning specification. A partitioning specification describes which design objects are to be placed into acceleration hardware and any synthesis attributes that are required. Partitioning specifications are written in a simple language called PartLang which allows sets of objects to be selected and manipulated.

A partitioning example

The purpose of PartLang is to help specify what set of objects are to be sent to hardware. In the examples released a sample file **tests.part** contains a very simple example of how PartLang is used and is shown in the Example 4-1. Here we see a set labled `all` that contains all objects that are to be implemented in hardware. This set is defined using a classic set notation and is then sent to hardware via the directive `tohw`.

Example 4-1. A Sample Partitioning File

```
1 set all = [ test.UUT ];
2 tohw 0 all;
```

PartLang specification

This subsection describes the constructs of PartLang.

Comments, file management, and delimiters

Comments can be introduced into PartLang code through either C or C++ style comments. PartLang files can be included in other files using the following syntax:

`include "`*`filename`*`";`

Statements are separated by semicolons.

Literals, predefined constants, identifiers, and variables

Literals in PartLang are of two kinds: numbers and strings. booleans, and sets. Number literals are double-precision floating point. Strings begin and end with double-quotes and may have escaped characters (`t`,`n`,`f`,`v`,`b`,`r`,`\`) within them.

There are several predefined constants. The boolean constants `true` and `false` are defined as `1` and `0`, repsectively. The `ALL` constant is the set of all SystemC objects. The `PLATFORM` constant is a string with the platform name.

Identifiers begin with an alphabet character and are followed by alphanumerics and/or underscores. There is a single global name scope. Identifiers are case-sensitive.

Variables may have one of three types: number, string, or set of model elements. Variables must be declared before they are used and can be optionally initialized, as seen below:

```
num i;
num i2 = 0;
string s;
string s2 = "Hello, world!\n";
set ST;
set ST2 = ALL;
```

Variables may not be re-declared, but they may be assigned new values using the = operator.

Control flow

PartLang contains a conditional construct, but no loop construct. The conditional construct is:

```
if (numeric expression) {
}
else {
}
```

The `else` clause is optional. Non-zero values of expressions are taken to mean `true`.

Numeric expression operators

The usual integer, logical, and comparison operators are supported. Comparison and negation operators return `1` for true and `0` for false. Update assignment operators as in C are also supported.

String expression operators

String concatenation is provided via the + operator. Strings can be compared with ==, `!=`, and =~ (the last treats the right-hand side as a regular expression and returns `true` if the left-hand side matches the regular expression.

Set expression operators

Sets in PartLang are sets of SystemC objects. The most basic set expression is a "filtering expression". It consists of an optional set expression called the "scope set" followed by square brackets with a regular expression between them. This set expression returns the SystemC objects from the scope set whose hierarchical name "matches" the regular expression. If no scope set expression is given, the scope set is assumed to be all SystemC objects.

Set union (`|`), disjuction (`&`), difference (-), and negation (~) operators are available.

Messages

PartLang supports the output of error, warning, and information messages as shown below:

```
error "This won't work at all!\n";
warning "This is a really bad idea.\n";
info "Did you know about this?\n";
```

Assigning a set of SystemC objects to hardware

The following syntax is used to assign SystemC objects to hardware:

```
tohw HWID setexpr;
```

This statement assigns the SystemC objects selected by set expression *setexpr* to hardware device number *HWID*. Because multi-FPGA simulators are not currently supported, *HWID* should always evaluate to zero at present. Only processes and module instances matter in the set assigned to hardware; the synthesis tool will infer which signals, ports, and elements of data structures must be in the hardware. If a module instance is moved to hardware, all processes of that instance are moved.

Assigning attributes

Attributes are a means to control the synthesis process through the partitioning specification. Attributes can be set on either SystemC objects or upon hardware devices.

Attributes are set on SystemC objects with the following syntax:

```
attrib attribute = value setexpr;
```

This syntax sets the value of attribute *replaceable* to *value* for all objects in *setexpr*. The only attribute currently supported is `compose`.

Attributes are set on hardware devices with the following syntax:

```
hwattrib HWID attribute = value;
```

This syntax sets the value of attribute *replaceable* to *value* for hardware device *HWID*. The meaning of the attributes are platform-dependent.

Memory annotations

On occasion, the hybrid simulator should initialize some piece of memory in the hardware to a different value from that which existed at synthesis time. One example of this would be a program memory where the program is loaded at simulator initialization in response to command-line arguments. The synthesis tool will assume that the memory's initial value is *always* the same value, which is not what is intended. This assumption can be overridden with an API call in the **bardd::simopti** namespace:

```
void flagDynamicMemory(void *addr, int size);
```

> Indicates that a block of memory of size *size* beginning at address *addr* may have different values at runtime than at synthesis time. The block of memory will be transferred from the host to the FPGA at runtime to ensure proper initialization. During finalization, the block of memory will be transferred back from the FPGA to the host. Over-use of this API when it is not needed will increase area usage and possibly FPGA cycle time.

This API call has a second use. It is normally the case that models collect statistics as they are running. For processes which are synthesized into hardware, these statistics need to be brought back from the FPGA into host

memory during finalization. The `flagDynamicMemory` call's second purpose, then, is to mark the variables holding statistics.

Precision annotations

Precision annotations are used to increase the precision of the analysis which FAMEbuilder makes of the SystemC processes. The purposes of this analysis are three-fold: to discover which SystemC objects are accessed by each SystemC API call, to discover which memory locations are shared between processes, and to discover the data and control dependences existing between the outputs and the inputs of processes. All this information is used to generate the correct FAME hardware and interfaces.

The analysis is interprocedural and context-sensitive. It has the capacity to be partially path-sensitive as well, but by default does not maintain path sensitivity in order to reduce synthesis time. In many situations, this loss of precision is insignificant. However, sometimes it can lead to later issues, particularly when spurious input-to-output dependences are reported which cause later analysis to believe there are cycles of dependence.

This loss of precision is manifest around the following five constructs in the SystemC process code:

- *Across loop iterations.* By default, loops are analyzed separately for their first iteration and all other iterations are considered together in the analysis. While greatly reducing runtimme, this convention can lead to the analysis running over the loop boundaries and considering memory locations and SystemC objects which are not accessed within the loop. It also leads to the aliasing of pointers used in the iterations after the first one, which can lead to spurious input-to-output dependences being reported, causing issues later on in the synthesis process.
- *At loop exits.* By default, all potentially exiting values from separate iterations of the loop are merged together, losing all information about how loop exit values correlate with the number of times the loop iterated before exiting. This convention can lead to aliasing and spurious input-to-output dependences.
- *At control-flow merges.* By default, all live values of a variable entering a basic block along different control paths are merged. (In other words, the PHI node's value is computed as the merge of the different incoming values.) This convention may also cause aliasing and spurious input-to-output dependences.
- *At returns from functions.* By default, all paths through a function are merged on return.
- *At calls to libraries not compiled using FAMEbuilder.* The LLVM code for the functions called from these libraries is not available for the analysis, so the analysis cannot make good assumptions about their behavior. By default, it assumes that all arguments are accessed (and thus must be computed) and that no memory is either read or written.

FAMEbuilder provides an API which allows you to override the precision defaults in portions of the model which require them. To use this API, you must first include `bardd/simopti/simopti.h` in the SystemC code you wish to annotate. The following functions are then available in the **bardd::simopti** namespace:

`void` **`annotateLoopStopAfter`**`(long` *`cnt`*`);`

When placed inside of a loop, indicate how many iterations to consider when analyzing the loop. The value passed in must be loop-invariant, but need not be constant. No attempt will be made to compute a fixed point for the remaining loop iterations if the loop has not exited before the indicated number of iterations. An attempt will be made to find earlier loop exits, which may cause the loop to be iterated fewer times than stated. Do not place more than one of this call or a call to `annotateLoopMax` at the same level of a loop nest. Note that if the loop is unrolled in compilation, this annotation will migrate outwards in the loop nest.

`void` **`annotateLoopConvergeAfter`**`(long` *`cnt`*`);`

When placed inside of a loop, indicate at what point to give up on computing a precise loop iteration count and attempt to instead find a fixed point for values computed the loop. An attempt will be made to find earlier loop exits, which may cause the loop to be iterated fewer times than stated. Do not place more than one of this call or a call to `annotateLoopExact` at the same level of a loop nest. Note that if the loop is unrolled in compilation, this annotation will migrate outwards in the loop nest.

`void` **`annotatePreciseOnExit`**`(void);`

When placed inside of a loop, indicates that live-out values from different loop iterations are not merged.

`template<T> T&` **`annotateMemoryEffects`**`(T &`*`call`*`, bool` *`reads`*`=true, bool` *`writes`*`=false);`

Indicates that at a particular point in the program, a memory location should be assumed to be read and/or written with arbitrary values. This annotation is used to wrap a call *`call`* to indicate its memory effects. The *`reads`* and *`writes`* parameters must be constant values. This annotation is a function template and returns the value returned by the call.

`void` **`annotateMemoryEffectsRange`**`(void *`*`p`*`, unsigned long`*`len`*`, bool` *`reads`*`=true, bool` *`writes`*`=false);`

Indicates that at a particular point in the program, a range of memory locations should be assumed to be read and/or written to with arbitrary values. This annotation is placed before or after a call to indicate the memory effects of the call; its exact location does not matter as long as it remains ordered with respect to loads and stores which preceed and/or follow the call. The *`reads`* and *`writes`* parameters must be constant values.

Chapter 5. Troubleshooting

This chapter discusses common issues that can lead to failures in simulator synthesis, bitfile generation, or incorrect execution at runtime. It is far from exhaustive; specific problems which aren't listed should be reported to <FAMEbuilder-support@googlegroups.com>.

famebuilder synthlink failures

Linking the synthesis executable rarely runs into difficulties. The most likely cause would be failing to include all of the libraries and/or linker flags needed to link the original SystemC model (with the exception of the SystemC library itself) on the command line.

famebuilder synth failures

There are a number of difficulties which may arise during the synthesis process. The most common ones are listed here.

A signal directly depends upon itself

This problem manifests itself by **famebuilder synth** terminating early with a message which looks like:

```
------------------------------------------------------------
--                 Generating LiBDN                       --
------------------------------------------------------------

Verifying Edges:
...
Edge: test_UUT_CombPortPtrs<test.UUT.CombPortPtrs_test.signal_3_out>->
test_UUT_CombPortPtrs<test.UUT.CombPortPtrs_test.signal_3_in>
cannot be dependent on itself
```

The key element is the phrase "cannot be dependent on itself". This message means that analysis has shown that a combinational process both writes and reads a signal and that the writing of the new value is either control- or data-dependent upon the current value of the signal. Such a situation is not valid in combinational logic and cannot be synthesized correctly.

The message indicates which signal and which SystemC process is involved; in this case the signal name is `signal_3` and the process is `test.UUT.ComboPortPtrs`.

There are two causes for this problem. One of the causes is an actual dependence of a signal upon itself. In this case, your model cannot be turned into a FAME simulator without some modification to make the combinational process truly combinational. However, the more likely cause is that analysis (and analysis of loops in particular) has not been sufficiently precise. If this is the case, annotating the loops inside the SystemC process using the annotation API described in the Section called *Precision annotations* in Chapter 4 should increase the precision to the point that this error disappears. Note that all loops in the process should be considered for annotation; often the imprecision arises because the analysis of what signals are accessed in a loop "steps off" the end of a loop and just happens to find another valid signal sitting there. Also, because of address-space randomization in Linux, this error may appear and disappear from run to run of **famebuilder synth**.

A signal transitively depends upon itself

This problem manifests itself by **famebuilder synth** terminating early with a message which looks like:

```
------------------------------------------------------------
--           Finished Generating LiBDN                    --
------------------------------------------------------------

0/test.signal_0.chg <= 1/test.signal_1.chg
1/test.signal_1.chg <= 0/test.signal_0.chg

Cost is: 3

( [ 0/1 ] . 1/2 )^2

famebuilder ERROR: cycle found in schedule.
```

The key element is the phrase "cycle found in schedule." FAME simulators built using FAMEbuilder must not have static cycles of dependence between signals driven by combinational processes. This restriction arises because the hardware components are built as a latency-insensitive bounded dataflow network, which does not permit such cycles of dependence.

The signals involved in the cycle can be determined by looking at any parenthesized expressions in the line of text after the "cost" line. This line of text is the order in which new signal values are computed in each simulated clock cycle and parenthesis indicate a repeated portion of the schedule. Within the repeated portion of the schedule are pairs of numbers separated by a `/`. The first of these numbers indicates the signal number, which can be cross-referenced against the list given before the "cost" line to find out the signal name (which will be on the left of the <=.)

There are two causes for this problem. One of the causes is actual static dependence of a signal upon itself through multiple signals. In this case, your model cannot be turned into a FAME simulator without some modification to break the cycle of dependence. However, the more likely cause is that analysis (and analysis of loops in particular) has not been sufficiently precise. If this is the case, annotating the loops inside the SystemC processes involved in the cycleusing the annotation API described in the Section called *Precision annotations* in Chapter 4 should increase the precision to the point that this error disappears. Note that all loops in the processes should be considered for annotation; often the imprecision arises because the analysis of what signals are accessed in a loop "steps off" the end of a loop and just happens to find another valid signal sitting there. Also, because of address-space randomization in Linux, this error may appear and disappear from run to run of **famebuilder synth**.

An indirect function is not inlined

This problem manifests itself by **famebuilder synth** terminating early with a message which looks like:

```
RTVR ERROR: Indirect call remaining in:
... -- listing of the LLVM bitcode for a SystemC process
```

The key element is the phrase "Indirect call remaining". This message means that a SystemC process which is meant to go to hardware calls a function using a function pointer which cannot be analyzed properly. You should see an indirect function call in the listed bitcode for the function.

This is a fairly rare problem, but its cause is a lack of precision in loop analysis. In the cases in which we have seen this error, a loop analysis has walked off the end of a loop and believes that some later iteration of the loop has stored a value into a vtable pointer. Annotation of loops inside SystemC processes using the annotation API

described in the Section called *Precision annotations* in Chapter 4 should be able to increase the precision sufficiently to solve the problem, but knowing which loop causes the problem is very difficult, as it depends upon the exact layout of data structures in memory.

To help you figure out what has happened, **famebuilder synth** will also print out for each load instruction in the SystemC process reporting the error a list of SystemC processes which wrote to memory locations which could be loaded by the load instruction. You can work back through the LLVM code to figure out where the offending vtable pointerwas loaded and then examine the loops in the writing processes.

Note that because of address-space randomization in Linux, this error may appear and disappear from run to run of **famebuilder synth**.

Pointers with multiple values cast to integers

This problem manifests itself by **famebuilder synth** terminating early with a message which looks like:

```
RTVR TODO: handle ptrtoint of address set in _ZN11RoutingLSE211RoutingLSE24CombEv
   %sub.ptr.lhs.cast.i3 = ptrtoint %struct.LSE_outport** %3 to i64
```

This message means that a pointer is being cast to an integer but the pointer does not have a unique value; such a situation is not yet handled, and likely can't be handled well at all. This situation seems to occur most frequently when the size of a vector is taken inside of a loop iterating across arrays or vectors of vectors.

This problem also arises due to a lack of precision in loop analysis; pointers to multiple vectors get aliased together, making it impossible to cast them to integers. Annotation of loops inside the SystemC process mentioned in the message using the annotation API described in the Section called *Precision annotations* in Chapter 4 should be able to increase the precision sufficiently to solve the problem.

famebuilder implementhw failures

Obvious failures in **famebuilder implementhw** are fairly rare, and when they do occur, are likely to be bugs in **famebuilder**. However, one non-bug, yet awkward situation, is known. When signal data types are large (on the order of 10s of Kilobytes), **xst** in the Xilinx toolchain can use extremely large amounts of memory and time (we've seen multiple 10s of GB used for 32 KB signals).

When **famebuilder implementhw** does fail, look in the `_synthFiles/log` directory under the build directory for the Xilinx tool log files.

famebuilder implementsw failures

These failures are very rare. When they occur, the most likely place to fail is in the final linking of the hybrid simulator; the cause is forgetting to add all the link-time options to the **famebuilder implementsw** command.

Runtime errors

There are several errors that may appear at runtime.

Changes in the SystemC database

Because the synthesis process is based upon SystemC objects, changes in the set of SystemC objects invalidate the results of synthesis. The hybrid simulators generated by FAMEbuilder check whether this has occurred, and report an error that looks like this:

```
Error! Runtime code has changed since synthesis!
The following objects have been removed since synthesis time!
Object name layout: Type%ObjectName%ObjectNameTypeID
sc_module%proc2%4Core
sc_signal%ugh_0%N7sc_core9sc_signalI9ifetchReqEE
sc_signal%ughR_0%N7sc_core9sc_signalI9ifetchReqEE
sc_signal%ughM_0%N7sc_core9sc_signalI9ifetchReqEE
sc_module%ifetchFlop_0%5cacheILj1ELj1ELj1ELj1EE
sc_module%ifetchFlop2_0%5cacheILj1ELj1ELj1ELj1EE
sc_signal%ugh2_0%N7sc_core9sc_signalImEE
```

Using the wrong bitfile

Loading the wrong bitfile into the FPGA is detected by the hybrid simulator runtime and reported as:

```
ERROR! Chksum value on FPGA differs from simulator!
```

Hangs and mismatches

When the hybrid simulator hangs or its results do not match those of the original software simulator, the most common cause for the problem is a timing error in the synthesized simulator. This cause can be tracked down by looking at `_synthFiles/fame.twr` to see if Xilinx reported any timing errors. Note that there are several different timing constraints; the `TS_CLK_250` is the constraint on the Pico platform which covers the synthesized logic. The timing report can be used to determine which process instances are involved, though some deduction based upon the signal and/or cell instance names is required.

The first solution for timing errors is to set the `compose` attribute to `0` for process instances in the critical path. This attribute prevents a process which analysis believes can be implemented in a single cycle from being "merged" with neighboring instances. Of course, if the process is not a single-cycle process, this solution has no effect.

The second solution is to manually insert delays between synthesized C++ operators. This is done by inserting a call to `bardd::simopti::insertFlop()` into the SystemC code for a process. Operations before the call and after the call will be synthesized to occur in different clock cycles.

Chapter 6. FAMEbuilder Command Reference

This chapter describes each of the FAMEbuilder commands with their options, listed in alphabetical order.

famebuilder bitlink

Name

`famebuilder bitlink` — link model bitcode files into a single file

Synopsis

famebuilder bitlink {*`model bitcode file`*...}

Description

Link model bitcode files into a single file named `sim.llvm.bc`.

famebuilder builddriver

Name

`famebuilder builddriver` — compile a platform device driver

Synopsis

famebuilder builddriver [--list] {*`platform`*}

Description

Compiles the driver for a given platform. The `--list` option lists all available platforms.

famebuilder buildplatform

Name

`famebuilder buildplatform` — Generate hardware support for a platform

Synopsis

famebuilder buildplatform [--list] [--dir=*`build directory`*] [--platformlib=*`platform support directory`*] {*`platform`*}

Description

Compiles the HDL source code for a given platform and generates a netlist file for the platform. The netlist is stored in the FAMEbuilder installation directories. The `--list` option lists all available platforms. The synthesis toolchain for your platform must be in your PATH variable when this command is executed. Synthesis takes place in the directory specified by the `--dir` option. The `--platformlib` option is used with some platforms to indicate where the platform support libraries have been installed; for these platforms, this command typically creates a file which "remembers" the platform library location.

famebuilder compile

Name

`famebuilder compile` — compile a model source code file

Synopsis

famebuilder compile [*`compiler argument`*...]

Description

Compiles a SystemC source code file using LLVM. The arguments provided to this command are passed to the LLVM compile comand (`clang++`). An argument to force the output format to LLVM bitcode and arguments to locate the FAMEbuilder version of SystemC are added. Most of the possible arguments are the same as those in g++.

famebuilder help

Name

`famebuilder help` — print help information for FAMEbuilder

Synopsis

famebuilder help [*command*]

Description

Prints out a list of FAMEbuilder commands. If a particular command is given as the argument, prints out help information for that command.

famebuilder installdriver

Name

`famebuilder installdriver` — install a platform device driver

Synopsis

sudo famebuilder installdriver [--list] {*platform*}

Description

Installs a given platform device driver. Only some platforms will have device drivers. This command may work only on Fedora/RHEL systems. The `--list` option lists all available platforms.

famebuilder implementhw

Name

`famebuilder implementhw` — compile the HDL for a FAME simulator to the reconfigurable platform

Synopsis

famebuilder implementhw [--dir=*design directory*] [--chipscope]

Description

Complies the HDL for a FAME simulator generated by **famebuilder synth**, links it to the platform HDL, maps, places and routes, and generates the final bitstream for the FAME simulator. The simulator HDL files must be in the current directory. The final bitstream will be named `fame.bit`. The synthesis toolchain for your platform must be in your PATH variable when this command is executed. The `--chipscope` option will generate a design with chipscope embedded in the platform interfaces; this option is not supported on all platforms.

famebuilder implementsw

Name

`famebuilder implementsw` — compile and link the software for a FAME simulator

Synopsis

famebuilder implementsw [--dir=*design directory*] [*linker argument*...]

Description

Compiles the interface source and bitcode files generated by **famebuilder synth** (using `gcc` and LLVM's `llc`) and links them with the original simulator and with runtime FAME libraries. The interface source and bitcode files must be in the current directory. The linker arguments should include any simulation libraries needed for the simulator. Most of the possible arguments are the same as those in g++.

famebuilder synth

Name

`famebuilder synth` — Generate FAME source files

Synopsis

famebuilder synth [--list] [--dir=*design directory*] [--logfile=*log filename*] [--bcfile=*bitcode filename*] [*synthesis options*...] {*platform*} {*synthesis executable*} [*runtime argument*...]

Description

Runs a synthesis executable generated by **famebuilder synthlink**. This executable is the original SystemC simulator modified to stop execution shortly after elaboration and then synthesize the FAME simulator. The runtime arguments are those that could be supplied to the SW-only SystemC simulator; these arguments are supplied to the synthesis executable and may be used to control how the model is instantiated.

famebuilder synthlink

Name

`famebuilder synthlink` — link synthesis executable

Synopsis

famebuilder synthlink [*g++ linker argument*...]

Description

Links the synthesis executable (named according to the g++ linker options given). The possible arguments are the same as those in g++.