Chapter 16. VHDL Domain

Authors: *Michael C. Williamson*

Other Contributors: *Christopher Hylands*
Edward A. Lee
José Luis Pino
William Tsu

16.1 Introduction

The VHDL domain generates code in the VHDL (VHSIC Hardware Description Language) programming language. This domain supports the synchronous dataflow model of computation. This is in contrast to the VHDLB domain, which supports the general discrete-event model of computation of the full VHDL language.

Since the VHDL domain is based on the SDF model, it is independent of any notion of time. The VHDL domain is intended for modeling systems at the functional block level, as in DSP functions for filtering and transforms, or in digital logic functions, independent of implementation issues.

The VHDL domain replaces the VHDLF domain. It is not, however, meant to be used in the same way as the VHDLF domain: the VHDL domain is for generating code from functional block diagrams with SDF semantics, while the VHDLF domain was intended to contrast with the VHDLB domain. It supported structural code generation using VHDL blocks with no execution delay or timing behavior, just functionality. The semantics for the VHDLF domain were not strictly defined, and quite a lot depended on how the underlying VHDL code blocks associated with each VHDLF star were written.

Within the VHDL domain, there are a number of different `Targets` to choose from. The default target, `default-VHDL`, generates sequential VHDL code in a single process within a single entity, following the execution order from the SDF scheduler. This code is suitable for efficient simulation, since it does not generate events on signals. The `SimVSS-VHDL` target is derived from `default-VHDL`, and provides facilities for simulation using the Synopsys VSS VHDL simulator. Communication actors and facilities in the `SimVSS-VHDL` target support code synthesis and co-simulation of heterogeneous CG systems under the `CompileCGSubsystems` target developed by José Pino. There is also a `SimMT-VHDL` target for use with the Model Technology VHDL simulator. The `struct-VHDL` target generates VHDL code in which individual actor firings are encapsulated in separate entities connected by VHDL signals. This target generates code which is intended for circuit synthesis. The `Synth-VHDL` target, derived from `struct-VHDL`, provides facilities for synthesizing circuit representations from the structural code using the Synopsys Design Analyzer toolset. Each of these targets is discussed in more detail in the next section.

Because the VHDL domain uses SDF semantics, it supports retargeting from other domains with SDF semantics (SDF, CGC, etc.) provided that the stars in the original graph are available in the VHDL domain. As this experimental domain evolves, more options for VHDL code generation from dataflow graphs will be provided. These options will include varying degrees of user control and automation depending on the target and the optimization goals of the code generation, particularly in VHDL circuit synthesis.

16.1.1 Setting Environment Variables

In order to have the Synopsys simulation target work correctly, you should make sure that the following environment variables and paths are set correctly. The `SYNOPSYS` and `SIM_ARCH` shell environment variables are settable within the Synopsys simulation target, SimVSS-VHDL, as target parameters by using edit-target (shift-t).

Also, you may need to permanently add the following lines to your .cshrc file and uncomment the ones you wish to take effect:

```
# For VHDL Synopsys demos, uncomment the following:
# setenv SYNOPSYS /usr/tools/synopsys
# setenv SIM_ARCH sparcOS5
# You need the last one of these (.../sge/bin) to run vhdldbx
# since vhdldbx looks for "msgsvr":
# set path = ( $path $SYNOPSYS/$SIM_ARCH/syn/bin $SYNOPSYS/$SIM_ARCH/
sim/bin $SYNOPSYS/$SIM_ARCH/sge/bin)
# You need this to run vhdlsim, and since vhdldbx calls vhdlsim, you
# need this to run vhdldbx also:
# setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${SYNOPSYS}/${SIM_ARCH}/
sim/lib
#
# For Motorola S56x card demos on the Sparc, you will need something
like:
# setenv S56DSP /users/ptdesign/vendors/s56dsp
# setenv QCKMON qckMon5
# setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${S56DSP}/lib
```

You will need to have a .synopsys_vss.setup file with the right library directive in it in order to use the communication vhdl modules needed for the CompileCGSubsystems target. This file in the root PTOLEMY directory has the correct directive defining the location of the PTVHDLSIM library. Synopsys simulation only sees the file if it is in one of three places: the current directory in which simulation is invoked, the configuration directory within the Synopsys installation tree, or the user's home directory. Since working directories are frequently created and destroyed, and since the Synopsys installation will vary from site to site, the user's home directory is the best place to put this file, but each user must do this if the root of their personal Ptolemy tree is anything other than their home directory.

Here is the text in $PTOLEMY/.synopsys_vss.setup:

```
-- This is so communication code can be
-- compiled into the PTVHDLSIM library:
PTVHDLSIM: $PTOLEMY/obj.$PTARCH/utils/ptvhdlsim
```

NOTE: If you build your own tree and it includes your own `$PTOLEMY/src/utils/ptvhdlsim` directory, then you will need to modify your .synopsys_vss.setup file to point to this directory prior to building the new tree. During the build process, this file is needed so that the ptvhdlsim executable can be correctly linked. If it is pointing to some other directory, then you may experience problems linking ptvhdlsim.

16.2 VHDL Targets

The targets of the VHDL domain generate VHDL code from SDF graphs. The targets differ from one another in the styles of VHDL code which they produce, or in the facilities they provide for passing the generated code to VHDL simulation or circuit synthesis tools. The graphs of VHDL actors in Ptolemy are meant to be retargetable in that one graph can be used with multiple VHDL targets, depending on the circumstances. The available targets in the VHDL domain are: `default-VHDL`, `struct-VHDL`, `SimVSS-VHDL`, `SimMT-VHDL`, and `Synth-VHDL`. There is also support for using `SimVSS-VHDL` as a child target of `CompileCGSubsystems` for heterogeneous code generation and co-simulation.

All of the VHDL targets share the following parameters, which are inherited from the base class `HLLTarget`:

directory (`STRING`) Default = `$HOME/PTOLEMY_SYSTEMS`
The name of the directory into which generated code files and supporting files are written. In derived targets, this is also the directory in which compilation for simulation and synthesis are performed.

Looping Level (`INT`) Default = `0`
The control for selecting the looping complexity of the SDF scheduler which is used. Note that looping of code is not supported in the current implementation, except at the main iteration loop on the outside. Therefore a looping level of zero should be used with all loop schedulers or incorrect code may result. In future releases, higher looping levels will be supported.

display? (`INT`) Default = `TRUE`
Option to display generated codefiles to the screen.

write schedule? (`INT`) Default = `FALSE`
Option to write the schedule to a file. The name of the file will be *<galaxy name>.sched*.

16.2.1 The `default-VHDL` Target

The `default-VHDL` target generates VHDL code in a simple and straightforward style which is designed to preserve the SDF scheduling order while incurring minimum VHDL simulation overhead. The code is generated as a single VHDL entity containing a single process of sequential statements. The sequential process reflects the order of execution determined by the SDF scheduler. All data values are stored and communicated through internal variables so that the simulation overhead of VHDL signals and the VHDL discrete-event scheduler can be avoided. No actual simulation is performed by the `default-VHDL` target. It is left to derived targets to support VHDL simulation.

To generate the code, the `default-VHDL` target first invokes the SDF scheduler, and then goes through the resulting schedule in order, firing each VHDL star in sequence. As each VHDL star is fired, a block of VHDL sequential statements is generated. `Porthole` and `State` references and values are resolved and any necessary VHDL variables are created and placed in the list of declared variables. One VHDL star may be fired multiple times and each firing will cause a new codeblock with new variables to be generated. The target manages the communication of data from one VHDL star to the next through VHDL variables. The target also manages state propagation from one firing to the next of the same VHDL star through VHDL variables. State values and tokens remaining on arcs at the end of the schedule iteration are also fed back through the correct variables so that the process can be looped repeatedly and function identically to the original SDF graph.

16.2.2 The `struct-VHDL` Target

The `struct-VHDL` target generates VHDL code in a structural style, in which firings of VHDL stars are individually encapsulated in VHDL entities. The entities are connected to one another through VHDL signals, and the flow of data and state from one firing entity to the next enforces the precedence relationships inherent in the dataflow graph and the resulting schedule. The overall structure of the completed code description parallels the precedence directed acyclic graph (DAG).

The procedure used by the `struct-VHDL` target to generate the code begins similarly to that of the `default-VHDL` target. First, the SDF scheduler is invoked and a valid schedule is computed. Then the schedule is run, and as each VHDL star is fired, the target generates an individual VHDL entity for each firing while keeping track of input and output references to portholes and states. The target manages the references so that it can correctly instantiate each VHDL entity and create VHDL signals to map to the VHDL ports for carrying data and state from one firing to the next. Only firings which have actual dependencies will be connected in the VHDL code representation. In this way, the code generated represents the maximum parallelism in the graph computation outside the granularity level of an individual firing.

The current version of the `struct-VHDL` target also generates registers for latching the values of states and remaining tokens at the end of an iteration. It feeds back the outputs of these registers to the correct inputs at the beginning of the graph so that the structure can be "clocked" by an input clock signal common to all such registers. This clock, on a positive transition, represents the tick of one completed iteration of the dataflow graph. This clock becomes an input to the entire top-level VHDL entity, and will presumably be supplied by an outside source or signal driver during simulation. Similarly, there is an input created for a control signal which selects between the initial values of states or initial tokens and the succeeding values which are passed from one iteration to the next.

16.2.3 The `SimVSS-VHDL` Target

The `SimVSS-VHDL` target is derived from the `default-VHDL` target. It generates code in the same single-entity, single-process, sequential style as the `default-VHDL` target, but it also provides facilities for simulation using the Synopsys VSS VHDL simulator. Depending on the target parameters set when running this target, following the code generation phase this target can compile, elaborate, and execute interactively or non-interactively the design specified by the generated VHDL code.

Communication actors and facilities in the `SimVSS-VHDL` target support code synthesis and co-simulation of heterogeneous CG systems under the `CompileCGSubsystems` target developed by José Pino. This allows a user to manually partition a graph using hierarchy so that multiple codefiles of different code generation domains can be generated. They are then executable if run on host machines which provide all the needed simulators and supporting hardware resources that the individual child targets require. The communication between the different code generation subsystems is automatically generated and correct synchronization and deadlock avoidance are guaranteed. This capability is demonstrated with VHDL in a number of demos included through the main VHDL demo palette.

The additional parameters of the `SimVSS-VHDL` target are as follows:

$SYNOPSYS (`STRING`) Default = `/usr/tools/synopsys`
Value of the `SYNOPSYS` environment variable. It points to the root of the Synopsys tools installation on the host machine.

$ARCH (`STRING`) Default = `sparcOS5`
Value of the `ARCH` environment variable. It indicates which architecture/operating system the Synopsys tools will be run on.

$SIM_ARCH (`STRING`) Default = `sparcOS5`
Value of the `SIM_ARCH` environment variable. It indicates which architecture/operating system the Synopsys VSS simulator will be run on.

analyze (`INT`) Default = `TRUE`
If `TRUE` then attempt to analyze the VHDL code using the `gvan` tool, checking for syntax errors.

startup (`INT`) Default = `TRUE`
If `TRUE` then attempt to startup the VHDL simulator (`vhdldbx` if *interactive* = `TRUE`, else `ptvhdlsim`).

simulate (`INT`) Default = `TRUE`
Currently unused. If *interactive* = `FALSE`, simulation under `ptvhdlsim` will begin automatically following startup.

report (`INT`) Default = `TRUE`
Currently unused.

interactive (`INT`) Default = `FALSE`
If `TRUE` then when simulating, run `vhdldbx`. Otherwise, run `ptvhdlsim`.

16.2.4 The `SimMT-VHDL` Target

The `SimMT-VHDL` target is derived from the `default-VHDL` target. It generates code in the same single-entity, single-process, sequential style as the `default-VHDL` target, and also provides facilities for simulation using the Model Technology VHDL simulator. Depending on the target parameters set when running this target, following the code generation phase this target can compile, elaborate, and execute interactively or non-interactively the design specified by the generated VHDL code.

The additional parameters of the `SimMT-VHDL` target are as follows:

analyze (INT) Default = TRUE
If TRUE then attempt to analyze the VHDL code using the vcom tool, checking for syntax errors.

startup (INT) Default = TRUE
If TRUE then attempt to startup the vsim VHDL simulator

simulate (INT) Default = TRUE
Currently unused. If *startup* = TRUE and *interactive* = FALSE, simulation under vsim will begin automatically following startup. If *startup* = TRUE and *interactive* = TRUE, vsim will startup but wait for user input.

report (INT) Default = TRUE
Currently unused.

interactive (INT) Default = FALSE
If TRUE, then when simulating, start up vsim and wait for user input. If FALSE, then when simulating, run vsim in the background.

16.2.5 The Synth-VHDL Target

The Synth-VHDL target is derived from the struct-VHDL target. It generates code in the same structural style as the struct-VHDL target, but it also provides facilities for synthesis and optimization using the Synopsys Design Analyzer toolset.

Not every design which can be specified as an SDF graph using the VHDL stars available in the main star palettes will be synthesizable. Some stars generate code which is not synthesizable under the rules required by the Synopsys Design Analyzer.

There is conceptually more than one way to generate synthesizable VHDL for a given dataflow graph. Just as the sequential VHDL of the default-VHDL target differs from the structural VHDL of the struct-VHDL target, so there are also multiple ways in which the structural VHDL could be generated. The struct-VHDL target as is only generates one particular style. A programmer with some experience could modify this target or create a new or derived target to generate the code in a different structural style to suit different needs. Future releases of Ptolemy may include additional structural VHDL targets for synthesis and/or a modified version of the ones included in the 0.7 release.

The additional parameters of the Synth-VHDL target are as follows:

analyze (INT) Default = TRUE
If TRUE then attempt to analyze the VHDL code using the design_analyzer tool, checking for syntax errors.

elaborate (INT) Default = TRUE
If TRUE then attempt to elaborate the analyzed design into a netlist form.

compile (INT) Default = TRUE
If TRUE then attempt to compile the elaborated design into an optimized netlist.

report (INT) Default = TRUE
If TRUE then generate reports on the compile-optimized designs for area and timing.

16.2.6 Cadence Leapfrog Ptolemy Interface

Xavier Warzee of Thomson-CSF and Michael C. Williamson created an interface for the Cadence Leapfrog VHDL Simulator.

`$PTOLEMY/src/domains/vhdl/targets` contains the Cadence Leapfrog VHDL Target. This target, `SimLF-VHDL`, allows simulation of generated VHDL code with the Leapfrog simulator from Cadence. This target is analogous to the `SimVSS-VHDL` target, which supports simulation with the Synopsys VHDL System Simulator.

Setup

To use the Leapfrog you need to have the following setup. Locally, our Cadence installation is at `/usr/eesww/cadence`, so your `.cshrc` would contain:

```
setenv PATH /usr/eesww/cadence/9504/tools/leapfrog/bin:$PATH
setenv CDS_LIB /usr/eesww/cadence/9504/tools/leapfrog
setenv CDS_INST_DIR /usr/eesww/cadence/9504
```

You also need to set up some files.

In the directory where the VHDL code is generated, for example `~/PTOLEMY_SYSTEMS/VHDL`, the following two files must be provided:

`cds.lib` contains

```
softinclude $CDS_VHDL/files/cds.lib
define leapfrog ./LEAPFROG
define alt_syn $CDS_INST_DIR/lib/alt_syn
```

`hdl.var` contains:

DEFINE WORK leapfrog include $CDS_VHDL/files/hdl.var

and the directory `~/PTOLEMY_SYSTEMS/VHDL/LEAPFROG` must exist

16.3 An Overview of VHDL Stars

The figure below shows the top-level palette of VHDL stars. The stars are divided into categories: sources, sinks, arithmetic functions, nonlinear functions, control, conversion, signal processing, and higher order functions. The higher order function stars are the same ones that are common to all domains and they are not particular to VHDL. Icons for `delay`, `bus`, and `BlackHole` appear in most palettes for easy access. Most of the stars in the VHDL domain have equivalent counterparts in the SDF domain. See “An overview of SDF stars” on page 5-4 for brief descriptions of these stars.

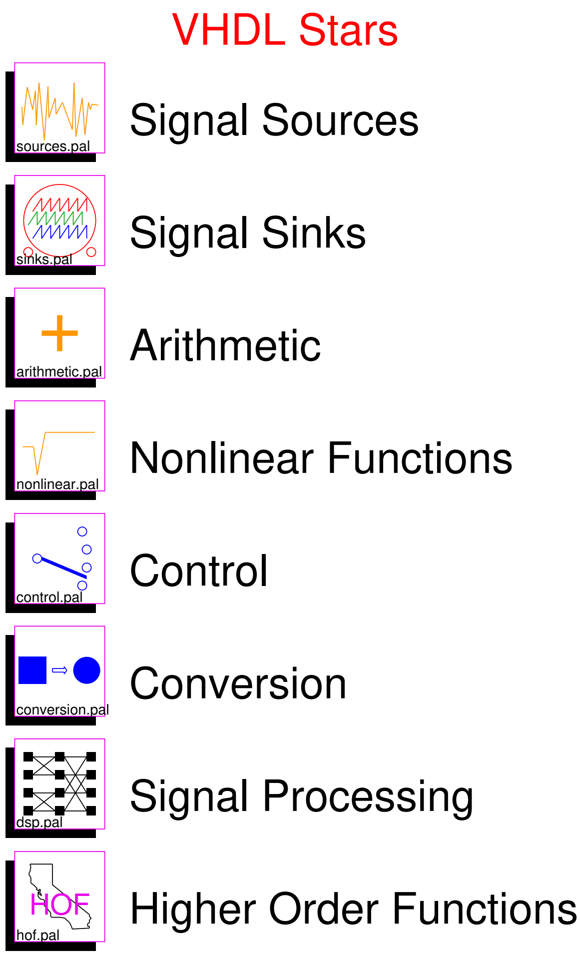

FIGURE 16-1: Top-level palette of stars in the VHDL domain.

16.3.1 Source Stars

Source stars have no inputs and produce data on their outputs. The figure below shows the palette of VHDL source stars. All of these are equivalent to the SDF stars of the same name.

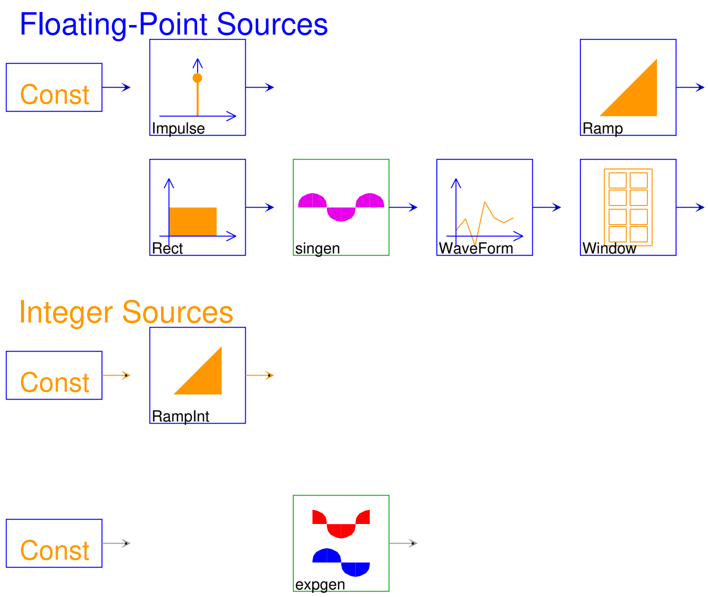

FIGURE 16-2: Source stars in the VHDL domain.

16.3.2 Sink Stars

Sink stars have no outputs and consume data on their inputs. The figure below shows the palette of VHDL sink stars. All of these are equivalent to the SDF stars of the same name.

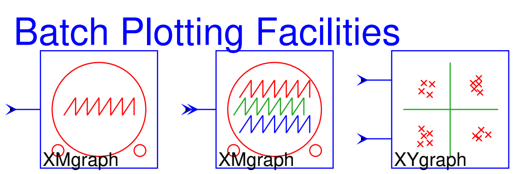

FIGURE 16-3: Sink stars in the VHDL domain.

16.3.3 Arithmetic Stars

Arithmetic stars perform simple functions such as addition and multiplication. The figure below shows the palette of VHDL arithmetic stars. All of the stars are equivalent to the

SDF stars of the same name.

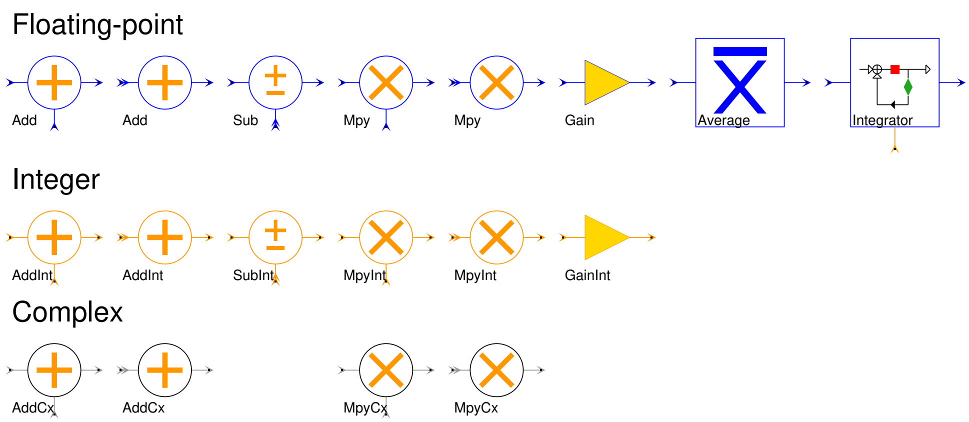

FIGURE 16-4: Arithmetic stars in the VHDL domain.

16.3.4 Nonlinear Stars

Nonlinear stars perform simple functions. The figure below shows the palette of VHDL nonlinear stars. All of these are equivalent to the SDF stars of the same name.

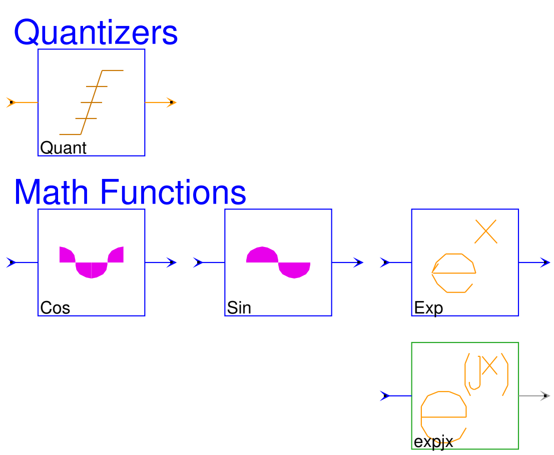

FIGURE 16-5: Nonlinear stars in the VHDL domain.

16.3.5 Control Stars

Control stars are used for routing data and other control functions. The figure below shows the palette of VHDL control stars. All of these are equivalent to the SDF stars of the same name.

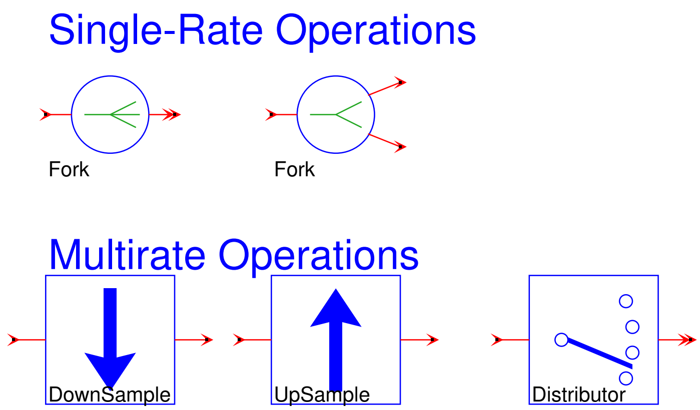

FIGURE 16-6: Control stars in the VHDL domain.

16.3.6 Conversion Stars

Conversion stars are used to convert between different data types. The figure below shows the palette of VHDL conversion stars. All of the stars are equivalent to the SDF stars of the same name.

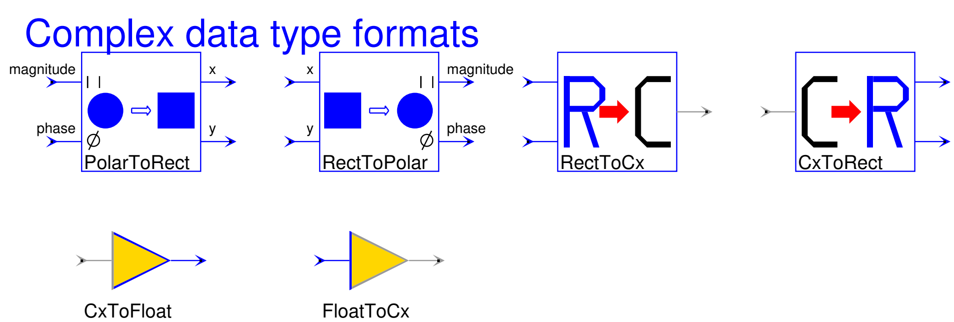

FIGURE 16-7: Type-conversion stars in the VHDL domain.

16.3.7 Signal Processing Stars

The figure below shows the palette of VHDL signal processing stars. All of the stars are equivalent to the SDF stars of the same name (see “Signal processing stars” on page 5-30).

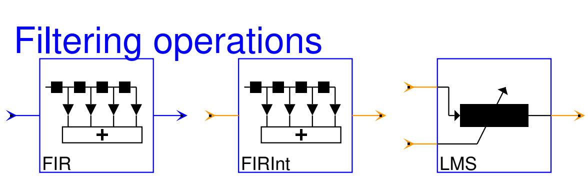

FIGURE 16-8: Signal processing stars in the VHDL domain.

16.4 An Overview of VHDL Demos

The figure below shows the top-level palette of VHDL demos. The demos are divided into categories: code generation, simulation, synthesis, and cosimulation. Some of the demos in the VHDL domain have equivalent counterparts in the SDF or CGC domains. See "An overview of SDF demonstrations" on page 5-51 for brief descriptions of these demos. Brief descriptions of the demos unique to the VHDL domain are given in the sections that follow.

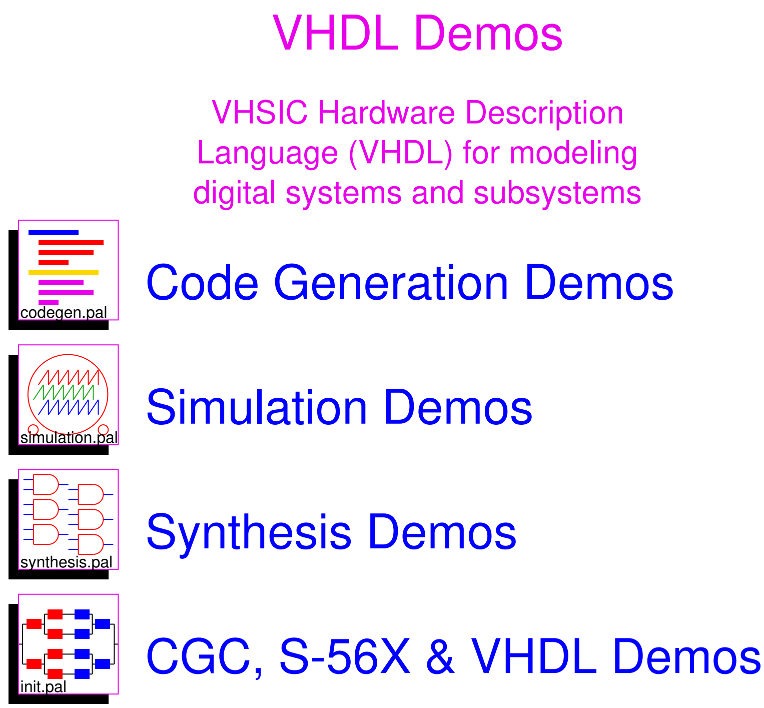

FIGURE 16-9: Top-level palette of demos in the VHDL domain.

16.4.1 Code Generation Demos

Figures below show demos that do nothing but generate code.

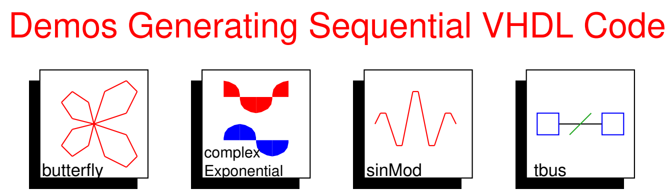

FIGURE 16-10: Sequential Code Generation Demos.

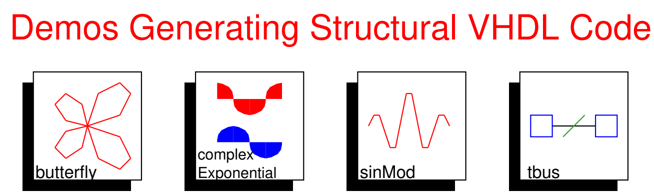

FIGURE 16-11: Structural Code Generation Demos.

The sequential demos use the `default-VHDL` target. The structural demos use the `struct-VHDL` target. They are essentially the same systems being run, but with two different targets producing two different styles of VHDL code. These demos provide a direct comparison of these two basic styles of VHDL code generation.

16.4.2 Simulation Demos

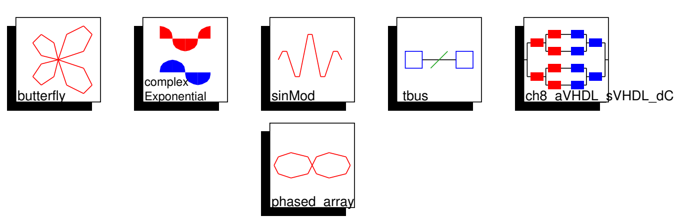

FIGURE 16-12: Demos using the Synopsys VSS Simulator.

These demos use the `SimVSS-VHDL` target. Each one generates VHDL code which is functionally equivalent to the SDF graph specification, and then the code is executed on the Synopsys VSS Simulator. Graphical monitoring blocks provide output analysis of the results of running these systems.

16.4.3 Synthesis Demos

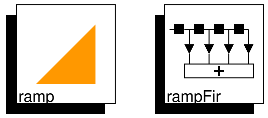

FIGURE 16-13: Demos using the Synopsys Design Analyzer for synthesis.

These demos use the `Synth-VHDL` target. Each one generates structural VHDL code which is equivalent to the SDF specification. One difference is that the data types are converted to simple 4-bit integers to speed up the synthesis process. Once the code is generated, the netlist is synthesized through the Synopsys Design Analyzer. Following that, the netlist is optimized and then control of the Design Analyzer is returned to the user for further exploration and inspection.

16.4.4 Cosimulation Demos

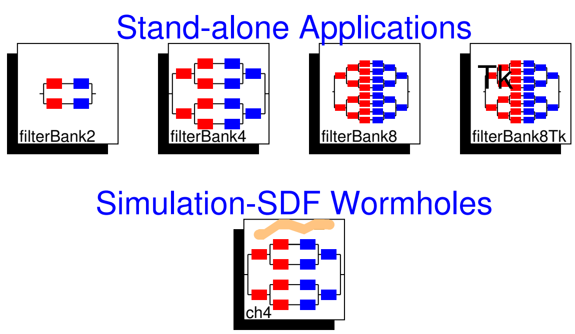

FIGURE 16-14: Demos mixing simulation in VHDL, C, and Motorola DSP56000 code.

These demos use the `CompileCGSubsystems` target which uses the SimVSS-VHDL target as a child target for the VHDL portions of the systems. The first three demos generate stand-alone heterogeneous programs which run in C, Motorola DSP56000 assembly, and VHDL. They produce analysis and synthesis filterbanks for perfect reconstruction using progressively more complex structures. The fourth demo also generates a Tcl/Tk user interface for selecting one of three waveform inputs to the system. The fifth and final demo generates the filterbank system, but instead of doing it as a standalone program, it incorporates the system into a wormhole inside a top-level SDF system. This way the subsystem can be executed in code which is potentially faster than SDF simulation, and it can be reused without having to recompile the subsystem each time the top-level system is executed.