PeaCE
USER’S MANUAL

version 1.0b

November 25, 2004

Copyright© 1995-2004 The CAP laboratory of Seoul National University
and The Prigent Corporation.
All Rights reserved.
# Table of Contents

Table of Contents ......................................................................................................................... i
List of Figures ............................................................................................................................... x

## Chapter 1. Introduction ............................................................................................................ 1
  1.1 Who may want to use PeaCE? ......................................................................................... 1
  1.2 HW/SW Codesign Methodology and Current Practice ................................................... 2
  1.3 PeaCE Codesign Flow ..................................................................................................... 3
  1.4 History .............................................................................................................................. 5
  1.5 Software Structure .......................................................................................................... 5
  1.6 Installation ...................................................................................................................... 7
  1.7 Terminology ................................................................................................................... 10

## Chapter 2. Getting Started ................................................................................................... 11
  2.1 Starting PeaCE ................................................................................................................. 11
    2.1.1 Running PeaCE daemon .......................................................................................... 11
    2.1.2 Running PeaCE client (HAE) ............................................................................... 12
  2.2 Running a demo application ............................................................................................. 14
    2.2.1 Opening a demo application .................................................................................... 14
    2.2.2 Running the demo application ............................................................................... 15
  2.3 Creating a simple application .......................................................................................... 16
    2.3.1 Creating a new schematic ...................................................................................... 16
    2.3.2 Configuring domain and target ............................................................................. 17
    2.3.3 Using blocks in SPDF library ................................................................................ 18
    2.3.4 Editing block states ............................................................................................... 20
    2.3.5 Connecting blocks ................................................................................................. 20
    2.3.6 Running the application ....................................................................................... 21

## Chapter 3. Graphical User Interface .................................................................................... 23
  3.1 Running PeaCE ................................................................................................................. 23
    3.1.1 Starting options ...................................................................................................... 23
    3.1.2 Overview of PeaCE window ................................................................................... 24
    3.1.3 Menus .................................................................................................................... 25
  3.2 Working with files ............................................................................................................. 27
    3.2.1 Creating and opening files ...................................................................................... 27
    3.2.2 Saving files ............................................................................................................ 28
    3.2.3 Closing files ........................................................................................................... 29
    3.2.4 Exporting files ....................................................................................................... 29
    3.2.5 Printing files .......................................................................................................... 29
  3.3 Working with designs ......................................................................................................... 30
    3.3.1 Using design browser ............................................................................................ 30
3.3.2 Creating and opening a design ................................................................. 30
3.3.3 Configuring domain ............................................................................. 31
3.3.4 Configuring target and editing target states ....................................... 31
3.3.5 Running a design ................................................................................ 33

3.4 Using block libraries .............................................................................. 34
3.4.1 Using library browser .......................................................................... 35
3.4.2 Loading a block (or super-block) from a specific domain library ......... 35
3.4.3 Editing block (and super-block) states ............................................... 36
3.4.4 Parameter expressions ....................................................................... 37
3.4.5 Complex-valued states ...................................................................... 38
3.4.6 Array states ........................................................................................ 38
3.4.7 String states ....................................................................................... 38
3.4.8 Using Tcl expressions in states ......................................................... 39
3.4.9 Inserting comments in a state ............................................................ 40
3.4.10 Building a combo-box for a state ..................................................... 40
3.4.11 Reading state values from a file ....................................................... 41
3.4.12 Using delays ..................................................................................... 41

3.5 Making your own block libraries ............................................................ 42
3.5.1 Creating blocks .................................................................................. 42
3.5.2 Importing a new block ....................................................................... 43
3.5.3 Registering a new block ..................................................................... 44
3.5.4 Link Star ............................................................................................ 45
3.5.5 Creating a super-block ...................................................................... 45
3.5.6 Adding super-block ports .................................................................. 46
3.5.7 Adding super-block states .................................................................. 47
3.5.8 Registering super-blocks ................................................................... 47
3.5.9 Create an Icon with Multi-port .......................................................... 48
3.5.10 Editing library browser .................................................................... 48

3.6 Editing and viewing schematics .............................................................. 49
3.6.1 Placing blocks .................................................................................... 49
3.6.2 Connecting blocks ............................................................................. 49
3.6.3 Selecting blocks and lines ................................................................. 49
3.6.4 Moving blocks and lines ................................................................... 49
3.6.5 Cutting/Copying/Pasting/Deleting blocks ......................................... 50
3.6.6 Rotating blocks ................................................................................ 50
3.6.7 Editing block icons ........................................................................... 50
3.6.8 Editing comments ............................................................................. 51
3.6.9 Zooming a schematic ........................................................................ 51
3.6.10 Looking inside of a block ................................................................. 52
3.6.11 Viewing Names of blocks .................................................................. 52
Chapter 6. SPDF Model : VHDL Code Generation

6.1 Before You Begin.................................................................124
6.2 Limitations ........................................................................124
6.3 Using VHDL Domain with a Simple Example ......................125
   6.3.1 Create a new schematic file ..........................................125
   6.3.2 Draw an SPDF graph with library blocks ......................126
   6.3.3 Set block parameters ...................................................129
   6.3.4 Make a resource File ..................................................130
   6.3.5 Run the example .......................................................132
6.4 VHDL Block Definition ......................................................133
   6.4.1 Block definition and the generated code: RampInt ..............135
   6.4.2 Setting the width of ports and states .............................136
   6.4.3 Type of block ...........................................................138
Multi-cycle sequential logic ....................................................139
   6.4.4 Timing Model of a HW Library Block ..............................140
6.5 Scheduling and Code Generation for HW Synthesis ...............141
   6.5.1 Schedule Information File ..........................................141
   6.5.2 Generated VHDL code structure ..................................143
6.6 Running a simple Demo : Butterfly_ModelSim ....................144
6.7 Running a Demo : JPEG Encoder ........................................146
   6.7.1 JPEG Encoder using ChenDCT ......................................147
   6.7.2 JPEG Encoder using 1-dimensional DCT ......................154
6.8 Running a Demo : H.263 Decoder example ............................161

Chapter 7. Specification with DE Model ....................................166
7.1 Draw a DE graph Using Pre-defined Library Blocks.................................................166
7.2 DE Block Definition.................................................................................................168
  7.2.1 Delay-type and Functional Blocks........................................................................170
  7.2.2 Blocks With Multiple Inputs ............................................................................171
  7.2.3 Event Generators ............................................................................................172
7.3 Discrete Event Simulation........................................................................................173
  7.3.1 Simultaneous events .......................................................................................174
7.4 Mixed Simulation Capability ....................................................................................175
  7.4.1 Mixed Simulation With SPDF Model ................................................................175
  7.4.2 Mixed simulation with Octave Analog Simulator ............................................177
Chapter 8. FSM Model Specification ..............................................................................180
  8.1 Basic FSM Definition .........................................................................................180
  8.2 fFSM: FSM extension ..........................................................................................185
    8.2.1 Concurrency .................................................................................................186
    8.2.2 Hierarchy ......................................................................................................186
    8.2.3 Internal event ...............................................................................................188
    8.2.4 Variable state ...............................................................................................190
  8.3 FSM Simulation ......................................................................................................191
  8.4 Talking with Computation Module ........................................................................194
    8.4.1 Simple Counter example ..............................................................................198
    8.4.2 Add a SPDF super-block .............................................................................201
  8.5 Code Generation ....................................................................................................203
    8.5.1 FSMCGCTarget ...........................................................................................203
    8.5.2 FSMVHDLTarget .......................................................................................204
    8.5.3 FSMSimVHDLTarget ..................................................................................204
Chapter 9. Task level specification model (BP Domain) ..................................................207
  9.1 Task model in PeaCE ..........................................................................................207
  9.2 How to design a task-model in PeaCE .................................................................209
  9.3 Sporadic task in the Task-model (EXPERIMENTAL) .........................................213
  9.4 SPDF extension in Task-model ..........................................................................214
    9.4.1 Variable-size dynamic-rate port ...................................................................215
    9.4.2 Exception handling ......................................................................................218
  9.5 Define task group to control multiple tasks simultaneously ..................................219
  9.6 Task-model design example ..............................................................................220
Chapter 10. PeaCE design flow overview ......................................................................225
  10.1 Algorithm specification and functional simulation .............................................226
  10.2 Candidate architecture description: arch tab ....................................................229
  10.3 Performance estimation for the base platform: estimation tab .............................230
  10.4 Generation of interfacing XML files: analysis tab ............................................232
  10.5 HW/SW partitioning: partition tab .....................................................................234
10.6 Design space exploration of communication: dse tab .................................................. 237
10.6.1 Memory trace generation ......................................................................................... 237
10.6.2 Design space exploration for bus architecture ......................................................... 240
10.7 Timing cosimulation for HW/SW verification: cosim tab .......................................... 241
10.7.1 Using PeaCE cosimulation tool ............................................................................... 241
10.7.2 Using Seamless CVE for verification .................................................................... 242
10.8 Cosynthesis for prototyping board: cosyn tab (NOT IMPLEMENTED) ....................... 244
Chapter 11. Interface Files ............................................................................................... 245
11.1 Graph topology description file: Clustered.xml ........................................................ 245
11.1.1 Usage ...................................................................................................................... 245
11.1.2 Syntax description .................................................................................................. 245
11.1.8 An illustrative example ......................................................................................... 249
11.1.9 BNF description of the syntax .............................................................................. 251
11.2 Block performance description file: TimeCost.xml .................................................... 253
11.2.1 Usage ...................................................................................................................... 253
11.2.2 Syntax description .................................................................................................. 253
11.2.5 An illustrative example: DivX player example ....................................................... 255
11.2.6 BNF description of the syntax .............................................................................. 257
11.3 Mode description file: mode.xml ............................................................................... 258
11.3.1 Usage ...................................................................................................................... 258
11.3.2 Syntax and description ........................................................................................... 258
11.3.5 An illustrative example: MMMT Terminal ............................................................. 259
11.3.6 BNF description of the syntax .............................................................................. 261
11.4 Partition and schedule description file: sched.xml .................................................... 262
11.4.1 Usage ...................................................................................................................... 262
11.4.2 Syntax description .................................................................................................. 262
11.4.7 An illustrative example: DivX player example ....................................................... 266
11.4.8 BNF description of the syntax .............................................................................. 268
11.5 Architecture description file: Arch.xml ..................................................................... 269
11.5.1 Usage ...................................................................................................................... 269
11.5.2 Syntax description .................................................................................................. 269
Description of processor .................................................................................................... 270
11.5.6 An illustrative example: a single bus architecture ................................................ 272
11.5.7 BNF description of the syntax .............................................................................. 273
11.6 Memory Trace Files .................................................................................................... 275
11.6.1 Usage ...................................................................................................................... 275
11.6.2 Format of memory traces ....................................................................................... 275
11.6.3 Memory trace to actual bus transfer ....................................................................... 276
11.7 Memory map description file ...................................................................................... 277
11.7.1 Usage ...................................................................................................................... 277
Chapter 15. Cosimulation ................................................................. 305
  15.1 Introduction ........................................................................ 305
  15.2 Cosimulation Environment Structure .................................. 305
  15.3 Interface Code Generation .................................................. 306
    15.3.1 Interface Code Generation for Seamless CVE .............. 306
    15.3.2 Interface Code Generation for Virtual Synchronization Scheme ........................................................................ 308
  15.4 Cosimulation Environment Setting For Virtual Synchronization ........................................................................ 309
    15.4.1 eCOS ........................................................................... 309
    15.4.2 Simulation interface ...................................................... 309
      1) ARMulator interface ...................................................... 309
      2) ModelSim interface ....................................................... 310
  15.5 Cosimulation for Trace Generation ...................................... 310
    15.5.1 How to perform cosimulation for trace generation ........ 310
    15.5.2 Generated trace files .................................................. 312
  15.6 Cosimulation for Evaluation of System Performance ........... 312
    15.6.1 How to perform cosimulation for system evaluation ...... 312
  15.7 Coverification ...................................................................... 313
    15.7.1 How to perform coverification with Seamless CVE ....... 313
Chapter 16. C++ and SystemC code generation (Experimental) ... 316
  16.1 C++ code generation target .................................................. 316
    16.1.1 CGC Star parser ......................................................... 316
16.1.2 Running C++ code generation target in Hae .................................................................317
16.2 SystemC code generation target .......................................................................................319
16.2.1 Running SystemC code generation target in Hae .........................................................319
16.2.2 Structure of generated SystemC codes .........................................................................320
Chapter 17. Cosynthesis ..................................................................................................................324
Chapter 18. StarEdit .........................................................................................................................325
18.1 Installation ......................................................................................................................................325
18.1.1 Tarball Installation ...............................................................................................................325
18.1.2 Individual Installation ..........................................................................................................325
18.2 Starting StarEdit ..........................................................................................................................326
18.2.1 Starting jEdit .........................................................................................................................326
18.2.2 Activating StarEdit Plugin ....................................................................................................326
18.3 StarEdit Basics .............................................................................................................................330
18.3.1 Interface Overview ..............................................................................................................330
18.3.2 Tree View ...........................................................................................................................332
18.3.3 Code View ........................................................................................................................333
18.3.4 Wizard View .......................................................................................................................333
18.3.5 Switching Code / Wizard View .........................................................................................334
18.3.6 Configuration ......................................................................................................................334
18.4 Star Sections Supported in StarEdit .........................................................................................335
18.5 Working With Star Files .............................................................................................................337
18.5.1 Creating a new Star File ......................................................................................................337
18.5.2 Editing a Star File ...............................................................................................................339
18.5.3 Editing Remote Files .........................................................................................................342
18.6 Making Star Code With Wizard ..............................................................................................344
18.6.1 Star Attributes ...................................................................................................................345
18.6.2 Input, Output Port ..............................................................................................................346
18.6.3 State ....................................................................................................................................349
18.6.4 Codeblock .........................................................................................................................350
18.7 Import Star With Hae ................................................................................................................353
Chapter 19. Utilities ..........................................................................................................................354
19.1 XMLConverter ........................................................................................................................354
19.1.1 Usage ..................................................................................................................................354
19.1.2 Syntax description of input file .......................................................................................355
19.1.3 Example of an input file ...................................................................................................358
19.2 makeStarSrcPath: find the source path of all blocks ..........................................................359
19.3 dbUpdate: update the CGC block performance database .....................................................360
19.4 peacepasswd: update a user's authentication tokens ..............................................................360
Index ..................................................................................................................................................362
List of Figures

Figure 1-1. Traditional design flow of embedded systems .................................................. 2
Figure 1-2 Current practice of HW/SW codesign .............................................................. 2
Figure 1-3 PeaCE re-configurable design flow ................................................................. 3
Figure 1-4 PeaCE Software Structure ............................................................................. 6
Figure 2-1 "Start PeaCE" dialog – Main panel ................................................................. 12
Figure 2-2 "Start PeaCE" dialog - Option panel .............................................................. 13
Figure 2-3 PeaCE window ............................................................................................... 13
Figure 2-4 Open dialog ................................................................................................. 14
Figure 2-5 Butterfly demo schematic ............................................................................ 14
Figure 2-6 Run butterfly dialog ..................................................................................... 15
Figure 2-7 Generated C code (butterfly.c) .................................................................... 15
Figure 2-8 Xgraph output of butterfly demo .................................................................... 16
Figure 2-9 New dialog .................................................................................................. 17
Figure 2-10 New schematic .......................................................................................... 17
Figure 2-11 Design browser window of demo ............................................................... 18
Figure 2-12 Library browser ......................................................................................... 19
Figure 2-13 Ramp block in the schematic window ....................................................... 19
Figure 2-14 SPDF blocks placed in the schematic window .......................................... 20
Figure 2-15 Ramp block states ..................................................................................... 20
Figure 2-16 Const block states ..................................................................................... 20
Figure 2-17 cosTest application .................................................................................. 21
Figure 2-18 Run cosTest dialog ................................................................................... 21
Figure 2-19 Generated C code (cosTest.c) ................................................................... 22
Figure 2-20 Xgraph for cosTest.................................................................................. 22
Figure 3-1 Option tab in Start PeaCE dialog ................................................................. 23
Figure 3-2 PeaCE window ............................................................................................ 25
Figure 3-3 new project created .................................................................................... 28
Figure 3-4 Design browser (a) before expanding folders (b) after expanding folders ... 30
Figure 3-5 Available domain lists ................................................................................ 31
Figure 3-6 Available target lists for CGC domain ......................................................... 32
Figure 3-7 Target Parameters of the default-CGC target ............................................. 33
Figure 3-8 Run dialog .................................................................................................. 33
Figure 3-9 Running the simulation design step ............................................................. 34
Figure 3-10 Architecture setting in architecture design step ........................................ 34
Figure 3-11 Library browser (a) before expanding folders (b) after expanding folders by clicking a dotted circle (c) after expanding folders by double clicking a folder ................................................. 35
Figure 3-12 Loading a block in the schematic window ................................................ 36
Figure 5-17 Generated code with bufferSharing option is set to “YES” ..................................................87
Figure 5-18 Register the block ..................................................................................................................88
Figure 5-19 Create the downImageTest ....................................................................................................89
Figure 5-20 downImage test .......................................................................................................................89
Figure 5-21 Add states ..................................................................................................................................90
Figure 5-22 Change block states .................................................................................................................91
Figure 5-23 Generated code .........................................................................................................................92
Figure 5-24 Result for downImage .............................................................................................................92
Figure 5-25 Down sample in SDF ................................................................................................................93
Figure 5-26 Down sample in FRDF .............................................................................................................93
Figure 5-27 DetImage demo .........................................................................................................................94
Figure 5-28 Inconsistent composite data type usage ..................................................................................94
Figure 5-29 Access sequence rule. If array a is accessed as a unit of [8][16][16] then call
setAccessSequence(“[8][16][16]”). ..........................................................................................................95
Figure 5-30 example of converting SDF down sample block to an FRDF block ........................................96
Figure 5-31 Register the block ....................................................................................................................97
Figure 5-32 Build a demo using FRDownSample block ............................................................................98
Figure 5-33 Matrix and image blocks .........................................................................................................99
Figure 5-34 Matrix test demo ....................................................................................................................100
Figure 5-35 Result for matrix test demo .....................................................................................................101
Figure 5-36 BlendImage demo ....................................................................................................................101
Figure 5-37 A simple SDF graph ...............................................................................................................102
Figure 5-38 Modified graph with a Piggyback block ..................................................................................103
Figure 5-39 If-then-else construct representation in PeaCE .......................................................................105
Figure 5-40 for construct in PeaCE ..............................................................................................................106
Figure 5-41 Block definition of DownCounter block ................................................................................107
Figure 5-42 A simple demo of nested dynamic constructs ..........................................................................108
Figure 5-43 An example graph using interactive display and simulation control ......................................109
Figure 5-44 An example that uses two HOF blocks and bus connections ..................................................110
Figure 5-45 butterfly demo .........................................................................................................................112
Figure 5-46 range.txt ..................................................................................................................................113
Figure 5-47 Setting “desired SQNR” to 30.0 ...............................................................................................113
Figure 5-48 FixedEstimation target ............................................................................................................114
Figure 5-49 (a) Result of fixed-point simulation (b) Result of floating-point simulation .........................114
Figure 5-50 Simple Matlab Demo ..............................................................................................................115
Figure 5-51 Matlab graph ...........................................................................................................................116
Figure 5-52 Ptc xterm ..................................................................................................................................116
Figure 5-53 Display Heart demo .................................................................................................................117
Figure 5-54 Matlab graph ...........................................................................................................................117
Figure 6-1 New schematic ............................................................................................................................125
| Figure 6-42 Setting Parameters of Blocks : H263 | 164 |
| Figure 6-43 Results of running : H263 Decoder | 165 |
| Figure 7-1 A simple demo of a DE system | 167 |
| Figure 7-2 Definitions of (a) the Ramp block and (b) the newly created PWM block | 169 |
| Figure 7-3 Screen snap-shot after inserting the newly defined PWM block | 170 |
| Figure 7-4 Sampler demo in Peace/Demo/De/Basic | 174 |
| Figure 7-5 de/wormhole/worm demo | 175 |
| Figure 7-6 “adder” demo | 176 |
| Figure 7-7 “sampled_adder” demo | 177 |
| Figure 7-8 simple RC circuit demo | 177 |
| Figure 7-9 Parameters of OctaveODESolver block for simple RC circuit | 178 |
| Figure 7-10 Parameters of OctaveODESolver block for van der Pol equation | 179 |
| Figure 8-1 A simple state transition graph | 181 |
| Figure 8-2 Locating state blocks | 181 |
| Figure 8-3 Setting state block states | 182 |
| Figure 8-4 Drawing transition lines | 182 |
| Figure 8-5 Setting condition and actions | 183 |
| Figure 8-6 Event definition | 183 |
| Figure 8-7 Example of event description | 185 |
| Figure 8-8 Concurrency expression | 186 |
| Figure 8-9 Concurrent FSMs | 186 |
| Figure 8-10 Hierarchy expression | 187 |
| Figure 8-11 Child FSM | 187 |
| Figure 8-12 Parent FSM | 188 |
| Figure 8-13 Interval event expression | 188 |
| Figure 8-14 Internal event example | 189 |
| Figure 8-15 Internal event across hierarchical FSMs | 190 |
| Figure 8-16 Variable state expression | 190 |
| Figure 8-17 Variable state example | 191 |
| Figure 8-18 Reflex Game FSM demo | 192 |
| Figure 8-19 Top level fFSM graph specification of reflex game demo | 193 |
| Figure 8-20 Child fFSM graph specification in the GameOn state | 193 |
| Figure 8-21 Simple Counter Demo | 194 |
| Figure 8-22 Changing the state of a computation module | 195 |
| Figure 8-23 Changing block parameters | 195 |
| Figure 8-24 Using a script in the state | 196 |
| Figure 8-25 Script language usage example | 197 |
| Figure 8-26 resetFSM | 197 |
| Figure 8-27 Parent FSM and related parameters | 199 |
| Figure 8-28 Child FSM and the states | 200 |
Figure 18-8 Popup menu on tree view: (a) click on state, (b) click on background
Figure 18-9 Switching view
Figure 18-10 Configure options
Figure 18-11 Make a new star
Figure 18-12 Star wizard
Figure 18-13 Star wizard generates a star code
Figure 18-14 Save the new star file
Figure 18-15 Buffer switcher
Figure 18-16 Popup menus
Figure 18-17 Delete state
Figure 18-18 Plugin manager
Figure 18-19 Download and Config FTP
Figure 18-20 Connect to the secure ftp server
Figure 18-21 After connecting a remote machine
Figure 18-22 Input port wizard
Figure 18-23 State wizard
Figure 18-24 Codeblock wizard
Figure 18-25 Generated codeblock
Figure 19-1 Result of executing ‘XMLConverter’
Figure 19-2 Input file format
Chapter 1. Introduction

*Author: Soonhoi Ha*

PeaCE is a system-level design environment for digital embedded systems based on hardware-software codesign methodology. It allows the user to model and simulate the system behavior in functional level and to synthesize software codes (C-code) or HDL codes for a given target architecture. There are good commercial tools that can model and simulate the system behavior, among which Matlab/Simulink is probably most well known. But, there are few, if any, that can synthesize a good quality code, either software or hardware, from the initial functional specification. PeaCE provides a unified framework that bridges the gap between functional simulation and embedded system synthesis. It includes optimal architecture selection, HW/SW partitioning, HW/SW co-simulation, and so on. The most prominent feature of PeaCE is reconfigurability: third-party design tools can be easily integrated with the environment thanks to its modular structure. It means that PeaCE provides a collection of design tools that can be used separately or collectively following the proposed design flow.

NOTE: While PeaCE provides various capabilities related with the overall design procedure from modeling to synthesis, you may need only some of them. Then, please figure out the related chapters of your interest and read them only.

### 1.1 Who may want to use PeaCE?

The main objective of PeaCE is to lessen the burdens of embedded system designers. If you feel the following burdens, PeaCE will help you.

1. From system specification, I have to find out the optimal algorithm to perform the desired behavior. I want to reuse predefined blocks as much as possible. I want to compare many alternatives in terms of performance. It is model-driven specification and functional simulation.

2. After functional simulation, I have to implement the selected algorithm on the processor. Developing a bug-free C-code is not an easy task. Moreover, I want to minimize the resource requirement since the target embedded system is very cost sensitive. Is there a CASE tool for embedded software development? It is C-code generation from the specification.

3. I have to develop a SW code for an embedded system. Somebody else should understand my code and upgrade it in the future. How can I let my code understood by others? It is model-driven specification.

4. I am not good at hardware design, HDL programming. But the full software implementation of a given algorithm does not satisfy the performance constraint. What can I do? It is HDL-code generation from the specification.

5. I am given an algorithm and constraints on performance and resource usage. I have to find out the optimal target architecture. Can you help me? It is design space exploration.

6. I’d like to find out an optimal mapping of a given algorithm to a given platform. And I’d like to determine which hardware IPs to be used in the platform. It is HW/SW partitioning.
7. I have to evaluate the performance for the final design that consists of multiple processor cores as well as hardware IPs. It is HW/SW cosimulation.

If you want to use PeaCE as a codesign tool, please skip the next two sections.

### 1.2 HW/SW Codesign Methodology and Current Practice

Let’s look at the traditional design flow of embedded systems in Figure 1-1.

![Figure 1-1. Traditional design flow of embedded systems](image)

It has the following problems that the HW/SW codesign methodology aims to solve.

1. System evaluation is done after the SW code is ported on the hardware prototype. It forms a very expensive and inefficient design loop if further debugging or optimization is requested. HW/SW codesign evaluates the system performance by HW/SW co-simulation on a virtual prototyping environment without a real hardware prototype.

2. System architecture is decided based on the past experiences or simple profiling. HW/SW codesign provides a systematic way of design space exploration.

Recently HW/SW co-simulation tools for HW/SW co-design have been introduced. The current practice of HW/SW co-design is based on these tools as shown in Figure 1-2.

![Figure 1-2 Current practice of HW/SW codesign](image)
It has the following problems that PeaCE aims to solve.

1. Each design step is isolated so that integration of design tools is burdensome. PeaCE provides seamless co-design flow from functional simulation to system synthesis.

2. Design flow heavily depends on which design tool is selected. PeaCE is a re-configurable framework to which 3rd party design tool can be easily integrated.

3. Design space exploration should be done manually using a TLM simulation tool. PeaCE includes an interactive DSE framework that consists of HW/SW partitioning and communication architecture optimization.

1.3 PeaCE Codesign Flow

The overall HW/SW codesign flow in PeaCE is shown in Figure 1-3.

Figure 1-3 PeaCE re-configurable design flow

The key feature of PeaCE codesign flow is reconfigurability. The rectangular boxes describe the design steps while the rounded boxes indicate the input and output xml files of the design steps. Each design step is modularized so that various algorithms or other CAD tools can be integrated with a minimal effort of wrapper design that translates the xml files. The numberings indicates the sequence of the design steps associated with design taps in the user interface.
The design flow begins with specification of the system behavior with formal models: a dataflow model for computation and an FSM model for control module of the system. At the top level, a task model is used to model the interaction between tasks. PeaCE graphical user interface, called Hae, allows the user to draw a hierarchical block diagram reusing as many predefined blocks as possible. Such design reuse reduces the design time. Since each block diagram is drawn with a well-defined execution rule the design can be easily understood without help of the design originator. It makes the design maintenance easy and enables people work together without misunderstanding. But it pays some cost that the designer should take efforts to learn how to specify the system: this is why this user manual is needed.

NOTE: The current release provides only the skeleton of the overall design flow: it does not contain full-fledged capability of each design step but only the primitive one compared with a corresponding commercial tool. More capabilities on each design step will be added in the future.

Step 1: Functional simulation of system behavior, possibly integrated with other tools, MATLAB and Octave for example. We generate a C code from the specification and compile and run it in the host machine for simulation. This capability is comparable to that of SPW or Simulink so that the user may replace SPW or Simulink with PeaCE. The main difference between PeaCE and others is that PeaCE is not a simulation engine but a code generation tool even for functional simulation. It makes simulation fast and debugging easy.

Step 2: We specify the candidate architectures to be explored. In this step, we list all processing elements, processor cores and hardware IPs, that are available in the design library. Or we select the platform to be used. The current release does not allow the user to select a specific platform, which will be corrected in the next release.

Step 3: We estimate the software performance of each function block on each candidate processor core if it is not available in the design library. In this step, we use an ISS (instruction set simulator) of the processor core. The user may skip this step if all performance estimates of function blocks are already recorded in the library.

Step 4: This step is to translate the graphical specification into three sets of textual files, which describe the graph topology, block-performance information, and the timing constraints of tasks. Such translation modularizes PeaCE so that the following design steps do not depend on the PeaCE kernel and PeaCE user interface. A third party tool can be used in PeaCE seamlessly if it can read these textual files.

Step 5. We perform HW/SW partitioning and component selection at the same time in this step. The output is written in a text file, which describes the mapping and scheduling of function blocks onto each processing element. Even though PeaCE provides an HW/SW partitioning tool, the user may use another tool thanks to the file interface mechanism.

Step 6. After partitioning is made, PeaCE generates partitioned codes and co-simulate the system to generate the memory traces from the processing elements. PeaCE provides a co-simulation tool based on the virtual synchronization technique, which is fast and time-accurate. At this step, it is assumed that communication architecture is not determined yet. Using the memory traces and schedule information, we explore the bus architectures. Later we will allow other communication architectures to be explored. The final bus architecture is recorded in a text file, archi.xml.

Step 7: After the bus architecture is determined, we want to verify the final architecture before synthesis. This step performs time-accurate HW/SW co-simulation for co-verification. We may use Seamless CVE in this step or our co-simulation tool used in Step 6 but using accurate modeling of bus architecture and OS.
Step 8: This step is to generate the codes for the processing elements in a prototyping board. For a processor core, we generate a C code and a VHDL code for FPGA hardware implementation.

In steps 6 through 8, we need to generate codes according to the partitioning decision made in step 5. It means that we need to generate the interface codes between processing elements and the wrapper code for the simulation tool (step 6 and step 7) or for the prototyping board (step 8). If you want to use a different simulation tool or a prototyping board, a new target object and interface blocks should be prepared, which is beyond the user's manual.

1.4 History

PeaCE has been developed as a research prototype for hardware-software codesign environment in the CAP laboratory of Seoul National University, Korea since 1995. PeaCE stands for “Ptolemy extension as Codesign Environment”. As the acronym indicates, PeaCE is based on Ptolemy project (Ptolemy classic) which had been developed in U.C. Berkeley during 1990 -1995 (http://ptolemy.eecs.berkeley.edu). Ptolemy has advocated the usage of formal models of computation for system specification and it has been renowned for its capability of seamless integration of diverse models of computation. While Ptolemy is a good tool for system modeling and simulation, its code synthesis capability is quite restricted for system implementation. Ptolemy could produce a C code from a dataflow representation but the code quality is not satisfactory. It could barely produce a synthesizable VHDL code, but not optimized at all. There is no way of architecture synthesis from Ptolemy. To solve all these problems, PeaCE project was started.

The PeaCE project has been supported by Korea government through NRL (National Research Lab) grant, BK21 project, IT-SoC program, and KOSEF(Korea Science and Engineering Foundation) project. Since it has been developed as a research prototype, it is not easy for a casual user to use this tool. But the CAP laboratory goes only forward to solve new research problems for embedded system design. So, we need a special effort to make the tool accessible and easily usable. On the other hand, the copyright policy of PeaCE is the same as the Ptolemy project. It is an open-source program. Anybody can use or replicate this program as long as he/she acknowledges the tool developers at his/her responsibility of program management and support.

Since it is built on top of Ptolemy, PeaCE basically inherits all nice features of Ptolemy: seamless integration of diverse models of computations, powerful simulation capabilities for DSP applications, and much more. However, PeaCE differs from Ptolemy that it does not allow arbitrary integration of heterogeneous models of computation. Instead, PeaCE supports only three models of computation: dataflow model, FSM model, and task model. It is all right if you do not understand what those models are at this moment. That is why this manual is written for. Through chapters 4 and 9, you will see what they are.

1.5 Software Structure

PeaCE has a server-client architecture as shown in Figure 1-4. PeaCE server is developed in a Linux platform though it can run in a Solaris or a Windows machine. PeaCE server is programmed in C++. PeaCE client is written in Java to provide a platform independent user interface. We named the PeaCE client program as “Hae” which has double meanings in Korean: “sun” and “solution”.
The server consists of three modules: daemon process, server kernel, and a block/component library. To use PeaCE, you should first install those modules in the server machine. See the next section for more details of installation procedure. After installation, the daemon process should be run in the server. When you run the daemon process, you specify an unused port number for socket connection.

On the other hand, a Java client program is installed in any client machine. When you start the client program, it is connected with the daemon process of the server side via the designated port. Then, the daemon process forks the server process and makes the server process connected to the client process. Now, you are ready to go on. The client program accepts user requests and delivers them to the server. The response from the server is displayed in the client side using an X window or a tk window.

The database in the Figure represents the block library and the component library. The current version of the server uses a file for the block library for simple installation. If a database program, such as postgres or MySQL, is used, you should install the database program as a separate process. If the number of blocks is not huge, file implementation is just good for use.
1.6 Installation

The officially supported platform is

- Red Hat 8.0 Linux on x86 machine

PeaCE package includes the following programs:

gcc-2.95.3
libg++-2.8.1.3
tcl 7.6 / tk 4.2
xgraph
xv
octave-2.0.17

1. PeaCE

If you run the server and the client in the same Linux machine, then installation can be done through the following steps:

1. add a new user of “peace” by root
   
   machine % su
   
   machine % adduser –s /bin/tcsh peace

2. login as “peace” and extract the sources
   
   machine % su – peace
   
   machine % cd ..
   
   machine % tar xvfz peace.tgz

3. configure .cshrc for your system
   
   machine % source .cshrc
   
   $PEACE and $PTOLEMY is home directory of the PeaCE(Ptolemy) installation. If you want to install other directory, modify this variables. $PTARCH is target architecture.
   
   Modify some variables for your system - ecos, armsd, ModelSim and seamless CVE.

4. Build and install the PeaCE
   
   machine % ./bin/mkPeaCE
   
   Build and install the PeaCE. You can use shell script - bin/mkPeaCE. If there are any problems compiling the PeaCE, you can refer detail installation guide.
5. installation is completed. Refer to “Getting Started” to start the PeaCE.

2. HAE

If you want to use a separate client, you can install the server from the above steps. And you can install the client program as follows:

1. install j2sdk1.4.1 by double clicking j2sdk-1_4_1_02-windows-i585.exe from windows directory that is also downloadable from http://java.sun.com.

2. install Hae by unzipping Hae.zip from windows directory.

* Detail PeaCE installation guide of step 4

This is detail description of ‘mkPeaCE’ shell script. If there are any problems compiling the PeaCE or if you want to building any steps, refer this guide.

**Preparing for building**

```
cd SPEACE/src
../bin/cpMakefile
.cd..
./MAKEARCH
```

After this step, you can see bin.$PTARCH, lib.$PTARCH, obj.$PTARCH directory.

**Build and install GCC 2.95.3**

```
cd obj.$PTARCH
cd gnu
make
```

For building octave 2.0.17, ptcl and PeaCE, we must use gcc 2.95.3.

**Build and install octave 2.0.17**

```
rehash
cd SPEACE/obj.$PTARCH/octave
```
make

To use GCC 2.95.3, first run ‘rehash’. And type ‘gcc -v’ to confirm gcc version. If gcc version isn’t 2.95.3 then check path configuration of ‘.cshrc’ and previous step - gcc 2.95.3 compilation.

**Build and install tcl7.6/tk4.2**

```bash
cd $PEACE/obj.$PTARCH/tcltk
make
```

**Build and install PeaCE Kernel**

```bash
cd $PEACE/src
make depend
cd ../obj.$PTARCH
make install -j 32
make install
```

*arm-elf-gcc (cross-compiler)*

arm-elf-gcc is used as a default cross-compiler for arm processor in PeaCE. It is recommended to build arm-elf-gcc in your own machine but PeaCE provides prebuilt binary in

```bash
$PEACE/arm-elf/.
```

The prebuilt binary must be located in the directory /opt/gnutools/arm-elf/3.4.3/bin/.
If you want to build the binary from the source, refer the site http://www.gnuarm.org/
1.7 Terminology

Here we explain some terminologies you will frequently encounter in PeaCE source.

**Domain:** an environment object in which a block diagram sits. It indicates which model of computation is used. FSM domain means that the block diagram follows the FSM semantics. CGC and VHDL domains use SPDF model and produce a C code and a VHDL code from the block diagram respectively.

**Galaxy:** a super-block that contains a block diagram inside. The inside block diagram is also called a “galaxy”. As the name implies it consists of stars or blocks.

**Scheduler:** an object that determines the execution order of blocks in a block diagram according to the semantics of the specified model of computation.

**Star:** an atomic block. Star is coined in Ptolemy project and remains as a class name in the server program.

**State:** An object is associated with parameters and state values. We do not distinguish parameters and (internal states) in PeaCE and call them all “states”. Instead, parameters and internal states are distinguished by state attributes. If a state is A_SETTABLE and A_CONSTANT, it indicates a parameter. If a state is A_NONCONSTANT, then the state value may be updated after each block execution. Then, it represents an internal state. Fortunately, you need not make such a distinction in using PeaCE.

**Target:** an object that manages the process of block diagram execution customized for a specific target. For example, when you generate a C code for a PC or an embedded system, we need to apply different optimization technique. Such customization is performed by the Target object.

**Tcl/Tk:** Tcl is an interpreted "tool command language" designed by John Ousterhout while at UC Berkeley. Tk is an associated X window toolkit. Both have been integrated into Ptolemy, so PeaCE. Parts of the graphical user interface and all of the textual interpreter ptcl are designed using them. Several of the blocks in the standard block library also use Tcl/Tk. Documentation is provided along with the PeaCE distribution in the $PEACE/tcltk/itcl/man directory in Unix man page format.
Chapter 2. Getting Started

Author: Soonhoi Ha

2.1 Starting PeaCE

PeaCE has a server-client architecture. The server runs the PeaCE daemon process and the PeaCE kernel (Ptcl). To get started, you should first run a PeaCE daemon on the service. Next, run a PeaCE client in any client machine. Since the PeaCE daemon invokes the PeaCE kernel, there is no need to run the kernel explicitly. The current version of the server uses a file for the block library.

2.1.1 Running PeaCE daemon

Environment variable DISPLAY setting

The environment variable `DISPLAY` is needed to execute PeaCE kernel. In your home directory of the server machine, type:

```
setenv DISPLAY     or     setenv DISPLAY localhost:0
```

Running PeaCE daemon

PeaCE daemon uses a designated port to communicate with PeaCE client. To run PeaCE daemon in your server machine with port number 5555, type:

```
peacedaemon 5555&
```

Setting user password file

To access PeaCE daemon from the client program, user authentication password should be set. To set your PeaCE password, type:

```
peacedpasswd
```

Then you can see following messages:
Changing password
New PeaCE password:
Retype New PeaCE password:

Then type your password twice.

2.1.2 Running PeaCE client (HAE)

HAE is the graphical user interface of PeaCE. The client machine on which you want to run a PeaCE client must have a java interpreter. To pop-up a “Start PeaCE” dialog in your client machine, change current directory to where HAE is installed, and type:

gohae

Then the following dialog box will appear (Figure 2-1):

![Start PeaCE dialog – Main panel](image)

To start PeaCE:

1. Ensure that the **Main** tab is selected. Type the server machine IP in the Server IP field. If the server machine is same as the client machine, type **localhost**. Type the user ID in the User ID field. Type the password in the Password field.

2. Click the **Option** tab. The Option panel is displayed(Figure 2-2). Type PeaCE daemon port number, e.g. **5555**, in the Port Number field. If you do not execute an external X server such as eXceed or Xmanager, check the XServer checkbox.
3. Click the **Main** tab again. Click the **Start PeaCE**. Then, the PeaCE window will be displayed as shown in Figure 2-3.
2.2 Running a demo application

This section describes how to run an existing demo application. Please ignore the computation model of the block diagram for now.

2.2.1 Opening a demo application

To open a demo application:

1. Select **Open** from the **File** menu. An Open dialog will be displayed.
2. Navigate to the CGC\Basic directory in the \{install_directory}\schematic\Peace\Demo directory and click **butterfly** to select it (Figure 2-4).

![Figure 2-4 Open dialog](image)

3. Click **Open**. PeaCE displays the demo schematic in the main window (Figure 2-5).

![Figure 2-5 Butterfly demo schematic](image)
2.2.2 Running the demo application

To run the demo application:

1. Select **Run It!** from the **Run** menu. A Run butterfly dialog is displayed (Figure 2-6).

![Figure 2-6 Run butterfly dialog](image)

2. Click **Go**. A generated source code (**butterfly.c**) will be displayed (Figure 2-7). The code will be compiled and run in the host machine. An xgraph is displayed as a result (Figure 2-8). This demo illustrates how functional simulation is performed in PeaCE.

![Figure 2-7 Generated C code (butterfly.c)](image)
2.3 Creating a simple application

This section describes how to create a simple application in PeaCE. Let’s make the following function:

\[ y = \cos(0.2 \times x) + 0.4 \]

2.3.1 Creating a new schematic

First, you have to create a new schematic to draw a block diagram:

1. Select New from the File menu. A New dialog will be displayed.
2. Navigate to the directory where you want to save the schematic and enter a schematic name, for example cosTest (Figure 2-9).
3. Click **Open**. PeaCE will create a new schematic and open it in the main window (Figure 2-10).

![Figure 2-9 New dialog](image)

![Figure 2-10 New schematic](image)

### 2.3.2 Configuring domain and target

After an empty schematic window is open, select the domain and the target of the design. For a computation task, you will draw a dataflow graph and generate a C code from the specification. The domain would be “CGC (Code Generation in C).” And the target is the default target that indicates functional simulation in the host machine.

To select the domain and the target, click the **Design** tab. Then you will get a browser window as shown in Figure 2-11. It contains a domain drop-down list and a target drop-down list and target states. See the next chapter for more information on domain and target and target states. Make sure that CGC is selected in domain drop-down list and default-CGC is selected in target drop-down list. Leave all the target states as they are.
2.3.3 Using blocks in SPDF library

The dataflow model used in PeaCE is called SPDF(Synchronous Piggybacked DataFlow). Chapter 4 is devoted to the SPDF domain. Since the “cosTest” example will be composed of SPDF blocks, first search for the library blocks to be reused. If you cannot find a library block, you have to create one. But in this example you are able to locate all the library blocks you will use. Now let’s figure our what blocks you need for the equation:

\[ y = \cos(0.2 \times x) + 0.4. \]

First you need a block to generate the argument of \( \cos \) function. The Ramp block is used to generate a sequence of argument values. Constant value 0.4 is provided by the Const block. And you need two operator blocks: Add and Cos.

To instantiate a Ramp block, do the following steps:

1. Click the Lib tab. The library browser will be displayed (Figure 2-12).
2. Navigate to the Src directory in the SPDF directory and click Ramp.ilink to select it.
3. Place the Ramp block in the schematic window (Figure 2-13) by moving your mouse to the desired location.

Place following blocks similarly (Figure 2-14).
- SPDF\Src\Const.ilink
- SPDF\Math\Cos.ilink
- SPDF\Math\Add.ilink
- SPDF\Sink\Xgraph.ilink
2.3.4 Editing block states

Now you should edit the block states if needed. In this demo, you have to edit the states of the Ramp block of which the step state is 0.2 and the value state is 0. And you edit the “level” state of the Const block to 0.4.

To edit the block states, do the following:

1. Ensure that the Design tab is selected.
2. Click the Ramp block to select it. The block states will be displayed in the Design browser window.
3. Type 0.2 in the value field of the step state (Figure 2-15).

![Figure 2-15 Ramp block states](image)

4. Click the Const block to select it and type 0.4 in the value field of the level state (Figure 2-16).

![Figure 2-16 Const block states](image)

2.3.5 Connecting blocks
Now you connect the blocks. To connect two blocks, Ramp and Const, do the following:
1. Move the mouse pointer to the output port of the Ramp block. You can see a green circle and the port name.
2. Click the circle to start drawing a line from the port.
3. Move the mouse pointer to draw a line as you wish. If you want to make a turn, click at the turning point.
4. Click the circle of the input port of the Cos block to complete the line.

Complete the other lines similarly (Figure 2-17).
- From the output port of the Cos block to the input port of the Add block.
- From the output port of the Const block to the input port of the Add block.
- From the output port of the Add block to the input port of the Xgraph block.

![Figure 2-17 cosTest application](image)

### 2.3.6 Running the application

To run the application:
1. Select **Run It!** from the Run menu. A Run cosTest dialog is displayed.
2. Type 100 in the When to stop field (Figure 2-18).

![Figure 2-18 Run cosTest dialog](image)
3. Click Go. A generated source code (cosTest.c) will be displayed (Figure 2-19), compiled and run in the host machine. An xgraph is displayed as a result (Figure 2-20).

Figure 2-19 Generated C code (cosTest.c)

Figure 2-20 Xgraph for cosTest

PeaCE has many built-in demos in schematic\Peace\demo directory. If you want to get an idea how to use PeaCE, we recommend you to run some demos. Some demos give you exciting outputs. Drawing a block diagram is basically what you have to do in PeaCE if all the blocks you need are already existent in the block library. Since PeaCE is not a commercial tool, not many library blocks are provided. So you should know how to create your own blocks.
Chapter 3. Graphical User Interface

Author: Hyunok Oh, Kyungjoo Oh, and Soonhoi Ha

In order to support multiple platforms, the graphical user interface of PeaCE called HAE is written in the Java language (Java2). Since HAE is written in Java, it can be run in any platform that supports Java (jdk1.2.2 over). This section explains all the features of HAE to be referred when needed. So just look over this chapter to see what features are included. The current version of HAE is 1.8b, and the build date is 2004-11-15

3.1 Running PeaCE

3.1.1 Starting options

In this section, we will discuss more details on options in PeaCE start-up dialog. As explained in the previous chapter, to pop-up a Start PeaCE dialog, change the current directory to where HAE is installed, and type “gohae” and click Option tab to get the window as shown Figure 3-1.

Figure 3-1 Option tab in Start PeaCE dialog
The options are listed below.

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server IP</td>
<td>The machine IP or name on which a PeaCE daemon is running or a PeaCE server</td>
</tr>
<tr>
<td>User ID</td>
<td>Your ID on the server machine.</td>
</tr>
<tr>
<td>Password</td>
<td>Your password on the server machine.</td>
</tr>
<tr>
<td>Client IP</td>
<td>The machine IP or name on which PeaCE client (HAE) is running.</td>
</tr>
<tr>
<td>use daemon</td>
<td>Is a PeaCE daemon running? Then check it.</td>
</tr>
<tr>
<td>use rexec</td>
<td>If you don’t run a PeaCE daemon and want to execute a PeaCE server remotely</td>
</tr>
<tr>
<td>Port Number</td>
<td>If ‘use daemon’ is checked, it is a daemon port number;</td>
</tr>
<tr>
<td>normal server</td>
<td>Default option. Execute a PeaCE server (Ptcl) before running a GUI (HAE).</td>
</tr>
<tr>
<td>lazy server</td>
<td>Run a GUI (HAE) without executing a PeaCE server. The server is activated</td>
</tr>
<tr>
<td>no server</td>
<td>It does not run the server but creates a text file describing the block</td>
</tr>
<tr>
<td>Peace Server</td>
<td>PeaCE server program: ‘ptcl’. If you know how to debug the server program,</td>
</tr>
<tr>
<td>Debugger</td>
<td>External debugger. Currently we have tested gdb and ddd.</td>
</tr>
<tr>
<td>XServer</td>
<td>When any x server is running on your client machine, don’t check it.</td>
</tr>
<tr>
<td>debug</td>
<td>If this field is checked, a debugger will be invoked.</td>
</tr>
</tbody>
</table>

The server related options are rather complicated. Please leave them as they are.

### 3.1.2 Overview of PeaCE window

Now we examine the main window that will come up when you click “Start PeaCE”. PeaCE window mainly consists of three parts: design and library browser, main window, log window (Figure 3-2)
3.1.3 Menus

PeaCE provides MS Windows style menus. Some menus are not explained below since they exist for developer’s convenience. Please ignore them when you use PeaCE.

<table>
<thead>
<tr>
<th>Menu</th>
<th>Submenu</th>
<th>Short key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>New Project</td>
<td>Ctrl+J</td>
<td>create a new project.</td>
</tr>
<tr>
<td></td>
<td>New</td>
<td>Ctrl+N</td>
<td>create a new schematic including palette, application or supper-block.</td>
</tr>
<tr>
<td></td>
<td>Open</td>
<td>Ctrl+O</td>
<td>open a project or a single schematic including palette, application or supper-block.</td>
</tr>
<tr>
<td></td>
<td>Close</td>
<td>Ctrl+D</td>
<td>close the selected window.</td>
</tr>
<tr>
<td></td>
<td>Save</td>
<td>Ctrl+S</td>
<td>save the selected window.</td>
</tr>
<tr>
<td><strong>Save as</strong></td>
<td>Ctrl+A</td>
<td>rename and save the selected window.</td>
<td></td>
</tr>
<tr>
<td><strong>Save All</strong></td>
<td>Ctrl+W</td>
<td>save all activated windows.</td>
<td></td>
</tr>
<tr>
<td><strong>Export</strong></td>
<td>Alt+E</td>
<td>export a schematic. All blocks and super-blocks in the current schematic are zipped into one file. You can easily give your schematics to others using this feature.</td>
<td></td>
</tr>
<tr>
<td><strong>PrintSetup</strong></td>
<td>Alt+R</td>
<td>set printer.</td>
<td></td>
</tr>
<tr>
<td><strong>Print</strong></td>
<td>Ctrl+P</td>
<td>print the selected schematic.</td>
<td></td>
</tr>
<tr>
<td><strong>Exit</strong></td>
<td>Alt+X</td>
<td>close all programs and exit PeaCE.</td>
<td></td>
</tr>
<tr>
<td><strong>Edit</strong></td>
<td><strong>Edit Icon</strong></td>
<td>Ctrl+M</td>
<td>modify the physical appearance of a block or super-block icon.</td>
</tr>
<tr>
<td><strong>Substitute Icon</strong></td>
<td>Ctrl+B</td>
<td>substitute a block or super-block icon.</td>
<td></td>
</tr>
<tr>
<td><strong>Undo</strong></td>
<td>Ctrl+U</td>
<td>undo any number of previous changes in text editor.</td>
<td></td>
</tr>
<tr>
<td><strong>Redo</strong></td>
<td>Ctrl+R</td>
<td>redo undoing changes in text editor.</td>
<td></td>
</tr>
<tr>
<td><strong>Cut</strong></td>
<td>Ctrl+X</td>
<td>cut the selected objects in a schematic and store it into the clip-board.</td>
<td></td>
</tr>
<tr>
<td><strong>Copy</strong></td>
<td>Ctrl+C</td>
<td>copy the selected objects into the clip-board.</td>
<td></td>
</tr>
<tr>
<td><strong>Paste</strong></td>
<td>Ctrl+V</td>
<td>paste objects from the clip-board onto a schematic.</td>
<td></td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>Delete</td>
<td>delete the selected objects.</td>
<td></td>
</tr>
<tr>
<td><strong>View</strong></td>
<td><strong>Zoom In</strong></td>
<td>Alt+Z</td>
<td>zoom in the schematic.</td>
</tr>
<tr>
<td><strong>Zoom Out</strong></td>
<td>Ctrl+Z</td>
<td>zoom out the schematic.</td>
<td></td>
</tr>
<tr>
<td><strong>Zoom Auto</strong></td>
<td>Ctrl+F</td>
<td>rescale the schematic to fit it in the window.</td>
<td></td>
</tr>
<tr>
<td><strong>Rotate</strong></td>
<td>Ctrl+T</td>
<td>rotate the selected objects.</td>
<td></td>
</tr>
<tr>
<td><strong>Look Inside</strong></td>
<td>Ctrl+I</td>
<td>look inside an icon for its definition.</td>
<td></td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td></td>
<td>Display block instance names and port names.</td>
<td></td>
</tr>
<tr>
<td><strong>Rate</strong></td>
<td></td>
<td>If it is checked, the sdf rates are displayed.</td>
<td></td>
</tr>
<tr>
<td><strong>Run</strong></td>
<td><strong>Run It</strong></td>
<td>F9</td>
<td>run an application in case of a single schematic file.</td>
</tr>
<tr>
<td><strong>Edit Run Parameter</strong></td>
<td>Alt+E</td>
<td>change the termination condition: how many iterations will be run for an SPDF graph. in case of a single schematic file.</td>
<td></td>
</tr>
<tr>
<td><strong>Set Architecture</strong></td>
<td>Alt+A</td>
<td>Set the initial target architecture in case of a single schematic file.</td>
<td></td>
</tr>
<tr>
<td><strong>Tool</strong></td>
<td><strong>Register Galaxy</strong></td>
<td>Ctrl+2</td>
<td>make an icon to represent the schematic as a super-block(galaxy).</td>
</tr>
<tr>
<td><strong>Register Star</strong></td>
<td>Ctrl+8</td>
<td>make an icon to represent a block.</td>
<td></td>
</tr>
<tr>
<td><strong>Link Stars</strong></td>
<td>Alt+K</td>
<td>dynamically link the blocks that have not linked to the server but already have icons.</td>
<td></td>
</tr>
<tr>
<td><strong>Import Star</strong></td>
<td>Alt+I</td>
<td>link a block dynamically with a source path.</td>
<td></td>
</tr>
<tr>
<td><strong>Library</strong></td>
<td><strong>Load Icon</strong></td>
<td>Alt+G</td>
<td>load a icon of block, super-block or application.</td>
</tr>
<tr>
<td><strong>Line</strong></td>
<td>Alt+L</td>
<td>connect a line between two icons.</td>
<td></td>
</tr>
<tr>
<td><strong>Bus Line</strong></td>
<td>Alt+B</td>
<td>connect a bus line between two icons.</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Auto Line Connection</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
<td>--------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Default</td>
<td>Ctrl+F</td>
<td>change the initial domain and the target</td>
<td></td>
</tr>
<tr>
<td>Comment</td>
<td>Rectange</td>
<td>draw a rectangle in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Oval</td>
<td>draw an oval in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Line</td>
<td>draw a line in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Polygon</td>
<td>draw a polygon in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gif</td>
<td>load a gif file in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Text</td>
<td>edit comment in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>InArrow</td>
<td>draw an incoming-arrow in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>InMultiArrow</td>
<td>draw an incoming-arrow with multi-port in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OutArrow</td>
<td>draw an outgoing-arrow in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OutMultiArrow</td>
<td>draw an outgoing-arrow with multi-port in the comment region</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Color</td>
<td>change the color of the selected comment objects and the current color</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fill</td>
<td>fill/empty the selected comment objects and set the fill mode to fill/empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Move To Front</td>
<td>move the selected comment objects over other comment objects</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Move To Back</td>
<td>move the selected comment objects under other comment objects</td>
<td></td>
</tr>
</tbody>
</table>

**3.2 Working with files**

This section gives information on how to use PeaCE to perform basic operations on files. It describes the basic file operations such as opening, closing, saving, and printing files.

**3.2.1 Creating and opening files**

**Creating a new project**

To create a new project file:

1. Select **New Project** from the **File** menu. PEACE displays a **New project** dialog.
2. Select a directory where you want to create a new project file and type the file name in the **File name** field.
3. Click **Open**.

See Figure 3-3. A new project has multiple design tabs.
Creating a new schematic

To create a new schematic file:
4. Select New from the File menu. PEACE displays a New dialog.
5. Select a directory where you want to create a new schematic file and type the file name in the File name field.
6. Click Open.

Opening a file from the File menu

To open a project or a schematic file:
1. Select Open from the File menu. PEACE displays an Open dialog.
2. Navigate to the directory where the file exists and click the file to select it. PEACE checks the file to open and know that the file is a project or a single schematic.
3. Click Open.

3.2.2 Saving files

Saving a project or a schematic

To save your changes to a project or a single schematic file:
1. Ensure that the schematic window you want to save is the active window.
2. Select Save from the File menu.

Saving all files

To save your changes to all the files currently open, select Save All from the File menu.
Renaming and saving a file

To save a project or a schematic file under a new name:
1. Ensure that the schematic window you want to save is the active window.
2. Select **Save as**... from the **File** menu. PeaCE displays a **Save as** dialog.
3. Enter the new name and the location of the file.
4. Click **Save**.

### 3.2.3 Closing files

To close a project or a schematic file:
1. Select **Close** from the **File** menu, or click the close box for the window. If you haven’t saved the changes to the schematic file, PeaCE will ask if you want to save the changes before closing the window.
2. Click one of the followings:
   A. **Yes**, if you want to save your changes before closing the file.
   B. **No**, if you want to close the file without saving your changes. All unsaved changes are discarded.
   C. **Cancel**, if you want to cancel the close command and return to the schematic window without saving your changes.

### 3.2.4 Exporting files

To export a schematic file, select **Export** from the **File** menu. All blocks and super-blocks in the current schematic are zipped into one file. You can easily give your schematics to others by sending the zipped file. This is same for the project file.

### 3.2.5 Printing files

#### Setting print options

To configure printing options:
1. Select **PrintSetup** from the **File** menu. PeaCE displays **Print Setup** dialog for your printer.
2. Use the dialog box to select the paper size, orientation, and other settings. The specific settings and options depend on the printer you have connected to your computer. See your printer and operating system documentation for more information on printer options.
3. Click **OK** to save your selected printer options.

#### Printing a window
To print a window:

1. Ensure that the window you want to print is the active window.
2. Select **Print** from the **File** menu. The **Print** dialog for your printer is displayed.
3. Select your print options. The options available will vary depending on your printer. See the printer and operating system documentation for more information on print options.
4. Click **OK** in the **Print** dialog. PeaCE spools the file to your printing software for printing.

### 3.3 Working with designs

#### 3.3.1 Using design browser

The design browser shows information about the schematic files and blocks of your designs (Figure 3-4(a)).

![Design browser](a) ![Parameter window](b)

**Figure 3-4 Design browser (a) before expanding folders (b) after expanding folders**

The design browser consists of a navigation window and parameter window that includes block parameters.

**Navigating the navigation window in the design browser**

The navigation window represents a design hierarchy as a folder. If you want to navigate the design then just click or double-click the folder like in MS windows. If you choose an internal folder or a block in the navigation window, you also select a block in the main window.

#### 3.3.2 Creating and opening a design

**Creating a new design**
To create a new design:

1. Select **New** from the **File** menu. PeaCE displays a **New** dialog. If you want to make a design project, select **New Project**. This procedure is same to creating a file.
2. Select a directory where you want to create a new design and type the design name in the File name field.
3. Click **Open**.

**Opening an existing design**

To open a design:

1. Select **Open** from the **File** menu. PeaCE displays an **Open** dialog.
2. Navigate to the directory where the design you want to open is and click the file to select it.
3. Click **Open**.

3.3.3 Configuring **domain**

You can configure or change the domain of the selected schematic in the design browser. You can select the schematic simply by clicking any place in the schematic. If you change the domain, target will also be changed accordingly.

To configure domain:

1. Click domain drop-down list to get the list of available domains (Figure 3-5)

![Figure 3-5 Available domain lists](image)

2. Move mouse vertically to highlight the domain you want to select. You may have to use the vertical scroll bar on the right side.
3. Click the domain you want to select.

3.3.4 Configuring target and editing target states
You can configure the target of the selected schematic and edit the target states in the design browser. In case you find the target list disappeared, widen the design browser by moving the border of the design browser to the right.

**Configuring target**

To configure the target:

1. Click the target drop-down list to pop-up the list of available targets for the domain, e.g. CGC (Figure 3-6)

![Figure 3-6 Available target lists for CGC domain](image)

2. Move mouse vertically to highlight the target you want to select. If it contains many items, use the vertical scroll bar on the right side.
3. Click the target you want to select.

**Editing target states**

To edit the target states, type the value you want to set in the value field of each state you want to edit.
3.3.5 Running a design

In case of a single design

After setting the domain and the target of an application schematic, we can run the application. To run a design:

1. Select Run It! from the Run menu. A Run dialog is displayed (Figure 3-8).

2. Type run counts in the When to stop field.

3. Click Go.
In case of a multiple design (project)
Each design step (simulation, architecture, estimation, analysis, partition, dse, cosimulation, cosynthesis) has its own domain, target and Run button. After setting the domain and the target, you can execute each design step by pressing Run button.

If it has a run count field beside Run button, edit run count number and then press Run. (Figure 3-9)

![Figure 3-9 Running the simulation design step](image)

To set the architecture, press the set architecture button in the ‘arch’ design tab. (Figure 3-10)

![Figure 3-10 Architecture setting in architecture design step](image)

### 3.4 Using block libraries

PeaCE provides primitive block libraries for all domains. You can easily find a block you want to load using the library browser. You can edit the block states and make a super-block from a block diagram.
3.4.1 Using library browser

The library browser shows the list of block libraries that both PeaCE provides by default and the user creates later (Figure 3-11 (a)).

![Diagram of library browser with navigation window and library window]

(a) (b) (c)

Figure 3-11 Library browser (a) before expanding folders (b) after expanding folders by clicking a dotted circle (c) after expanding folders by double clicking a folder

The library browser consists of a navigation window and the library window.

Navigating the navigation window in the library browser

You can navigate the library browser in MS windows style. However there is one difference on folder expansion. When you expand a folder by clicking a folder expander that is dotted circle, you can see the block name list as shown in Figure 3-11 (b). However when you expand a folder by double clicking a folder, you can see the list of block icons in the library window as shown in Figure 3-11 (c). Since it takes long time to load block icons, you should wait enough time to see icons. In case of Figure 3-11 (b), if you choose a block in the navigation window then you can see its description in the library window as shown in Figure 3-12. Note that the description is dependent on the domain of the design schematic.

3.4.2 Loading a block (or super-block) from a specific domain library

To load a block from a block library:

1. Click the Lib tab. The library browser is displayed.
2. Navigate to the directory where you want to browse.
3. Click a block icon to select it in the navigation window or library window. Do not drag the mouse.
4. Move the mouse pointer to the desired position in the schematic window. See the block icon following the mouse pointer.

5. Click to place the block where you want to locate in the schematic window (Figure 3-12).

![Figure 3-12 Loading a block in the schematic window](image)

Other tip: When you do not know where the block icon is, select a Load Icon from the Library menu or press Alt+G. Then you can see an open dialog that displays all blocks. You find a block and type the block name or click the block.

### 3.4.3 Editing block (and super-block) states

A block (or a super block) has parameter (and/or state) values. We do not distinguish parameters and internal states in PeaCE and call them all “states”.

Super-block states are globally visible from all the blocks inside the super-block.

To see the state values of a block (or a super-block), select the block (or the super-block). Then the parameter window now displays the block states. Each state is associated with three columns: Name, Value and Type. The “Name” field represents the state name and the “Type” indicates the data type. What you have to change is the value in the “Value” field. In the value field, you may use some predefined constants such as “PI”, “TRUE”, “FALSE”, “YES”, and “NO”.

To edit the block states:

1. Ensure that the design browser is selected by clicking **Design** tab.
2. Click the block you want to edit the states. The block states are displayed in the parameter window as shown in Figure 3-13.
3. Type the value in the value field of the state you want to edit.

The block and super-block state values can be defined hierarchically. Consider an H.263 encoder example in Hae/schematic/Peace/demo/CGC/H263/H263FREncoder. The states of FRDiffImage super-block are displayed when the block is selected (Figure 3-14).

Notice that the values of the width and the height state are symbolic expression, "width" and “height”. These refer to the states of its super-block, in this example H263FREncoder is the super-block. Then the width and the height states of FRDiffImage are inherited from the states of its super-block, H263FREncoder: 176 and 144 respectively as defined in the H263FREncoder super-block states (Figure 3-15). Thus, state values can be passed down through the hierarchy. These symbolic references can appear in expressions, which we discuss next.

### 3.4.4 Parameter expressions

State values can be expressed as arithmetic expressions. This is particularly useful for propagating values down from an application state to block states somewhere down in the hierarchy. An example of a valid state expression is:

\[ \frac{\pi}{2 \times \text{order}} \]

where \( \text{order} \) is a state defined in the super-block or the application. The basic arithmetic operators are addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^). These operators work on integers and floating-point numbers. Currently all intermediate expressions are converted to the type of the state being computed. Hence, it is necessary to be very careful when, for example, using floating-point values...
to compute an integer state. In an integer state specification, all intermediate expressions will be converted to integers.

### 3.4.5 Complex-valued states

The syntax of complex variables is

\[(\text{real}, \text{imag})\]

where `real` and `imag` evaluate to integers or floats.

### 3.4.6 Array states

When defining the array of integers, floats, complex numbers, or strings, the syntax is a list of values separated by spaces. For example,

```
1 2 3 4 5
```

defines an integer array with five elements. The elements can be expressions if they are surrounded by parentheses:

```
1 2 PI (2*PI)
```

Repetition can be expressed using the following syntax:

```
value[n]
```

where `n` evaluates to an integer. An array or a portion of an array can be input from a file using the symbol `<` as in the following example:

```
1 2 < filename 3 4
```

Here the first two elements of the array will be 1 and 2, the next elements will be read from file `filename`, and the last two elements will be 3 and 4. In the `WaveForm` block definition, you can find an example to read a signal from a file.

### 3.4.7 String states

It is a little bit complicated to define a string state or a string array state to a super-block. This is because distinction must be made between a sequence of characters that give the name of a symbol and a sequence of characters to be interpreted literally. The syntax to use is explained in the following example:

```
This string has the word `{word}` taken from another state
```

Here `{word}` represents the value of a string state inherited from the super-block. Its use can be found in the display block for constructing labels for output plots. String arrays have a few more special restrictions. Each word (separated by white space) is a separate entry in the array. To include white space in an element of the array, use quotation marks. Thus, the following string array

```
first "the second element" third
```

has three elements. The string array

```
repeat[10]
```

has ten separate copies of the string "repeat" to make a 10 element array.

Curly braces are used to inherit the values from the super-block states. Thus, in
{paramname}

*paramname* must be the name of a string array or a string state (an integer, float or complex array is not permitted). If it is a string state, the value becomes a single element of the string array.

To use one of [ ], { }, or ) within a string, quote them with double quotes. To turn off the special meaning of a double quote, precede it with a backslash: ". Similarly, use \ to get a single backslash.

String array values may also be read from files using the < symbol. For details on how to use file references, see section *Reading parameter values from a file*. Note that for string arrays, the filename can be a literal string such as

< $PEACE/data/filename

as well as a string that refers to states such as

< $PEACE/{data_dir}/data_file

in which case the value of the state *data_dir* would be substituted. PeaCE does not perform expansion of filenames such as file.{1,2} into file1 file2 as a Unix shell might do.

### 3.4.8 Using Tcl expressions in states

Tcl is an interpreted "tool command language" designed by John Ousterhout while at UC Berkeley. Since the Tcl is embedded in PeaCE, you can use Tcl expressions in a state.

Arbitrary Tcl expressions can be embedded in a state expression by preceding the expression with character ! as in the following example:

! "expression"

First, states in the form of {state} appearing in the expression are replaced by their values. Then, the string is sent to the Ptcl interpreter for evaluation. Finally, the result is spliced into the state expression and re-parsed. The Ptcl interpreter is the same interpreter that appears as an xterm when PeaCE is started by using gohae.

This facility is general and supports both numeric and symbolic computing of expressions. Through Tcl, one can access all of its math functions, which generally behave as the ANSI C functions of the same name: abs, acos, asin, atan, atan2, ceil, cos, cosh, double, exp, floor, fmod, hypot, int, log, log10, pow, round, sin, sinh, sqrt, tan, and tanh. So, a state expression could be

! "expr sqrt(2.0 / {BitDuration})"

for the amplitude of the oscillators in a binary frequency shift keying system, in which *BitDuration* is a state. The *expr* command is a Tcl command that treats its arguments as a single mathematical expression that must evaluate to a number.

The Tcl mechanism can be used to return symbolic expressions:

! "join 2*gain1"

Because curly braces do not surround gain1, its value is not substituted before passing the expression to the Tcl interpreter. The Tcl interpreter will return 2*gain1 which is then evaluated by the state parser.

Note that white space between ! and " is permitted in numeric states, but not in string states: to get a Tcl call to be recognized in a string state you must write:

!"list /users/ptolemy/myfile"

There are several Tcl commands embedded in Ptcl that help support state calculations. They are: listApplyExpression, max, min, range, rangeApplyExpression, and sign. For example,

! "min [max 1 2 3] [sign -2]"

first evaluates to min 3 -1 and then to -1. The procedure range returns a consecutive sequence of numbers:

! "range 0 5"
returns 0 1 2 3 4 5. The \texttt{rangeApplyExpression} procedure generates a sequence of values by applying a consecutive sequence of numbers to a Tcl expression that is a function of \texttt{i}. For example, you can generate the taps of an FIR filter that is a sampled sinusoid by using

\begin{verbatim}
! "rangeApplyExpression { cos(2*{PI}*$i/5) } 0 4"
\end{verbatim}

generates one period of sinusoidal function and returns

1.0 0.309042 -0.808986 -0.809064 0.308916

The \texttt{listApplyExpression} is similar to \texttt{rangeApplyExpression} except that it only takes two arguments: the second argument is a list of numbers to substitute for \texttt{i} in the expression. The command

\begin{verbatim}
! "listApplyExpression { cos(2*{PI}*$i/5) } [range 0 4]"
\end{verbatim}

is equivalent to the previous example of the \texttt{rangeApplyExpression} function.

If you are running Tycho TclShell from within \texttt{pigi} or \texttt{pigi -console}, you can receive help on the new Tcl procedures \texttt{listApplyExpression}, \texttt{max}, \texttt{min}, \texttt{range}, \texttt{rangeApplyExpression}, and \texttt{sign}, by typing \texttt{help sign} at the prompt.

### 3.4.9 Inserting comments in a state

Comments are also supported for non-string states. A comment is specified with the \# symbol. Everything after the \# until the end of the line is discarded when the state is evaluated. Comments are especially useful in combination with files as they can help remind the user of which super-block or block state the file was written.

For example, a comment could be added to the \texttt{frequency} state:

\begin{verbatim}
freq # This is set to the Galaxy state
\end{verbatim}

Comments are not supported for the String state or String Array state types.

### 3.4.10 Building a combo-box for a state

Sometimes you want to select one item among the item list. In this case, you can build a combo box by specifying special characters “##” in the state field. For example, suppose that the “\texttt{step}” has four values such as \texttt{PI/100}, \texttt{PI/50}, \texttt{PI/10} and \texttt{PI/2}. Fill “##PI/100,PI/50,PI/10,PI/2” in the step field (Figure 3-16).

![Figure 3-16 Ramp block step state](image)

After pressing Enter key, you can choose an item from the specified items (Figure 3-17).
To eliminate the combo box, type “##” and press Enter key (Figure 3-18).

After then, you can edit the step field (Figure 3-19).

### 3.4.11 Reading state values from a file

The values of most state types can be read from a file. This syntax for this is to use the symbol `<` as in the following example:

```plaintext
< filename
```

First, any state appearing in the `filename` in the form of `{state}` are replaced with their values. Then, any references to environment variables or home directories are substituted to generate a complete path name. Finally, the contents of the file are then read and spliced into the state expression and re-parsed. File inputs can be very useful for array states that may require a large amount of data. Other expression may come before or after the `<filename` syntax (any white space that appears after the `<` character is ignored).

### 3.4.12 Using delays
In several domains, delays can be placed on arcs. A delay is not a block, but rather is a property of the arc connecting two blocks. The interpretation of the delay in the dataflow domains (code generation domains with SPDF semantics) is as an initial sample on the arc. An initial sample for the scalar data types is one whose value is zero. When the arc passes samples containing "message" type data, a delay on the arc will create an "empty" or "garbage" message.

To use these delays in PeaCE, select the wire connecting two instances and edit the state. You can specify the number of delays (Figure 3-20).

![Figure 3-20 Using delays](image)

Then you can see the delay is annotated on the arc where the number represents the number of initial samples.

In order to specify the initial values to those initial samples, you should embrace the values of delays with braces. The number of values becomes the number of initial samples that will be added to the buffer of the arc corresponding to the delay. These values are recognized as a string and are parsed according to the data type associated to the arc. For example, an initializable delay with state "{1 0 1}" on an arc passing float values will have a buffer with three initial float samples. The three samples will have the values 1.0, 0.0, and 1.0 respectively. If the arc works on complex samples instead, an error would be given since complex numbers must be specified using a pair of numbers. A proper argument list for the delay in that case would be "{(1,0) (0,0) (1,1)}". The shorthand for declaring multiple values is valid, just as in the arraystate case. For example, an argument list of "{2 [5]}" would specify five initial samples with value 2.

### 3.5 Making your own block libraries

#### 3.5.1 Creating blocks

How to create a block depends on the model of computation. To aid the creation of a new block, PeaCE provides a tool, called Star Edit, as explained in Chapter 18. See Chapter 18 for detailed information on creating blocks.
3.5.2 Importing a new block

When you do not have registered the block as an icon after writing a block code, you had better import a block. **Import Block** (or **Import Star**) from **Tool** menu stores the directory name, and compiles and links the block code dynamically to PeaCE.

To register a block:

1. Select **Import Star** from the **Tool** menu. An **Import Star** Icon dialog is displayed.
2. Select a domain from the domain name drop-down list and type the block name in **star name** field.
3. Click **OK**.

For example, when ‘CGCRamp.pl’ is registered, ‘domain name’ is ‘CGC’, ‘star name’ is ‘Ramp’, and directory name is ‘$PEACE/src/domains/cgc/stars’. When you write a CGCYourStar.pl in $HOME/work in the server machine, domain name is ‘CGC’, star name is ‘YourStar’, and directory name is ‘$HOME/work’.

![Figure 3-21 Make a Star Icon dialog including directory name](image)

After pressing OK button, you can see a dialog window asking where to put the block(star) icon as shown in Figure 3-22. Using the Save Dialog, you should place the Ramp.ilink to an appropriate directory. Afterwards you can see a new star(block) icon after pressing “Save” in the saved directory.

![Figure 3-22 Star Icon Save Dialog](image)

(a) Initial directory (b) Directory navigation

Since the Ramp is already registered, warning message will be popped up. If you ignore it, you will have a new icon for the Ramp block.
3.5.3 Registering a new block

The **Register Star** menu is equivalent to **Import Star** except that you do not need to specify the directory name. Therefore **Register Star** is used after **Import Star** is once performed. Once the directory name is stored, you use **Register Star** to change the star icon after modifying the block code. Suppose that you created a new block and made an icon. Later you add a new port to the block, then you perform **Register Star** to update the icon with additional port information.
3.5.4 Link Star
When you do not change port name or port number, use Link Star from Tool menu to compile and link the star dynamically after you have changed the star code. You need not update the block icon if port information is not changed. To use Link Star, you should select the block that will be recompiled and re-linked dynamically. If you do not specify a block, you will compile and link all the blocks in the design schematic, which may not be what you want. Be cautious!

![Link Star](image)

Figure 3-26 Link Star (a) menu selection (b) compiling and linking the Ramp block

3.5.5 Creating a super-block
When a subgraph of blocks is used frequently, it is useful to make the subgraph as a super-block. To begin with, open a new schematic for a super-block. Put necessary block icons and connect them. In order to make this schematic communicate with other blocks, input and/or output ports must be added.

For example, create a cosTest design schematic in “./schematic/User” directory and build a design as shown in
3.5.6 Adding super-block ports

Open the library browser and see inside the Port directory. You can find the icons for input port and output port. Put the ports on the appropriate location in the super-block schematic and connect them. Select a port and check the port property in the design tab. You can specify port name and port type and whether this port is a multi port or not. If there are more than one port at input or output, you must confirm that the port names are all different.
3.5.7 Adding super-block states

When you make a super-block, it’s useful to create some states of the super-block. To do so, click Add Parameter in the Design browser and specify the state variable (Figure 3-29). Remember that you must click the schematic to specify the schematic property. You also have to specify its type correctly.

3.5.8 Registering super-blocks

After all editing is done for a super-block, you must register the schematic as a super-block in order to use it with a single icon.

To register a super-block:

1. Select Register Galaxy from the Tool menu. A Save dialog is displayed (Figure 3-30).
2. Navigate to the directory you want to save the super-block and type locations for icon, image, ilink files.
3. Click **Save**. You can find the super-block icon just registered in the directory you specified in the library browser.

### 3.5.9 Create an Icon with Multi-port

When you register a block (or star) with a multi port such as Add, Mpy, Fork, etc, you want to create an icon with the specified number of ports.

Figure 3-31 (a) shows the result of registering “Add” block. If you set the number of input port to two, then you should make new icons and image and ilink file names in order to distinguish two input port Add icon from multi port Add icon as shown in Figure 3-31 (b).

![Figure 3-31](image)

**Figure 3-31** Make an icon for multiport block (a) setting the number of ports, (b) when the number of input ports is 2 in case of Add block

### 3.5.10 Editing library browser
The library directory includes many icon link files denoted by *.ilink. If you make a new directory and copy some icon link files then the modifications are shown in the library browser. Moreover the library browser can read a schematic. Therefore you can rebuild a library path in your style.

3.6 Editing and viewing schematics

3.6.1 Placing blocks

To place a block in a schematic:
1. Click a block icon in the library browser to select it. Do not drag the mouse.
2. Move the mouse pointer in the schematic window. See the block icon following the mouse pointer.
3. Click to place the block where you want to locate in the schematic window.

3.6.2 Connecting blocks

To connect two blocks:
1. Move the mouse pointer to the output port of one block. You can see a green circle and the port name.
2. Click the circle to start a line from the port.
3. Move the mouse pointer to draw a line as you wish. If you want to make a turn, click at the point where you want to make a turn.
4. Click the circle of the input port of the other block to complete the line.

3.6.3 Selecting blocks and lines

Selection of icons and lines is basic operation to change their properties.
You can draw a box by pressing and dragging a mouse point. Every icons and lines in the box will be selected after releasing the mouse point. The selected objects are displayed with enclosed by green boxes.
The other selection method is clicking an object. Just click on the object that you want to select. Be careful not to move a mouse point between pressing and releasing a mouse button since the movement results in drawing a box.
When you want to select an object one by one, click a mouse with pressing a control button on the keyboard.

3.6.4 Moving blocks and lines

To move blocks and lines:
1. Select blocks as explained in Selection blocks and lines.
2. Click and drag the selected objects to locate where you want to put.
3.6.5 Cutting/Copying/Pasting/Deleting blocks

**Cutting blocks**

To cut a block (or blocks):
1. Select blocks as explained in *Selection blocks and lines*.
2. Select **Cut** from the **Edit** menu, or press **Ctrl+x**.

**Copying blocks**

To copy a block (or blocks):
1. Select blocks as explained in *Selection blocks and lines*.
2. Select **Copy** from the **Edit** menu, or press **Ctrl+c**.

**Pasting blocks**

To paste the cut or copied blocks:
1. Select **Paste** from the **Edit** menu, or press **Ctrl+v**. The pasted blocks are displayed at the same position where the original lies.
2. Click and drag mouse to locate the pasted block where you want to.

**Deleting blocks**

To delete a block (or blocks):
1. Select blocks as explained in *Selection blocks and lines*.
2. Select **Delete** from the **Edit** menu, or press **Delete** key.

3.6.6 Rotating blocks

To rotate a block (or blocks):
1. Select blocks as explained in *Selection blocks and lines*.
2. Select **Rotate** from the **View** menu, or press **Ctrl+t**. The selected blocks rotate 90° counter-clockwise. Connected lines change accordingly while preserving connections.

3.6.7 Editing block icons

You can edit an icon image of a block or a super-block. The default icon image is a rectangle with related gif file, if any. Load an icon image first by clicking the icon in the library browser and moving it to a schematic, then click **Edit Icon** in **Edit** menu. You can use any functionality provided in the **Comment** menu. To help you to
draw ports easily, InArrow and OutArrow are presented. For multi ports, use InMultiArrow and OutMultiArrow. In case you need some object appear in front of other objects, select the object and click MoveToFront. You can open several icon images simultaneously and select multiple objects simultaneously. Thus, if you need to draw an icon image that is little different from an existing one, open both icon images and select and copy the whole objects in existing one to the new icon image. The useful function that is not explicitly presented is to resize an existing object. First, select an object to resize. By pressing shift key and moving a mouse back and forth, you can resize it.

### 3.6.8 Editing comments

You can put some comments on a schematic (Figure 3-32).

![Figure 3-32 Comments in a schematic](image)

Click Comment in the toolbar. Click Text in order to put some description about the selected schematic. You can choose the font size. You can also choose the color of the font by clicking Color in the menu. If you have already written a text, you should select a text first and change the color in Color menu. To draw a straight line, click Line in the menu and click a point where a line starts and drag the mouse pointer to the point where a line ends. You can draw Rectangles and Ovals and Polygons if you need. These objects will be filled if you checked Fill option in the menu. When you draw an object using Polygon, the last point and the first point in the object is always automatically connected forming a polygon. To finish drawing a polygon, click the right button. In case you need to insert a gif file to a schematic, click Gif. Now, click a point where the left top of the gif file will locate. Drag the mouse pointer, and the gif file will appear.

### 3.6.9 Zooming a schematic

To zoom in a schematic, select Zoom In from the View menu or press Alt+z.
To zoom out a schematic, select **Zoom Out** from the **View** menu or press **Ctrl+z**.
To make your design fit in the schematic, select **Zoom Auto** from the **View** menu or press **Ctrl+f**.
In some platform, zooming in or out may put the screen out of shape a little. Sorry for any inconvenience you might have.

### 3.6.10 Looking inside of a block

To look inside a block (or super-block):

1. Click a block (or super-block) to select it.
2. Select **Look Inside** from the **View** menu or press **Ctrl+i**. If it is a block, PeaCE displays a block description file (.pl). If it is a super-block, PeaCE displays a schematic of the super-block.

Just double-clicking works as the same way as looking inside.

### 3.6.11 Viewing Names of blocks

To view names of all blocks and ports, check **Name** from the **View** menu.

### 3.6.12 Viewing Rates of blocks

When you enable **Rate** from the **View** menu, you can see the sample rates after running the schematic once.

### 3.7 Directory structure

#### 3.7.1 Data structures

- **.schematic**: This directory contains the design schematic files. A schematic file contains the domain and the target names, target parameters, run count, block location and states and connectivity as well as comments.
- **.icon**: The files in this directory contain general information about a block or the corresponding schematic when it points to a super-block. Moreover the file includes port information and the location of its image file.
- **.iconImage**: The files in this directory describe what the image looks like. It contains a gif file textual information such as its location and size, and shape information.
- **.library/*.link**: The files in “.library” directory just contain the location of the icon files. Therefore you can change the block location in the library directory by copying and moving the file.
- **.param**: The files in this directory are created when super-blocks are registered. A file in this director contains the corresponding super-block parameter information.

Since the icon and image files for various schematics are saved in the same directory, you have to be careful not to overwrite the existent icon and/or image files.
3.7.2 Hae root directory

- **bin**: Binary executable files
- **classes**: Java classes
- **com**: External tools
- **config**: Configuration files
- **doc**: Documentation files
- **gif**: Gif files used by PeaCE and block icons
- **hae**: HAE kernel
- **icon**: Icon files
- **iconImage**: Icon image description files
- **Jars**: Java archive files
- **Library**: Block libraries
- **menuIcon**: Menu icon gif files
- **Param**: State files
- **schematic**: Schematic files

3.7.3 Schematic directory

<table>
<thead>
<tr>
<th>Peace</th>
<th>Demo</th>
<th>BP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Cgc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Communication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H.263</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H.264</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Image</td>
</tr>
<tr>
<td></td>
<td></td>
<td>matlab</td>
</tr>
<tr>
<td></td>
<td></td>
<td>matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MP3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multirate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Signal Processing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Spdf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DE-BP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Continuous Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Miscellaneous</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Networking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Queue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TclTk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wormhole</td>
</tr>
<tr>
<td>Fsm</td>
<td>Reflex Game</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Simple Counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Traffic Signal</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Highlights</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VHDL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Galaxy</td>
<td>Bp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cgc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>De</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FSM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sdf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VHDL</td>
<td></td>
</tr>
</tbody>
</table>

User

### 3.7.4 Library directory

<table>
<thead>
<tr>
<th>SPDF</th>
<th>Src</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sink</td>
<td>Control</td>
</tr>
<tr>
<td>CPP</td>
<td>mmt</td>
</tr>
<tr>
<td>TclTk</td>
<td>Math</td>
</tr>
<tr>
<td>Processing</td>
<td>Communications</td>
</tr>
<tr>
<td></td>
<td>Matrix</td>
</tr>
<tr>
<td></td>
<td>Image and Video</td>
</tr>
<tr>
<td></td>
<td>Signal Processing</td>
</tr>
<tr>
<td></td>
<td>Spectral Analysis</td>
</tr>
<tr>
<td>Matlab</td>
<td></td>
</tr>
<tr>
<td>BP-DE</td>
<td>Src</td>
</tr>
<tr>
<td>Sink</td>
<td>Control</td>
</tr>
<tr>
<td>Math</td>
<td></td>
</tr>
<tr>
<td>Processing</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td></td>
<td>Networking</td>
</tr>
<tr>
<td></td>
<td>Queue</td>
</tr>
<tr>
<td></td>
<td>Timing</td>
</tr>
<tr>
<td></td>
<td>Continuous Time</td>
</tr>
</tbody>
</table>

FSM
Port
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H.263</td>
</tr>
<tr>
<td></td>
<td>H.264</td>
</tr>
<tr>
<td></td>
<td>Image Tracking</td>
</tr>
<tr>
<td></td>
<td>Jpeg</td>
</tr>
<tr>
<td></td>
<td>MP3</td>
</tr>
<tr>
<td></td>
<td>MP3-Old</td>
</tr>
<tr>
<td>Galaxy</td>
<td>CGC</td>
</tr>
<tr>
<td></td>
<td>DE</td>
</tr>
<tr>
<td></td>
<td>FSM</td>
</tr>
<tr>
<td></td>
<td>VHDL</td>
</tr>
<tr>
<td>User</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4. Introduction to the Models of Computation

Author: Soonhoi Ha

Block diagram specification enables us to reuse pre-defined blocks and to understand the overall structure of the system behavior in a hierarchical fashion. But, there are numerous ways of interpreting the block diagram unless we do not specify the execution rule. For example, see Figure 4-1 and guess what the meaning of this block diagram is.

![Block Diagram](image)

Figure 4-1 A simple block diagram

One possible interpretation is that block C is executed as soon as it gets an input sample either from block A or block B. Another is that block C becomes executable only after it gets input samples from both block A and B. When block A is executed twice to produce two output samples to block C, either two samples are queued on the arc or the later sample overwrites the former sample.

Therefore, it is necessary to assign a specific execution rule to a block diagram for unambiguous interpretation by defining a “formal” model of computation. Therefore using formal models of computation is recently advocated because there are several benefits: (1) Static analysis of the specification allows one to verify some correctness measures of the functional specification. (2) The specification model can be refined to an implementation to ease the system validation by the principle of "correct-by-construction". (3) A formal specification model is not biased to any specific implementation method, so allows one to explore the wider design space. (4) It represents the system behavior unambiguously so that collaboration and design maintenance can be accomplished easily. In this chapter, we overview four basic models of computation used in PeaCE while each model will be explained in the following chapters in great details.

In PeaCE, a block diagram is assigned its domain and target. The domain of a block diagram implies the model of computation and its purpose. For example the CGC domain in PeaCE implies that the block diagram follows the SPDF model and its purpose is to generate C codes from the graph. The target of a block diagram indicates the target architecture or environment we want to run the graph. A domain usually has various targets associated. For example, we can generate different C codes from the same graph depending on whether the target architecture is the host machine for simulation, a prototype hardware with a single processor, a multiple processor SoC, and so on.
4.1 Dataflow Model: SPDF (Synchronous Piggybacked Dataflow Model)

In a dataflow model, a block is executable only if there exists a specified number of input samples on each input arc. The number of samples produced (or consumed) per block execution is called the **output (or the input) sample rate** of the block. In case all sample rates are statically determined and fixed at runtime, and can be any integer, the block diagram is called a **synchronous dataflow** (SDF) graph. By default, the sample rate of an input or output port is unity. And, there is assumed an unbounded FIFO queue on each arc.

Based on this model, examine the execution scenario of the block diagram in Figure 4-1. Block C is executable only after block A and block B are executed to produce at least one sample on each input arc of block C. It means that a block execution sequence of \{ABC\} is valid while \{ACB\} is not. When block A is executed twice in sequence, two samples are accumulated on arc AC.

Dataflow graph has become a successful representation for DSP algorithms since dataflow semantics is well matched with algorithmic function flow in DSP applications. You may think that an input port of a block represents an input argument for a function that corresponds to the body of the block. A function can be called only after all input arguments are provided. An output port corresponds to a return value of the function. Then, a block diagram with dataflow model looks like a function call graph. **In fact, a dataflow program graph is used to specify a digital signal processing (DSP) task such as H.263 encoder, MP3 decoder, and so on.**

Even though the SDF model and its related models are widely adopted for algorithm specification in numerous DSP design environments, it has a couple of serious limitations to represent multimedia applications: First, the SDF model can not express conditional execution of a block. Second, the model can not use the shared data structure between blocks and pointer operations so that the efficiency of the synthesized code is noticeably worse than the hand-optimized code in terms of memory requirements. Therefore, we make two extensions to the SDF model to overcome these problems, which are fractional rate dataflow (FRDF) and synchronous piggybacked dataflow (SPDF), which will be discussed in the next chapter. (@ SPDF)

In PeaCE, we distinguish two different domains associated with the SPDF model: CGC and VHDL. These domains generate a C code and a VHDL code from a SPDF graph respectively.

4.2 DE Model: DE (Discrete Event Model)

In a Discrete Event (DE) model, a sample is associated with a **time stamp** indicating when it is generated. And a block processes input events in the order of arrival times. In Figure 4-1, block C is executed as soon as it receives an input event from any input port. In the DE model there is a run-time scheduler that collects and sorts all events in a queue, and delivers the earliest event to the destination block. The scheduler manages the global clock. When a block generates an event with a smaller time stamp than the global clock, the scheduler signals a **causality error** and aborts the execution of the block diagram.

Unlike the SDF model, there is no guarantee that an output event is produced from an output port after the block is executed. In Figure 4-1, block A may never produce any event to block C though it is executed multiple times. Therefore, the execution sequence of blocks cannot be determined a priori before run time.
The DE model is widely used for dynamic system behavior as a function of time: for example, queuing systems, communication networks, and hardware simulations. The DE model in PeaCE is inherited from the Ptolemy classic. While the DE model is provided in PeaCE, note that it is not integrated in the HW/SW codesign flow.

Even though a block in the DE model is event-driven, or triggered by an event arrival, it can be modeled as time-driven if we attach a self-loop to the block. Through the self loop, the block may trigger itself at the predetermined future time. See chapter 7 for more explanation.

### 4.3 FSM Model: fFSM (flexible Finite State Machine Model)

An FSM graph is a directed connected graph composed of states and transition arcs. A state describes a specific status of the system and a transition arc is associated with a condition that must be fulfilled to make state transition from a source state to a destination state. Finite state machine (FSM) is the most popular model to describe the control module of a system. Even though FSM is simple to use, its unstructuredness and state explosion problem due to system concurrency and memory prohibit FSM model from practical representation of complex control behavior. So many extensions have been proposed to overcome these problems.

To describe complex control modules, the extensions should support various kinds of concurrency. Another desired feature is compositionality whether the complex module can be represented as a composition of simpler modules. Then, modules can be easily reused to construct a large system. Moreover, because of their complexity, it is desired to have a static analysis method to check ambiguous behavior. Finally for fast prototyping, automatic hardware (or software) synthesis is needed from the extended model. To meet these requirements, PeaCE devises a new extension of FSM model, called fFSM Model. It extends the expression capabilities by concurrency, hierarchy, and state variable. See chapter 8 for detailed explanation.

### 4.4 Task Model: BP(Backplane)

The SPDF model and fFSM model is used to describe the internal behavior of a computation and a control task respectively. We use another model, called Task Model, to describe the system behavior at the task level. Thus we are able to represent multi-tasking applications. In PeaCE, we can define a super block that encapsulates a block diagram with a different model of computation to make a hierarchical block diagram. At the top-level, the Task model represents a composition of multiple tasks: each task is modeled as a SPDF model or fFSM model. Since we use the Task model in this fashion, we call it a BP (Backplane) model.

The Backplane model defines additional features for inter-task interaction and task management: when a task is triggered to execution or suspended, and how tasks can communicate with each other, and so on. See chapter 9 for more explanation.
Chapter 5. SPDF Model: C code generation

Author: Soonhoi Ha and Hyunok Oh

SPDF model is used to specify DSP algorithms or computation intensive tasks re-using the predefined blocks. Since it is extended from SDF (Synchronous Data Flow) model, we start our explanation with the SDF model. In this chapter, we will show how to draw an SPDF graph and how to define a new SPDF block. In PeaCE, we call an atomic block a “block (or star)” and a hierarchical block a “super block (or galaxy)”; you may easily catch the analogy. We will see that software code or RTL-level hardware will be automatically synthesized from the SPDF block diagram.

5.1 Draw an SDF Graph Using Pre-defined Library Blocks.

Let us draw a simple SDF task that adds a square-wave signal and a random noise and displays the output. Figure 5-1 displays what we are now going to draw and what we obtain from the graph.

Figure 5-1 HAE screen snap-shot of a simple SDF example
The procedure you have to perform in HAE is following:

1. Open a new facet and name it as you like ("File"-"New"). And set the domain to "CGC" and the target to "default-CGC". "Domain" defines the model of computation. The CGC domain indicates that the SPDF model is used and a C code will be generated from the block diagram. In reality, CGC is an acronym of "Code-Generation domain in C" borrowed from the Ptolemy project. And, "Target" specifies the target environment in which the generated C code is run. If "default-CGC" target is chosen, the generated code is run in the host machine. We will omit more detailed explanation on Target at this moment.

2. Bring two signal sources from the block library (library/SPDF/Src): one is a Waveform block and the other is an IIDUniform block. The color of the port indicates the type of the sample. We select a block with “blue” port to use float-type samples. We set the “value” parameter of the Waveform block to be “1[20] 0[20]” meaning that one period of the waveform consists of 20 samples of value “1” and 20 samples of value “0”. By default, the Waveform block generates a periodic signal (note that the “period” parameter is set to 0). The IIDUniform block generates a random signal ranging between two parameter values: we set the “lower” value to “-0.1” and the “upper” value to “0.1”. Please refer to the document for the library blocks to understand the block function.

3. Now, we bring a two-input add block from (library/SPDF/Math). Two input ports of the adder block are connected to the output port of two signal sources.

4. For output display, we bring a display block called “Xgraph” from (library/SPDF/sink) and connect the output of the adder block to its input port. For now, we use the default parameter values for the Xgraph block.

5. Now, "run" the application by clicking the “Run” menu and setting the “When to stop” value to 200. Then, we get two types of output results: one is an automatically generated C code and the other is the display output showing that a random noise is overlaid to a square-wave. If you examine the generated C code, the sequence of block execution is {Waveform, IIDUniform, Add, and Xgraph}. We will discuss how the C code is generated from the SDF graph later.

Now, we believe that you understand how to draw and execute an SDF graph using pre-defined blocks. It is fairly straightforward except that it is your responsibility to find out suitable blocks from the block library. If there is no suitable block for your purpose, you have to define your own. This is the topic of the next section.

5.2 SDF Block Definition

Let’s make “AveDown” block which will output the average value of as many input samples as block parameter “count” is given. And, insert this block between the Add block and the Xgraph block in Figure 5-1. Since there is no “AveDown” block in the block library, it has to be created.

To begin with, double click the “Add” block in the block diagram to figure out how a block is defined inside PeaCE. Then, you can see a popped-up file named “CGCAdd.pl” in the screen as Figure 5-2 shows.
A block is defined in a file whose name is a concatenation of the domain name, block name, and “.pl” suffix: “CGC” is the domain name, “Add” is the block name in “CGCAdd.pl”. As can be seen in Figure 5-2, the body of a block consists of sections: each section is defined as the section directive and the section body enclosed by curly braces: “name{}” and “domain{}” are two examples. The sections are placed in any order. In Table 5-1, we show the list of sections with brief summary of how they are used. In the description field, (R) indicates that this section should exist in any block definition. Not all sections are needed for block definition. Only a few sections are usually used for block definition. It means that you do not need to understand all sections of Table 5-1 at this moment. Just look over it. A separate section is devoted to explain the preprocessor language in more details.

Since the PeaCE kernel is written in C++, a block is defined inside the server as a C++ class. A “.pl” block definition file is pre-processed by “ptlang” program in the server to create a pair of CGCAdd.{h,cc} files. Therefore, some sections in Table 5-1 need some C++ programming skill. But, never mind that you do not know C++ much. Only handful of C++ knowledge is needed at best, and we will let you know what it is.

Table 5-1 Sections in the block def

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>(R) Block name</td>
<td>name {AveDown}</td>
</tr>
<tr>
<td>domain</td>
<td>(R) Domain name</td>
<td>domain { CGC }</td>
</tr>
<tr>
<td>derivedFrom</td>
<td>Specify the base block class</td>
<td>derivedFrom {Xgraph}</td>
</tr>
<tr>
<td>desc or</td>
<td>Short summary of block definition</td>
<td>desc { This block outputs the average value of as many input samples as block parameter “count” is given. }</td>
</tr>
<tr>
<td>descriptor</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>version</strong></td>
<td>Version number and date</td>
<td>version { 2003. 1. 8, version 0.1 }</td>
</tr>
<tr>
<td><strong>author</strong></td>
<td>Author of this block</td>
<td>author { Soonhoi Ha }</td>
</tr>
<tr>
<td><strong>copyright</strong></td>
<td>Copyright information</td>
<td>copyright { Copyright (C) Seoul National University, the CAP Lab. }</td>
</tr>
<tr>
<td><strong>location</strong></td>
<td>Where to put this block</td>
<td>location { working directory }</td>
</tr>
<tr>
<td><strong>htmldoc</strong></td>
<td>Full documentation</td>
<td>htmldoc { }</td>
</tr>
<tr>
<td><strong>explanation</strong></td>
<td>Full documentation</td>
<td>Omitted</td>
</tr>
<tr>
<td><strong>hinclude, ccinclude</strong></td>
<td>Specify other header files to be included in “.h” file or “.cc” file</td>
<td>hinclude { “Message.h” } ccinclude {&lt;math.h&gt;, “Error.h”}</td>
</tr>
<tr>
<td><strong>input, output</strong></td>
<td>Define input (output) port. It has subsections of “name”, “type”, and “desc” (optional). “type” can be int, float, fix, message, and anytype.</td>
<td>input { name {input} type {float} } , output { name {output} type {float} }</td>
</tr>
<tr>
<td><strong>inmulti, outmulti</strong></td>
<td>Define a set of input ports or output ports. It also has the same subsections as input/output.</td>
<td>inmulti { name {in} type {float} } , outmulti { name {out} type {float} }</td>
</tr>
<tr>
<td><strong>state or defstate</strong></td>
<td>Define a state or a parameter. It has five subsections: “name”, “type”, “default”, “desc” and “attributes”</td>
<td>state { name {gain} type {float} default {1.0} desc {output gain} attributes {A_SETTABLE</td>
</tr>
<tr>
<td><strong>private, protected, public</strong></td>
<td>Member variables to be included in the block class header file with the specified access privilege</td>
<td>private { int index; }, protected { double foo; }, public { }</td>
</tr>
<tr>
<td><strong>code</strong></td>
<td>Code segments to be included after the include files in the block class “.cc” file</td>
<td>code { #define XY 0 }</td>
</tr>
<tr>
<td><strong>constructor</strong></td>
<td>C++ code to be included in the block class</td>
<td>constructor { noInternalState(); }</td>
</tr>
<tr>
<td><strong>setup</strong></td>
<td>C++ code to be executed at start time before compile-time scheduling. It may be called several times.</td>
<td>setup { input.setSDFParams(2,1); }</td>
</tr>
<tr>
<td><strong>begin</strong></td>
<td>C++ code to be executed after compile-time scheduling but before run-time.</td>
<td>begin { }</td>
</tr>
<tr>
<td><strong>initCode</strong></td>
<td>C++ code to be executed after compile time scheduling but before run-time. Only for code generation purpose.</td>
<td>initCode { }</td>
</tr>
<tr>
<td><strong>go</strong></td>
<td>C++ code to be executed at run-time. In the CGC domain, it emits the block function code.</td>
<td>go { addCode(main); }</td>
</tr>
<tr>
<td><strong>wrapup</strong></td>
<td>C++ code to be executed at the end of the block diagram execution</td>
<td>wrapup { }</td>
</tr>
<tr>
<td><strong>codeblock</strong></td>
<td>C code to be emitted by addCode() method or similar methods.</td>
<td>codeblock(main) { $\text{ref(output)} = \text{ref(input,0)} + \text{ref(input,1)}$ }</td>
</tr>
<tr>
<td><strong>method</strong></td>
<td>Specify an additional method to the block class. It has subsections such as “name”, “access”, “arglist”, “type” and “code”.</td>
<td>method { name {exec} access {protected} arglist {“(int index)”}</td>
</tr>
</tbody>
</table>
In the server, the pre-defined block file is stored in the following directory: `$PEACE/src/domains/{domain_name = CGC}/stars`. Since it is not a good idea to add a block to the official block library before it is fully tested, you have to put a new block definition into your private working directory in the server. So, make sure that you are assigned a private working directory there.

Now, we will explain how to define a block and use the block in the following subsections. There can be many different implementations for an SDF block: we may generate a C-code, a VHDL code, or a systemC code. We consider C-code generation in this chapter.

### 5.2.1 Inline-style Code Generation

If the block body is simple, the popular way of defining a block is to generate inline-style code for the block. We will show how to define this kind of block. The AveDown block belongs to this kind since its functionality is pretty simple.

The easiest way of creating a new block is to copy a library block definition into your working directory with a new name and change the body of the file. Since the “AveDown” block has one input and one output port, find out a simple library block with the same number of ports: Gain block is an example. Figure 5-3(a) shows the definition of the Gain block. Then, copy the block into your working directory “~your_name/work”:

```
cp $PEACE/src/domains/cgc/stars/CGCGain.pl ~your_name/work/CGCAveDown.pl
```
Figure 5-3 Block creation by “copy-and-modify” approach: (a) CGCGain and (b) CGCAveDown block

Then, you have to modify the body of the file as shown in Figure 5-3(b). What sections are changed?

(1) name: Sure, you have to change the name of the block.

(2) author, location, desc, version: These are optional.

(3) input, output: You may change the names.

(4) state or defstate: change the name to “count” or whatever name you want to use. This state will indicate how many samples to be consumed and averaged per block execution.

(5) constructor: No change. “noInternalState()” method indicates that this block has no state variable update at run-time. The “count” state is not a state in a strict sense. It is a parameter of the block. A parameter is updated at compile-time and unchanged at run-time.

(6) setup: You have to add “setup” section to set the input sample rate as the “count” value by calling “setSDFParams()” method. This first argument is the sample rate and the second argument indicates the maximum array index the block uses for the port buffer. The port buffer index starts with 0 which stores the newest token. Since the “count” state is defined as a C++ class (IntState class inside the PeaCE kernel) we have to use a type-cast operator, say int(count), to get the integer value of the state. If the method is not called, the port has a unity sample rate: the CGCGain block does not call this method since the sample rates are all unity. If a block has a port with non-unity sample rate, we have to define the sample rate by calling “setSDFParams()” method.

(7) codeblock: Modify the codeblock to perform the desired function: take the average of input sample values. You have to use “$ref” macro to access a port. “$ref” macro is preprocessed to assign a unique name in the
generated code. For a state you may use $ref or $val macro. If $ref macro is used, a separate variable is defined for the state in the generated code. If $val macro is used, the value of the state is used in the generated code without declaring any variable. In this example, you need not assign a variable for the “count” state since the value will not change at run time. Therefore, use “$val” macro to access the “count” state.

(8) go: This section defines the main function body of the block. It usually contains addCode() methods to generate the function body at the scheduled position in the generated code.

After you define a new block, insert the block into PeaCE by “importing” the block by clicking “Tool – Import Star” in HAE. Then, the block is compiled and dynamically linked to the PeaCE kernel and a new icon is created in HAE. See 5.2.5 How To Use A New Block in HAE on page 68 for more information. Figure 5-4 shows how the newly created block is inserted into the block diagram. On the right side, the generated code from the AveDown block can be seen in the scheduled position. $val(count) value is preprocessed to value 2. $ref(output) is named “output_3” while $ref(input, i) is named “output_2[]”.

Figure 5-4 Screen snap-shot after inserting the newly defined AveDown block

5.2.2 Code Generation with Procedures

If the body of a block is not simple, it is a good practice to use procedures inside the block definition. It may happen that there are multiple instances of the same block in a block diagram. Or the same block body may
Appear multiple times in the generated code if a schedule of the block diagram is determined so. Then it is a waste if we have multiple copies of the same block body. If we define the block body as a procedure inside the block, we can avoid such a waste because the function is defined only once in the generated code.

Now, we define another new block “Clamp” that clamps the input value to a given magnitude threshold if it is greater than the threshold. We will append this block after the IIDUniform block in Figure 5-1. Figure 5-5 shows a preprocessor file for the block, in which the clamping function is implemented with a procedure, “noise_clamp”.

![Figure 5-5 CGC Clamp block definition using a procedure](image)

In Figure 5-5, the “noise_clamp” procedure is defined in a codeblock and this codeblock is emitted by “addProcedure” method in the “initCode” section. The “initCode” section is used to generate code before the main schedule loop of the block diagram. The addProcedure method is usually called in the initCode section. The first argument of the addProcedure method is the name of the codeblock while the second argument is the name of the procedure. If more than one block define the same procedure multiple times, only one definition is emitted in the generated code. Note that the “main” body of the block is just calling the procedure.

Sometimes, we want to reuse a procedure defined in a separate file. This is the topic of the next section.

### 5.2.3 Importing A Pre-defined Procedure

Suppose that we have a file called “foo.c” that contains a procedure developed elsewhere but needed in the current block. Or, we may want to make a new block to perform the procedure. How can we reuse the procedure? There are a couple of answers:

1. Make a new block and copy the procedure into a new codeblock section as shown in the last section.
(2) Make an “extern” declaration of the procedure and use the procedure in the block. Later, we should link foo.o together with the generated code. Figure 5-6 shows how to make an “extern” declaration by “addDeclaration” method. If there is a header file for the procedure, we have to insert the header file by calling “addInclude(“foo.h”)” in the initCode section instead of using addDeclaration method.

```c
Name { ImportFunc }
Domain {CGC }
Input { } output { }
Codeblock (import_func) {
    extern double importfunc(double);
}
Codeblock(main) {
    $ref(output) = importfunc($ref(input));
}
initCode{ addDeclaration ( import_func, "import_func");}
Go { addCode(main);}
```

Figure 5-6 A block that imports a predefined procedure from “foo.c”

### 5.2.4 Code Generation with Multi-Ports

When we define a block, sometimes we want to leave the number of ports unspecified in the block definition. Let us consider an Add block. Do you want to have multiple Add blocks for each number of input ports? Surely not! In this case, we have to use a “Multi-port” as the input port of the Add block. A multi-port is not an actual port but a list of ports. If a new connection is made to the multi-port, a new port is created for the connection and enlisted. Therefore, the number of connections becomes the number of actual ports, which is determined after a dataflow graph is completely drawn. Even though we explain the case of input multi-ports, a similar explanation is hold for output multi-ports.

If the name of a multi-port is “in”, then the names of the actual ports are assigned to “in#1”, “in#2”, and so on. These ports should be assigned to different variables in the generated code using “ref” macro. How can we generate the code for the multi-ports? Since the number of ports is not known, it is not trivial to generate the code using codeblocks explained above. In fact, we use StringList object (C++ class) to stitch code fragments to make a whole. Figure 5-7 shows the block definition of Add block.
Note that the section name of the input port is “inmulti” instead of “input” to indicate that it is a multi-port. Now let us look at the “go” section. If the number of input port is $k$ the generated code would look like $\text{out} = \text{out} + \text{input}_1 + \text{input}_2 + \ldots + \text{input}_k$. Since the number of terms in the right side is not known at the block definition time, we make the code at run time by examining the number of connections, or number of ports: “input.numberPorts()” returns this number.

How the code fragments are defined and stitched together into a whole is depicted as bold-face statements in the Figure 5-7. First, we define a StringList object, “out” in this example. Operator “=” is used to initialize the StringList object with the right-hand side string. Operator “<” is used to concatenate the right-hand side string into the StringList. You may add a codeblock into the StringList by using the codeblock name in the right-hand side. After finishing the StringList formation, we call “addCode” method to emit the code.

There is another way of doing this without using StringList object. See 5.2.8 Preprocessor language on page 71 for more advanced usage of codeblocks with arguments.

### 5.2.5 How To Use A New Block in HAE

Once completing block definition, we have to add the block into our environment through dynamic linking. Simply clicking “Import Star” in the “Tool” menu of HAE can do it. A dialogue box will pop up as shown in Figure 5-8(a). Specify the block(or star) name, the domain name, and the directory where the source of the block definition can be found. In the example of the AveDown block, the name will be “AveDown” and the domain will be “CGC”, and the directory will be “$HOME/work”. After clicking OK you will see another dialogue box (Figure 5-8(b)) if linking is successfully performed. If linking fails, an error message is generated in the monitoring window.
If we click “OK”, then another window is popped up to locate the block icon in the local HAE directory. Then, we select the appropriate directory to place the block icon. Initially, the top level directory “library” is open. After selecting the directory, make sure that the file names are all set correctly. And, click “Save” button. Then a new icon is registered in the specified directory and we can see it in the library. Figure 5-4 is obtained through this procedure.

5.2.6 Code Generation Methods

So far we have seen several code generation methods: addCode, addDeclaration, addProcedure, and addInclude. What other methods do you have to know? This section will give you the answer. Figure 5-9 illustrates the code structure of the generated C code from the SDF graph in CGC domain. The code is divided into 7 sections where you may need to put some code segment for each block. That is why you have several methods that emit the code block. Some code fragment is assigned an identifier in order to prevent the same code fragment from being emitted multiple times. If a code generation method is invoked multiple times, the code fragment with an identifier is emitted only once.

To add an include file into the include section, use addInclude(const char* file). The argument itself becomes the identifier.

To add global declarations, use addGlobal(const char* text, const char* name = NULL). The second argument is the identifier of the text argument. If missing, the text becomes the identifier. The global declarations that have the same second argument appear once in generated codes. When a PeaCE block is used several times in an application, the second argument prevents from duplicating the same code. A caution should be made that the name of the global variable is unique. Therefore, a popular identifier should be avoided for the global variable.
To add a **procedure**, use `addProcedure(const char* text, const char* name)`. The second argument is the procedure name that also plays the role of the identifier of the procedure code.

To add local variables for a block into the **main declarations** section, use `addDeclaration(const char* text, const char* name = NULL)`. The second argument is the identifier of the code fragment. The port variables and state variables of a block are declared in this section by default.

To add initialization code of local variables in the **mainInit** section, use `addCode(const char* text)` in the `initCode` method of the block definition file.

To add code into the **mainLoop** section, use `addCode(const char* text)` in the `go` method of the block definition file.

To add code into the **mainClose** section, use `addCode(const char* text)` in the `wrapup` method of the block definition file.

```c
#include <math.h>
int value_0;
void vset() { value_0 = 1; }
int main() {
  int output_0;  int file_2;
  output_0 = 0; file_2 = open(…);
  { int k; for (k = 0; k < 100; k++) {
      /* block code */ …. }
  }
  close(file_2); return 1;
}
```

---

**Figure 5-9 Code structure of a generated C code**

If a block needs special compile options or link options, you have to call `addCompileOption(const char* text)` or `addLinkOption(const char* text)` method to indicate what they are.

### 5.2.7 Macros

You have seen two macros, `$ref` and `$val`, to access a port or a state of the block. More than one block may use the same name for their port variables or state variables that should be distinguished in the generated code. That is why you have to use macros. In this section, we will show what macros are available.

**$ref(name)**

Returns a unique identifier to a state or a port: (ex) `$ref(output)` returns “output_0”
$\textit{ref(name,offset)}$

Returns the reference to an array state or a port with an offset that is not negative: (ex) $\textit{ref(input,2)}$ returns “input_1[2]”.

$\textit{val(state-name)}$

Returns the current value of the state. If the state is an array state, the macro will return a string of all the elements of the array spaced by the new line character. The advantage of not using $\textit{ref}$ macro in place of $\textit{val}$ is that no variable is defined in the generated code.

$\textit{size(name)}$

Returns the size of the state/port argument. The size of a non-array state is one; the size of an array state is the total number of elements in the array. The size of a port is the buffer size allocated to the port.

$\textit{starName()}, \textit{fullName()}$

Returns the instantiated name of the block.

$\textit{starSymbol(name)}$

Returns a unique variable in the block instance scope. Suppose that there are two instances of the same block in a block diagram. And, the block should declare a local variable in the main declaration section. Then, you have to use this macro to assign a unique name to the local variable. An example can be found in the Printer block in the CGC domain. Inside the initCode method, it says “FILE* $\textit{starSymbol(fp)}$;”. Then, multiple instances of the same Printer block are assigned their unique file pointers.

5.2.8 Preprocessor language

In this section, we explain some sections listed in Table 5-1 with more details that haven’t discussed so far. Some sections may require C++ knowledge for your understanding. If you do not know C++, just use the default setting. Some explanations (“state” and “codeblock” sections in particular) are borrowed from the Ptolemy programmer manual because PeaCE uses the kernel of the Ptolemy classic.

(1) \textit{derivedFrom}

Since a block is defined as a C++ class in the PeaCE kernel, we can use the “class inheritance” feature of the C++ language. We may derive a new block from an existent block by adding more features. For example, we may add a new port to an existent block by deriving a new block from the block. In the derived block, we do not have to specify the sections that the parent block already has unless modification is needed. An example can be found in the library: the XYgraph block in the library is “\textit{derivedFrom}” the Xgraph block by adding an additional port.

(2) \textit{hinclude}, \textit{ccinclude}

These directives cause the extra files to be included in the .cc file, or the .h file of block definition (not in the generated code). Each filename must be surrounded either by quotation marks or by "<" and ">" (for system include files like <math.h>). Suppose that we want to augment the following code in the “setup” section of AveDown block of Figure 5-3.
setup { if (int(count) < 0 ) Error::abortRun("this, “ negative count value is not allowed"); }

In PeaCE, we report an error by calling the “Error::abortRun(class reference, string)” method. The first argument is where the error occurs. Inside the block definition, “*this” is used. To use this error-reporting facility, you have to include “Error.h” file in the “.cc” file (CGCAveDown.cc) by specifying

ccinclude { “Error.h” }

Note that these directives are not used to include files in the generated code. To include files in the generated code, we have to use “addInclude” method in the section bodies. You can include the <stdio.h> file in the generated code by the following section definition: initCode { addInclude("<stdio.h>"); }

(3) input, output

These sections define the ports of a block. Here, “name” subsection specifies the port name while “type” specifies the data type. The available types are int, float, fix, complex, message, or anytype. Again, case does not matter for the type value. Although Ptolemy allows some complex types such as matrix types, PeaCE does not. Instead, we extend the usage of the message type in the CGC domain: more detailed discussion can be found in 5.4 Complex and Message Type on section 5.4.

The type of a port is determined after type negotiation between the connected ports. If an integer port is connected to a float port, the type of the source port is converted to that of the destination port. In some cases, for example Mux block, we do not need to specify the type of ports. Then, the port type is declared as “anytype”. Then, the type of the port is resolved by the type of the connected port. If you want to enforce that two anytype ports have the same type after type resolution, then you should use an alternative syntax for the type field of a port; this syntax is used in connection with ANYTYPE to specify a link between the types of two portholes. type { = name } where name is the name of another port. This indicates that this porthole inherits its type from the specified porthole. For example, here is a portion of the definition of the Fork block:

input { name{input} type{ANYTYPE} }
outmulti { name{output} type{= input} desc{ Type is inherited from the input. } }

(4) inmulti, outmulti

These sections define a set of ports with the same type. In some blocks such as Fork and Mux, the number of ports may vary depending on how many blocks are connected to. To make a single block definition support a variable number of port connections, PeaCE use multi-ports. From the block diagram, the number of ports is decided and as many ports are created dynamically. The syntax of these sections is the same as that of the input/output section.

(5) state or defstate

This section is used to define a state or a parameter of the block. The following types of states are allowed: int, float, string, complex, fix, intarray, floatarray, complexarray, or stringarray. Case is ignored for the type argument. The “default” subsection specifies the default initial value of the state; its argument is either a string (enclosed in quotation marks) or a numeric value: default{1.0} or default{“1.0”}. Furthermore, if a particularly long default is required, as for example when initializing an array, the string can be broken into a sequence of strings. The following example shows the default for a ComplexArray:

default {
"(-.040609,0.0) (-.001628,0.0) (.17853,0.0) (.37665,0.0)"
}
For complex states, the syntax for the default value is “(real, imag)” where real and imag evaluate to integers or floats.

The “attributes” keyword specifies state attributes. At present, two attributes are defined for all states: A_CONSTANT and A_SETTABLE (along with their complements A_NONCONSTANT and A_NONSETTABLE). If a state has the A_CONSTANT attribute, then its value is not modified by the run-time code in the block (it is up to you as the block writer to ensure that this condition is satisfied). States with the A_NONCONSTANT attribute may change at run time. If a state has the A_SETTABLE attribute, then you can set the value in HAE. Otherwise, it is invisible from the user interface. If no attributes are specified, the default is A_CONSTANT|A_SETTABLE meaning that they both apply (A_CONSTANT and A_SETTABLE).

To initialize an Array State (FloatArrayState, IntArrayState and ComplexArrayState), you may use a file name with a prefix <. If you have a file named foo that contains the default values for an array state, you can write,

```java
default { "< foo" }
```

If you expect others to be able to use your block, however, you should specify the default filename using a full path. For instance,

```java
default { "< ~/user_name/directory/foo" }
```

The format of the file is also a sequence of data separated by spaces (or newlines, tabs, or commas). File input can be combined with direct data input as in

```java
default { "< foo 2.0" }
default { "0.5 < foo < bar" }
```

A "repeat" notation is also supported for ArrayState objects: the two value strings

```java
default { "1.0 [5]" }
default { "1.0 1.0 1.0 1.0 1.0" }
```

are equivalent. Any integer expression may appear inside the brackets []. The number of elements in an ArrayState can be determined by calling its size method. The size is not specified explicitly, but is calculated by scanning the default value.

(6) **private, protected, public**

These three keywords allow the user to declare extra members for the class with the desired protection. If you want to use a variable in multiple sections, say in “setup” and “go” sections, you may want to define the variable as a private or protected member in the block class definition. Here is an example:

```java
private { int test; }
setup {test = some_statement; }
go { if (test == 0) addCode ("XXX") else addCode("YYY"); }
```

(7) **code**

This section is used to include a code segment after the include files in the block class file. A typical usage is to put #define statements or extern declarations.
(8) constructor
This section is used to specify extra C++ code to be executed in the constructor for the class. It can be of any length. Note that the constructor is invoked only when the class is first instantiated; actions that must be performed before every simulation run should appear in the setup or begin methods, not the constructor. In the CGC domain, you are encouraged to call “noInternalState()” method in the constructor indicating that this block can be parallelized if there is no NON_CONSTANT state. If the current execution of a block affects the next execution, the block should be executed in a serial fashion.

(9) setup
This section defines the setup method of the block, which is called every time the simulation is started, before any compile-time scheduling is performed. It is common for this method to set parameters of input and output portholes, and to initialize states. In some domains, with some targets, the setup method may be called more than once during initiation. You must keep this in mind if you use it to allocate or initialize memory.

(10) begin, initCode
These sections define the begin method and the initCode() method of the block while the initCode section is applicable only to the CGC domain. They are called every time the simulation is started, but after the scheduler setup method is called (i.e., after any compile-time scheduling is performed). They are always called exactly once when a simulation is started. These methods can be used to allocate and initialize memory.

(11) go
This section defines the main function of the block when it is executed. Usually, a block consumes input events (or data) and produces output in this section. In the code generation domain, this section emits the block function code by using addCode methods.

(12) wrapup
This section defines the wrapup method of the block, which is called after the graph execution is completed. The wrapup method is not invoked if an error occurs during execution. Thus, the wrapup method cannot be used reliably to free allocated memory. Instead, you should free memory from the previous run in the setup or begin method, prior to allocating new memory, and in the destructor.

(13) codeblock
This section defines the C-code segment to be emitted by code generation method defined in section 5.2.5.
The explanation of this section is borrowed from Ptolemy programmer manual. Codeblocks are implemented as protected static class members (e.g. there is one instance of a codeblock for the entire class). The codeblock directive defines a block of code with an associated identifying name ("add2" in this case).

codeblock (add2) {
    /* output = input1 + input2 */
    $ref(output) = $ref(input1) + $ref(input2);
}

Simple codeblocks (as described above) have a name and are implemented as static member strings. Extended codeblocks have a name, optional arguments, and are implemented as non-static functions. They have an escape mechanism so that C++ expressions may be evaluated at run time and inserted into the generated code. However,
in order to take advantage of this escape mechanism, a codeblock must be defined and called with arguments, even if those arguments are empty. An example:

```java
void codeblock(cbLoop,"int N, double x") {
    for (i=0; i < @(N); i++) {
        $ref(output,i) = sin(i*@(x));
    }
}
```

This defines a codeblock named `cbLoop` with two arguments: `N` and `x`. The variable `i` will appear in the generated code, while the C++ expressions `N` and `x` are escaped by `@` and will be evaluated at code-generation time. In this example, parenthesis after `@` is omitted since a single identifier is used. When this is called as `cbLoop(5, 0.1);` the following string will be returned:

```java
for (i=0; i < 5; i++) {
    $ref(output,i) = sin(i*0.1);
}
```

This might be used within a `go()` method as: go { addCode(cbLoop(5, 0.1));}

Using this feature, we rewrite the body of Add block discussed in Figure 5-7 in section 5.2.4. The Add block has an input multi-port. Instead of using the `StringList` object, we can use codeblocks with arguments as follows.

```java
void go {
    addCode(StartOp);
    for (int k = 2; k <= input.numberPorts(); k++) addCode(DoOp(k));
}
void codeblock(StartOp) { $ref(output) = $ref(input#1); }
void codeblock(DoOp, "int k") { $ref(output) += $ref(input#@k); }
```

A more complicated example follows:

```java
void codeblock(cbLoop2,"char *portname, int N, double x") {
    for (i=0; i < @(int(length)); i++) {
        $ref(@portname,i) = sin(i*@(x/N));
    }
}
```

In this example, `length` is a data member of the block (typically a state). When called as:

```
void cbLoop2("ina", 3, 0.2);
```

it would generate (assuming the value of `length` is 20):

```java
for (i=0; i < 20; i++) {
    $ref(ina,i) = sin(i*0.6666666);
}
```
In order to trigger the C++ expression processing via @-escapes in codeblocks that would otherwise have no arguments, add in a null argument list as in:

```cpp
codeblock(cbLoop3,"") {
    for (i=0; i < @(int(length)); i++) {
        $ref(output,i) = sin(i*0.1);
    }
}
```

In the example above, the `@(int(length))` will be replaced with the value of the class member `length`. The above example would be called with an empty argument list as: `go { addCode(cbLoop3());}

(14) in-line codeblock

Inside the definition of “go” section, all contiguous blocks of lines with a leading @ will be translated into an in-line codeblock (i.e., an addCode() statement). The @ escape mechanism for C++ expressions works as described above for codeblocks with arguments. Within @-escaped expressions, in-line codeblocks may reference local method variables as well as member variables.

Leading white-space before a leading @ will be ignored. Note that no override mechanism is provided to prevent the in-line codeblock interpretation. Note also that @ has dual meanings: the first @ on the line introduces in-line codeblock mode, while subsequent @ characters on the same line escape into C++ expressions. For example:

```cpp
go() {
    @CMAM_wait( &$ref(ackFlag), 1);
}
```

is equivalent to:

```cpp
go() {
    addCode("CMAM_wait( &$ref(ackFlag), 1);\n");
}
```

A more complicated example:

```cpp
go {
    @$ref(output) = \\
    int ni = input.numberPorts();
    for (int i = 1; i <= ni; i++) {
        @$ref(input#@i) @(i < ni ? " + " : ";\n")
    }
}
```

If "input.numberPorts()" returns 3 when the above program is run, the generated code will be:

"$ref(output) = $ref(input#1) + $ref(input#2) + $ref(input#3);\n"

(15) method

The C++ class associated with a block defines the following methods by default: setup, begin, go, wrapup. You can add another method in the class definition by using this section. Here is an example:

```cpp
virtual method {
    name { incIndex } access { protected } arglist { "(int index)" } type { int }
```
code { return index + 1; }
}

An optional function type specification may appear before the method keyword, which must be one of the following: virtual, inline, pure, pure virtual, inline virtual. You need C++ knowledge to understand how they are distinguished. Fortunately, this qualifier is only optional.

The “access” subsection indicates the level of access for the method. If omitted, protected is assumed by default. The “arglist” section lists the method arguments as a quoted string. If this is omitted, the method has no arguments. The “type” section specifies the return type of the method. If the return type is not a single identifier, you must put quotes around it. If this is omitted, the return type is void (no value is returned). The “code” section implements the method. This is a required item, unless the pure keyword appears, in which case this item cannot appear.

5.3 Scheduling and Code Generation

The fact that a code can be automatically generated from the block diagram is a good thing because the designer is exempted from the coding labor except for block definition. But we have to answer two fundamental questions to enjoy this benefit: how easy is to draw the block diagram and how good is the code quality. This section concerns the latter. If the code quality is worse than a manually written code, the code generation facility of PeaCE will be of little use. Therefore, PeaCE provides many optimization skills that will be explained in this section and sections followed. We believe that you can obtain a fairly good code from PeaCE if you make the best use of PeaCE optimization facility. A main criterion of code quality is the memory requirement: sum of the code size and the data size.

First of all, the block function should be highly optimized and this task is up to the designer who creates the block and puts it to the library. Only after the code quality is confirmed, the block should be put into the official block library in PeaCE. So, we would like to stress that you are also in charge of the code quality as a block designer. In this section, we assume that all blocks are sufficiently optimized.

5.3.1 Scheduler

The memory requirement of the generated code depends on the execution order of blocks. Recall that the SDF model allows us to determine the execution order of blocks at compile time and we call the order the schedule of the graph. The graph execution repeats the schedule of the graph as many times as the designer specified. One execution of the schedule is called an iteration of the graph. The generated code from an SDF graph has a main loop (for-loop) whose body is nothing but the list of block executions determined by the schedule. In general there are more than one feasible schedules among which PeaCE provides the user the option to select some. Consider an SDF graph in Figure 5-10(a).
The scheduling option is selected from the target parameter, “loopingLevel”, as shown in Figure 5-10(a). To view the scheduling result, we set the “write schedule?” parameter to “YES” and run the graph. Figure 5-10(b) shows the schedule of the graph when the “loopingLevel” is set to “DEF” (default) and Figure 5-10(c) shows the schedule when the “loopingLevel” is set to “ACYLOOP”. Contrary to the name, “ACYLOOP”, not “DEF” is the default value PeaCE uses. What does it all about? Let’s look at the procedure how a schedule is constructed.

First, the repetition ratios between blocks are determined from the SDF model of computation. In Figure 5-10(a), the adder block produces one sample per execution while the AveDown block consumes two samples per execution. It means that the adder block should be executed twice more than the AveDown block in one iteration of the schedule. On the other hand, the repetition ratio between the AveDown block and the Xgraph block is 1:1 since they produce or consume the same number of samples per execution. Likewise, we compute the repetition
counts of all blocks. Table 5-2 shows the repetition counts of all blocks. All values are maximally reduced so that there is no common divisor but 1.

<table>
<thead>
<tr>
<th>Block</th>
<th>Waveform</th>
<th>IIDUniform</th>
<th>Adder</th>
<th>AveDown</th>
<th>Xgraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repetition count</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Then, we construct a schedule that contains as many invocations of the blocks as the table indicates. To construct a schedule, we find out the executable block of the graph. Initially two source blocks, Waveform and IIDUniform, are executable (also called “runnable”) since they do not need any input sample. Among runnable blocks, we choose one and put into the schedule: in the example of Figure 5-10(b), the Waveform block is first chosen. After the Waveform block is scheduled, it produces one output sample on its output arc. The Adder block is not runnable until it receives a sample from the IIDUniform block. However, the Waveform block remains runnable until it is scheduled as many times as its repetition count. Thus, there are two runnable blocks: Waveform and IIDUniform. The “DEF” scheduler chooses the IIDUniform block. If the Waveform block is chosen, two samples are accumulated on the output arc of the Waveform block before the Adder block is executed. It means that the generated code should allocate a buffer of size two for the arc. After the IIDUniform block is scheduled, one sample is placed on the output arc. Now, the Adder block is runnable. Among three runnable blocks, the “DEF” scheduler chooses the Adder block to minimize the buffer requirement on the arc. So, you may understand that the “DEF” scheduler tries to minimize the buffer requirement on the arcs.

In the schedule of Figure 5-10(b), however, some blocks appear twice in the list. It means that the generated code contains two copies of their function bodies to make the code size large. Since the code size waste is much larger than the buffer size reduction, the “DEF” scheduler is usually inefficient in terms of memory requirement. A popular method to minimize the code size is to cluster the blocks with the same repetition count and to schedule them as a unit. Then, we can obtain a looped schedule as shown in Figure 5-10(c): { 2(Waveform, IIDUniform, Adder), AveDown, Xgraph}. If you look at the generated code, the main loop contains a loop of the clustered blocks so that the code contains only one copy of each function block. In general, there are many looping possibilities though the simple example of Figure 5-10(a) has only one. Therefore, PeaCE provides three different looping method: CLUST, SJS, and ACYLOOP. The “ACYLOOP” scheduler is most complicated but most efficient when the graph is acyclic. The “CLUST” scheduler is the simplest loop scheduler and the “SJS” scheduler is applicable to the graphs with cycles. In summary, the loop schedulers try to minimize the code size and the ACYLOOP scheduler provides the most memory-efficient schedule when the graph is acyclic.

You may add another scheduler to the PeaCE by augment the scheduling option in the target parameter. Table 5-3 summarizes the “loopingLevel” option.

<table>
<thead>
<tr>
<th>option</th>
<th>Scheduler characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEF</td>
<td>Default SDF scheduler: tries to minimize the buffer size on the arcs.</td>
</tr>
<tr>
<td>CLUST</td>
<td>Basic loop scheduler: tries to minimize the code size</td>
</tr>
<tr>
<td>SJS</td>
<td>Loop scheduler to be used for the graphs with cycles</td>
</tr>
<tr>
<td>ACYLOOP</td>
<td>Default but most complicated loop scheduler that minimizes both the code and the buffer size. It is only applicable to an acyclic graph.</td>
</tr>
</tbody>
</table>
5.3.2 Code Generation Targets

To run a block diagram, you have to specify the target of the graph, depending on the target environment on which the generated code will run. The “default-CGC” target uses the PeaCE server machine to run the generated code. This target is the default target when developing and simulating the algorithm of the system function. If you want to generate the code for a different processor, you have to change the target object: for example “CGC-Armulator” for an ARM processor. You can make your own target though it requires more knowledge on the PeaCE kernel: It is beyond the scope of user’s manual. A separate document will be prepared in the future.

Each target is associated with the target parameters. Table 5-4 summarizes the parameters of the “default-CGC” target. Since the “default-CGC” target is the base target of other derived target, those parameters are shared with other targets. If you do not understand the explanation of an option, just use the default value.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>NULL</td>
<td>Host machine to compile or assemble code on. Default is the server machine.</td>
</tr>
<tr>
<td>directory</td>
<td>$HOME/PTOLEMY_SYSTEMS/CGC</td>
<td>Where to save the generated code</td>
</tr>
<tr>
<td>file</td>
<td>NULL</td>
<td>File name. Default is the given facet name.</td>
</tr>
<tr>
<td>loopingLevel</td>
<td>ACYLOOP</td>
<td>Scheduling option. See section 5.3.1 for details.</td>
</tr>
<tr>
<td>display?</td>
<td>YES</td>
<td>Whether display the generated code or not.</td>
</tr>
<tr>
<td>compile?</td>
<td>YES</td>
<td>Whether compile the generated code or not.</td>
</tr>
<tr>
<td>run?</td>
<td>YES</td>
<td>Whether run the compiled code or not.</td>
</tr>
<tr>
<td>write schedule?</td>
<td>NO</td>
<td>Whether write the schedule of the given block diagram or not.</td>
</tr>
<tr>
<td>staticBuffering</td>
<td>YES</td>
<td>Static buffering simplifies buffer indexing in the generated code, but with buffer size overhead in some cases.</td>
</tr>
<tr>
<td>funcName</td>
<td>main</td>
<td>Function name of the whole graph execution.</td>
</tr>
<tr>
<td>compileCommand</td>
<td>gcc</td>
<td>Which compiler to use.</td>
</tr>
<tr>
<td>compileOptions</td>
<td>NULL</td>
<td>Compiler option can be added such as “-g”.</td>
</tr>
<tr>
<td>linkOptions</td>
<td>-lm</td>
<td>By default, math library is linked together.</td>
</tr>
<tr>
<td>resources</td>
<td>STDIO</td>
<td>A set of special resources that the target has. This information is needed for assigning a block into the target when the block requires a special resource.</td>
</tr>
<tr>
<td>optLevel</td>
<td>0</td>
<td>If non-zero, code sharing option is turned on. If a block is defined in a functional style as discussed in 5.2.2, optLevel 0 is just O.K.</td>
</tr>
<tr>
<td>bufferSharing</td>
<td>NO</td>
<td>If this option is turned on, buffer sharing is applied in the generated code. Useful for message type data handling. See section 5.4.</td>
</tr>
</tbody>
</table>
Among these target parameters, “staticBuffering” needs more explanation. Suppose that two blocks A and B are connected together and block A produces 2 samples and B consumes 3 samples per execution. The repetition ratio between A and B becomes 3 to 2. And, assume that the schedule is AABAB. Then, the maximum number of samples accumulated on the arc AB is 4. If we allocate a buffer of size 4 to the arc AB, block B should access the buffer with a modulo addressing. If we increase the buffer size to 6, no modulo addressing is needed and the buffer index can be statically determined. We call such optimization as “static buffering”. Static buffering however costs extra buffer space in general. Figure 5-11 shows changing buffer requirement as changing schedule.

![Figure 5-11 Change of buffer requirement as the change of schedule](image)

Now, we list all targets that PeaCE currently supports for the CGC domain in Table 5-5. The first two targets differ from each other depending on which processor we are targeting. The next three targets are related with fixed point conversion of the program. If the target processing element is an embedded processor without floating point unit or an ASIC implementation, floating point arithmetic is very inefficient. Instead, the common practice is to convert floating point arithmetic to fixed point arithmetic before implementation.

To use the fixed point arithmetic, we have to convert all floating numbers to the corresponding fixed point numbers by determining the word-length and the radix point position. You have to apply all three targets in the listed order to determine the fixed point format of each variable before ASIC implementation and software implementation of an embedded processor. After you obtain the fixed number format of all ports and states variables, you should redraw the application by using integer version of blocks and shifter blocks for code synthesis. Note that all blocks should be rewritten manually for fixed-point implementation. And PeaCE does not consider any internally defined floating-point variable while it does with port and state variables.
### Table 5-5 CGC targets

<table>
<thead>
<tr>
<th>Targets</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default-CGC</td>
<td>Default target for the CGC domain. Generate and run the code in the server machine.</td>
</tr>
<tr>
<td>RangeEstimation</td>
<td>Estimates the range of the port and the state variables to determine the maximum word-length of the variables through simulation. Output files store the word length information of the variables and their statistics: “word length file” and “estimated range file”.</td>
</tr>
</tbody>
</table>
| WLOptimization  | Determine the word length and the radix point position of each variable. It receives the “word length file” from the “FixedEstimation” target as the input file. It has the following parameters:  
  “use maximum WE” (default: YES) – use the same word-length for all variables.  
  “maximum WE” (default: 32) – the maximum word length size up to 32.  
  “use saturation” (default: NO) – use the saturation arithmetic. Default is “overflow”.  
  “signed” (default: YES) – assume two’s complement numbers.  
  “quantization mode” (default: truncation) – use truncation or rounding  
  “desired SQNR” (default: 60.0 dB) – depending on this value, word-length is decided.  
  “optimization report file”(default: optimal.txt) – output file name |
| FixedEstimation | Verify the integer word length of each variable and increase it if necessary. Integer word length is computed from “RangeEstimation” target using the floating point simulation. This target uses fixed point simulation. The input file is the “optimization report file” given from the WLOptimization target. The “word length file” and the “estimation range file” are two output files. |

### 5.4 Complex and Message Type

So far we assume that the data types of ports and states are primitive: integer or float. In this section, we explain how to define blocks that use “complex” data or “structure-type” data that are heavily used in multi-media signal processing applications. Let us first find out how to use complex data type.

#### 5.4.1 Complex Type

Figure 5-12 shows the definition of an adder block of complex data: CGCAAddCx block.
In the generated code, the complex type is defined as followed:

```c
typedef struct complex_data { double real; double imag; } complex
```

Since $ref(input) or $ref(output) is expanded into a “complex” type variable, we use “.real” or “.imag” suffix to access the real or imaginary member. And, we use a codeblock with argument to emit the code with varying number of input connections. Refer to 5.2.8 for the explanation of this type of codeblock definition.

In case complex computation is used inside the codeblock, we define the following macros for complex computation, as can be found in CGCFIRCx block definition for example.

```c
#define COMPLEX_ADD(a,b,c) c.real = a.real + b.real; c.imag = a.imag + b.imag
#define COMPLEX_MUL(a,b,c) c.real = a.real * b.real – a.imag * b.imag; c.imag = a.real * b.imag + a.imag * b.real
```

It should be noted that such definition should be replicated in any block you want to use. Since we use addGlobal() method, only one definition will appear in the generated code even though the definition is used in several blocks.

**5.4.2 Message Type**
The primary support for structure types in PeaCE is the message data type, which is very powerful. To describe the message data type in a block code, you set the port type to “message” type. Moreover additional specifications are required to be made in the setup stage for the message data type in setup stage. In the setup section, you should specify message name and message size.

```
input { name { msgIn } type { message } }
...
setup {
    msgIn setMessageName("struct XXX");
    msgIn setMessageSize(16);
}
```

The setMessageName(const char*) function call is mandatory. If you do not specify the name or you specify a different name from the connected porthole, then you will meet an error “Mismatch Message Name”. On the other hand, setMessageSize(int) is an additional information that is for memory optimization. You notify the size of message structure in setMessageSize(int). Usually the message data type requires a large data memory so it is suggested that you turn on memory sharing algorithm and set the message size. If the specified message size is less than the required memory size, the synthesized code will have a semantic error in which new data is overwritten into a memory before data in the memory is read.

In C code generation, the message name indicates the data type of the port. Therefore you should define the message structure definition code. For the above example, “struct XXX” can be defined in codeblock defStructXXX as follows.

```
codeblock(defStructXXX) {
    struct XXX {
        int data[4];
    }
}
initCode {
    addGlobal(defStructXXX);
}
```

Since struct XXX is defined in the generated code and the PeaCE does not know the size of the struct XXX, do not set the size like “msgIn.setMessageSize(sizeof(struct XXX));”. Instead, you can specify the message size like “msgIn.setMessageSize(4*sizeof(int));”.

It is easy to use message port in the block code if you remember that the message name string replaces the port. Consider the following example code.

```
$ref(msgIn).data[3] = 2;
```

Since msgIn has struct XXX as message name, it will be defined as “struct XXX msgIn” in the generated C code. Therefore if you want to access the msgIn then you access the member of struct XXX.
5.4.3 Message Type Demo

Let us examine an image processing demo with message data type, which includes edge detection algorithms. Open a schematic of ./schematic/Peace/Demo/CGC/Image/edgeDetect. You can see Figure 5-13. This demo compares four different edge detection algorithms: Sobel, Roberts, Prewitt and Frei-Chen. The green port (or green triangle) represents that it is message type.
After you run the demo, you can see the following generated code which requires 5 frames of 512x360 size since one frame is needed between the ‘ReadImage’ block and four edge detect blocks and four frames are between four edge detect and display blocks (Figure 5-15).

![Generated code for edgeDetect demo](image)

Figure 5-15 Generated code for edgeDetect demo

Figure 5-16 shows the results.

![Edge detection results](image)

Figure 5-16 Edge detection results
In this example the generated code requires large buffer size. In order to minimize the memory for message data type, set ‘YES’ to the target parameter of ‘bufferSharing’.

Then, two frames of 512x360 size are required since the memory between edge detect block and display block is reused. Therefore we recommend that you always turn on the buffer sharing parameter when you use message data type. Figure 5-17 shows the generated code.

![Figure 5-17 Generated code with bufferSharing option is set to “YES”](image)

### 5.4.4 Build a Star with Message Data Type

Let us make a new block with message data type, which reduces the frame size by the specified ratio. The input frame size is determined by two parameters “width” and “height”.

First, copy $PEACE/src/domains/cgc/stars/CGCGain.pl to $HOME/work/CGCMyDownImage.pl and modify the code as the following code.

```plaintext
defstar {
    name { MyDownImage }
    domain { CGC }
    author { H. Oh }
    input { name { input } type { message } }
    output { name { output } type { message } }
    defstate { name { width } type { int } default { 512 } desc { image frame width } }
    defstate { name { height } type { int } default { 360 } desc { image frame height } }
    defstate { name { count } type { int } default { 2 } desc { down image ratio } }
    hinclude { "Message.h" }
    constructor { noInternalState(); }
}
```
setup {
    StringList inputFrameName,outputFrameName;
    inputFrameName << "struct UcharMatrix" << int(height) << "x" << int(width);
    outputFrameName << "struct UcharMatrix" << int(height)/int(count) << "x" << int(width)/int(count);
    input.setMessageName((const char*)inputFrameName);
    input.setMessageSize(int(width)*int(height)*sizeof(unsigned char));
    output.setMessageName((const char*)outputFrameName);
    output.setMessageSize(int(width)*int(height)*sizeof(unsigned char)/(int(count)*int(count)));
}

codeblock(defFrame) {
    struct Frame$val(width)x$val(height) {
        unsigned char data[$val(height)*$val(width)];
    };
}

codeblock(block) {
    int i,j;
    for(i=0; i<$val(height); i+=$val(count))
        for(j=0; j<$val(width); j+=$val(count))
            $ref(output).data[(i*$val(width)/$val(count)+j)/$val(count)] = $ref(input).data[i*$val(width)+j];
}

initCode { addGlobal(defFrame); }
go { addCode(block); }

The code in setup represents what data type each port has. As the frame size is changed by the block parameters, the message name is changed accordingly.

After defining the new block, we register it (Figure 5-18). Note that since the DownImage block is already registered, you should regard the word of “DownImage” as “MyDownImage”.

![Figure 5-18 Register the block](image)

Now build a demo using the new block. First create a new schematic in File/New. The schematic name is “downImageTest” (Figure 5-19).
Draw a schematic as Figure 5-20. The library block for image processing can be found in SPDF/Processing/Image And Video.

Now, we change the target name and add the galaxy parameter by using ‘Add Parameter’ button. Add ‘width’, ‘height’ and ‘count’ and set 512, 360 and 2 respectively (Figure 5-21).
Change the parameters of ‘ReadImage’, ‘DownImage’ and ‘DisplayImage’ as shown in Figure 5-22. And run it.
Figure 5-22 Change block states
You can see the following results including the generated code in which the $val()$ macro is replaced by the value of the state (Figure 5-23 and Figure 5-24)

![Generated code](image1)

$\text{Figure 5-23 Generated code}$

![Result for downImage](image2)

$\text{Figure 5-24 Result for downImage}$

### 5.5 Fractional Rate Block Definition

Composite data types, such as video frame and network packet, are used extensively in recent networked multimedia applications and become the major consumer of the scarce memory space in an embedded system. Existent dataflow models have inherent difficulty of efficiently expressing the mixture of a composite data type and its constituents, for example a video frame and the macro blocks. A video frame is regarded as a unit of data sample in the SDF model and should be broken down into multiple macro blocks explicitly by data copying. It incurs time and memory overhead. To overcome this difficulty, we propose a new extension of SDF, called fractional rate dataflow (FRDF) in which fractional number of samples can be produced and consumed. In the proposed FRDF model, a constituent data type is considered as a fraction of the composite data type. An FRDF graph can be synthesized into efficient codes with shorter latency and significantly less buffer memory than the corresponding SDF graph.
Fractional rate is also defined for the primitive data type. Then the fraction should be understood statistically. Consider the following down sample example that shows the differences between SDF model and FRDF model (Figure 5-25 and Figure 5-26)

![Figure 5-25 Down sample in SDF](image1)

![Figure 5-26 Down sample in FRDF](image2)

In the SDF model, the down sample block reads $n$ inputs at once and produces one output where $n$ is 4 in this figure. However in FRDF model, it reads one input and produces one output every $n$ executions. Then, the output sample rate is said to be $1/4$: one output out of 4 executions. There is no difference of functionalities between two models even though the down sample block in SDF requires 4 buffers as input buffer and it needs only one buffer in FRDF.

### 5.5.1 Fractional Rate for Atomic Type

As above example, consider atomic types such as int, float, and complex. The notion of fractional rate is the average of the output sample rates. For example, $1/4$ rate means that one sample is produced or consumed per four execution of the block without knowing when the sample is produced or consumed. More over, $2/8$ rate means that two samples are produced or consumed during 8 executions of the block. Therefore the rate of $1/4$ is different from that of $2/8$: the block writer is responsible to define the correct fractional rate.
5.5.2 Fractional Rate for Message Type

The FRDF is more efficient and intuitive for message data type. Since the message type data can be read or filled partially per block execution, the fractional rate is interpreted as how much ratio the sample is read or filled per block execution.

For example, open a demo of “./schematic/Peace/Demo/CGC/Image/DctImage” (Figure 5-27). When you enable “View/Rate” in Menu and run the schematic, you can see the schematic that represents the rate. In this demo, the input of MatrixDecom and the output of MatrixCom have fractional rates. The schedule becomes “ReadImage BlackHole 1620*(MatrixDecom DCTImage8x8 IDCTImage8x8 MatrixCom) Const DisplayImage”. If the fractional rate is not supported then the schedule will become “ReadImage BlackHole MatrixDecom 1620*(DCTImage8x8 IDCTImage8x8) MatrixCom Const DisplayImage” which requires 1620 times as large as memory size between MatrixDecom and DCTImage8x8, and IDCTImage8x8 and MatrixCom compared with FRDF implementation.

When the data sequence accessed by a source block is not equal to that by a sink block, the partial access is not allowed. The consumer and the producer should have the same interpretation on the fraction of the composite data type. If they do not agree, the data type should be considered as atomic.

Suppose that there is a two-dimensional array and the producer regards it as an array of row vectors while the consumer regards it as an array of column vectors. In this case, the two-dimensional array may not be regarded as a composite type data. Figure 5-28 illustrates this fact pictorially.
In order to specify the access pattern and atomicity of the data sample, you can use two more functions related with FRDF: setAccessSequence(const char* sequenceName) and setAtomicType(int flag). The function of setAccessSequence indicates the pattern of accessing a sample. We suggest that you use the following name pattern since we already build blocks with message type ports as the following rule.

1. use brackets to notify the dimension of the data structure. For example, if you define the structure type as “struct XXX { int data[176][144]; }” then you set setAccessSequence(“[[]]”).

2. fill the number of accessing samples accessed as a unit in the bracket. For example, if the port with “struct XXX” is accessed by 16x16 macroblock unit then you should set setAccessSequence(“[16][]”) since “[16]” indicates that the struct XXX is accessed by 16 rows simultaneously. Similarly, if a 2D video frame is given and it is accessed by row, column and 8x8 block then sequenceName is “[]”, “[1]” and “[8][]” respectively.

![Access Sequence Rule](image)

Figure 5-29 Access sequence rule. If array a is accessed as a unit of [8][16][16] then call setAccessSequence(“[8][16][]”).

In FRDF, you can use additional macro $phase(port). The $phase(portName) macro generates a variable that indicates how many times the block is invoked. For example, if the port of “output” is defined as “setSDFParams(1,0,10)” and you want to initialize the value of the output port every ten times, you can write the code as following:

```java
output{ name { output } type { int }}
setup {
    output.setSDFParams(1,0,10);
}
codeblock(block) {
    if($phase(output)==0)
        $ref(output) = 0;
}
```

Note that you should not use $phase(thePort) when the “thePort” has integer rate since the phase variable is not defined in the synthesized code.

### 5.5.3 Make a new FRDF star
Let us convert an SDF DownSample block to a fractional rate block as shown in Figure 5-30. DownSample block receives $n$ samples and sends only one sample. State “factor” indicates the number of samples that DownSample block must receive for sending one sample and state “phase” indicates which sample must be sent. If “phase” is set to “2”, the third sample among $n$ samples will be sent. You can reduce the buffer size by using FRDF.

First build the following code to $\text{HOME/work/CGCFRDownSample}$.

Please focus on the bold statements in the following code. In the setup section, add “output.setSDFParams(1,0, int(factor));” for setting the rate of port. The denominator of the fraction rate is specified as the third argument of “setSDFParams” method. So it means that the output rate is $1/n$. And then, in codeblock section, you should check whether block’s invocation number gets equal to state “phase”. If it does, block sends that sample to the output port.

defstar {
    name { FRDownSample }
    domain { CGC }
    input {
        name {input}
        type {float}
    }
    output {
        name {output}
        type {float}
    }
    state {
        name {factor}
        type {int}
        default {2}
        desc { Downsampling factor. }
        attributes { A_SETTABLE }
    }
    state {
        name {phase}
        type {int}
        default {0}
        desc { Downsampling phase. }
        attributes { A_SETTABLE }
    }
}
constructor { noInternalState();

setup {
    output.setSDFParams(1,0,int(factor));
    if (int(phase) >= int(factor))
        Error::abortRun("this, ": phase must be < factor");
}

codeblock (sendsample) {
    if($phase(output)==$val(phase))
        $ref(output) = $ref(input);
}

go { addCode(sendsample); }
}

After defining the block, we register the star (Figure 5-31).

![Figure 5-31 Register the block](image)

Next, build a new demo with the FRDownSample star (Figure 5-32).
The following code shows the generated code. In the code, output_4_phase is generated from $phase(output) in CGCFRDownSample.

```c
/* main function */
int main(int argc, char *argv[]) {
    double value_3;
    double output_0;
    double output_1;
    int output_4_phase = 0;

    mainInit:
    value_3=0.0;
    output_0 = 0.0;
    output_1 = 0.0;
    (int sdfLoopCounter_5;for (sdfLoopCounter_5 = 0; sdfLoopCounter_5 < 10; sdfLoopCounter_5++) {
        (int sdfLoopCounter_2;for (sdfLoopCounter_2 = 0; sdfLoopCounter_2 < 2; sdfLoopCounter_2++) {
            /* star downSampleTest.RampI0 (class CGCRamp) */
            output_0 = value_3;
            value_3 += 1.0;
        }
        /* star downSampleTest.FRDownSampleI2 (class CGCFRDownSample) */
    }
}

98
```
if(output_4_phase==0)
    output_1 = output_0;
output_4_phase = (output_4_phase+1)%2;
}
}
} /* end repeat, depth 1*/
{
   /* star downSampleTest.PrinterI5 (class CGCPrinter) */
   printf("%f\n", (double) (output_1));
   printf("\n");
}
} /* end repeat, depth 0*/

return 1;
}

5.6 Matrix and Image Blocks

In this section, we explain the matrix and image blocks and explain the demos (Figure 5-33).

Figure 5-33 Matrix and image blocks
All matrix and image blocks are inherited from “CGCMatrix.pl.” It has states of “Width”, “Height” and “Type”. The “Width” is the number of columns and “Height” is that of rows of the matrix. The “Type” indicates the type of the entry in the matrix. Each state has default value: “Width” has “width”, “Height” has “height” and “Type” has “Type” as default values. Therefore you are recommended to define “width”, “height” and “Type” as super-block(galaxy) states.

Note that you are recommended to avoid a word of “type” since “type” is a reserved word for Conditional Construct: if-then-else that will be explained in the next section. In case that you use a piggyback block, you should use the other name instead of “type”. This name conflict is due to the fact that Matrix and Image data types are inherited from the Ptolemy classic. Even though you may use Matrix and Image data types and the associated blocks, we recommend you to use the message type blocks for handling large size data samples since the buffer sharing optimization is performed only with the message type samples.

For the matrix blocks, you can find them in “library/SPDF/Processing/Matrix”. In “schematic/Peace/Demo/CGC/matrix/MatrixTest1”, you can test matrix blocks (Figure 5-34).

![Matrix test demo](image)

Figure 5-34 Matrix test demo

You can see the following results (Figure 5-35).
The image library is located in “library/SPDF/Processing/Image and Video”. And simple demonstrations are included in “./schematic/Peace/Demo/CGC/Image”.

Open “./schematic/Peace/Demo/CGC/Image/blendImage” which mixes two input images (Figure 5-36).

5.7 Piggybacking State-Update Request
In the SDF model, a state (or parameter) of a block is invisible and inaccessible from other blocks. Sometimes, we want to update the parameter of a block at run-time. Let’s draw a simple SDF graph that consists of a sine-waveform galaxy, a gain block, and a display block as shown in Figure 5-37.

![Figure 5-37 A simple SDF graph](image)

If you run this graph 100 times, you get a period of sampled sine waveform with amplitude 1. Now, assume that you want to change the amplitude of the sine waveform at run time: say increase the amplitude by 1 after each period. There are several approaches you may think of. First, you can create another block that can change the “gain” value periodically with another state called “period”. Second, we can add to the Gain block a port that receives the gain value from the outside. Or we can redraw the block diagram replacing the Gain block with a multiplier block that receives the gain value from an input port. The first two approaches have a serious drawback that you have to create a new block just to update the state value. If a block has many states, you may have to create too many associated blocks for state update requests. So, they are abandoned. The last approach of redrawing the graph with other blocks is not possible in general. Can you think of another way to solve this problem?

In PeaCE, we solve this problem by extending the SDF model: defining a special block called “Piggyback” block. The theoretical treatment of this extension can be found in a technical document that is likely to be beyond your concern.

NOTE: PeaCE supports SPDF extension only for single processor implementation. As of now, you may not use SPDF for HW/SW codesign flow.

The Piggyback block receives two inputs, one is the data input and the other is the state-update input. Figure 5-38 shows how the graph is modified.
A Piggyback block is inserted on the arc between the sinegen block and the Gain block. The “param” input port of the Piggyback block is fed from the Ramp block that generates the “gain” parameter value of the Gain block. The state list of the Piggyback block is shown in the left column where the “period” parameter is set to 100. It means that one sample from the “param” port is associated with (or piggybacked to) 100 samples from the “input” port. The samples from the “input” port are passed to the output port and delivered to the Gain block. When the Gain block consumes a sample, it refers to the piggybacked state value and copies it to the “gain” state if the value is changed. Because the 100 samples compose a period of the sine waveform, the “gain” value of the Gain block is updated after each period as demonstrated in the Xgraph display.

Here is the procedure you have to follow in using the Piggyback block.

(1) Insert the Piggyback block on the data path to the target block that you want to change the parameter (or state) value dynamically at run-time. And make sure that the param” port of the Piggyback block receives the updated state value per execution.

(2) Set the “period” state of the Piggyback block to the period of the state update in terms of the data sample counts. And set the “offset” state if necessary. The “offset” value indicates how many samples will use the initial state value without piggybacked state update information. In the example of Figure 5-38, the offset value is set to 0. If you change the offset value to 10, The periodic state update will start after initial 10 samples are processed at each iteration.

(3) Set the “stateName” state of the Piggyback block to the name of a state in the parent galaxy (super-block). You have to add a super-block state by clicking “Add Parameter” button. Then, a window is open as shown in Figure 5-38, asking you to put the type and the state name. We set the state name to be “gvalue” in this example.
Finally, we set the “gain” parameter of the target Gain block to “gvalue”. Now you are ready to run the modified graph.

The super-block state plays the role of the global state whose scope is within the super-block. The Piggyback block updates the global state. **There may be more than one target blocks that update a local state with the global state value.** In short, The Piggyback block is the sole writer of the global state and the target blocks are the readers. Single write and the scheduled read guarantee the correctness of dataflow model. Note that the pure SDF model does not allow to use any global state. But, our extended SDF model does. So, we call this extended SDF model as “**SPDF, Synchronous Piggybacked Data Flow**” meaning that a data sample can be piggybacked with a state-update request.

If you look inside the Piggyback block definition, you can notice that the “param” port has a fractional sample rate, 1/(“period” state value). Thus, one iteration of the graph consists of 100 invocations of the sinegen, Piggyback, Gain, and Xgraph blocks. That is why we set the “When to stop” value of the control panel to 5 instead of 500 to specify 5 periods of sine wave.

In some cases, the period of the state update is not fixed at run-time. Then, you have to set the “period” value of the Piggyback block to “1”.

Without this piggyback facility, the state value should be delivered through a data arc, which incurs data copy overhead in the generated code. By using this Piggyback block, we could save some data memory space and data copy overhead using pointers in the generated C code.

You may find that a single addition of the Piggyback block enhances the expression power of the SDF model significantly. We can make a block executable conditionally by adding a condition parameter that refers to a global state. In the block body, the condition parameter is first examined whether to execute the block body or just to produce the default output values. Using this conditional execution capability is a significant improvement from the SDF model. But we do not stop here. Using the piggyback block, we will express the dynamic constructs graphically, which is the topic of the next section.

### 5.8 Dynamic Constructs

Before reading this section, make sure that you understand the previous section on SPDF.

Dynamic constructs such as if-then-else and data-dependent iteration can be found in most advanced multimedia algorithms. Therefore, how to express the dynamic constructs is a very important issue in using a formal model of computation. Although the SDF model has many advantages for design reuse, static analyzability, it cannot be used for real examples since it can NOT express the dynamic constructs efficiently. Recently, some approach of combining the SDF model and the FSM model appears to express the dynamic constructs. But, PeaCE provides a much simpler and more intuitive way of expressing the dynamic constructs. Currently, we support two types of dynamic constructs: if-then-else and for.

**NOTE:** PeaCE supports SPDF extension only for single processor implementation. As of now, you may not use SPDF for HW/SW codesign flow.
5.8.1 Conditional Construct: if-then-else

Figure 5-39 If-then-else construct representation in PeaCE

Figure 5-39 shows a toy example how to express an “if-then-else” construct in PeaCE. This graph shows a simple statement: if (c == 0) z = x + y; else if (c == 1) z = x-y; The if-body and the else-body are defined in a separate super-block as shown in the figure. Then, we use the “piggybacking” technique to execute the super-block conditionally. In this example, the condition variable is provided by the WaveForm block that produces a periodic sequence of {1 0 0} as output values. We put a Piggyback block on the data path to both super-blocks. And, we append a Mux block to select one of two super-block outputs according to the condition variable.

At last, we define the following three states in the super-blocks. They are “type”, “conName” and “conValue”. The “conName” state is the target state to be updated by piggybacked state value. In this example, we name the global state as “condition” so that the “conName” state value is set to “condition”. The “conValue” state defines the condition value that is compared with the “conName”. The “type” state defines in which dynamic construct this super-block resides. If the “type” is set to “if”, then the following code template is generated:

```java
if (conName == int(conValue)) { if_galaxy_body; }
```

Now, we summarize how to draw a conditional construct in PeaCE.
(1) Draw each if-body as a super-block. You may have any number of if-bodies to be distinguished by their “conValue” super-block parameter.
(2) Insert a Piggyback block on the data path to each super-block. The conditional variable should be piggybacked and delivered to each super-block to update the “conName” super-block state. Even though there are multiple input ports to a super-block, you don’t have to have a Piggyback block on all input paths. It is enough to have at least one port to receive a piggybacked data.

(3) Append a MUX block to select one of the if-bodies according to the condition value.

(4) Make sure that each if-body super-block defines the following three states: type, conName, and conValue. If any one is missing, an error signal is reported.

5.8.2 Data Dependent Iteration: for

Figure 5-40 demonstrates how to represent a “for” construct in PeaCE. Similarly to the if-then-else construct, we first encapsulate the for-body inside a super-block. In this example, the for-body receives an integer sample and counts down to 0 and displays the values. As the integer value varies, the number of executions of the for-body also varies. We piggyback to the data input the condition variable that indicates how many times the super-block will be executed once invoked. At last, we add two states to the super-block: the “type” state defines which dynamic construct is the super-block for, and the “conName” state is the target state of condition variable.

![Figure 5-40 for construct in PeaCE](image)

When the “type” of the super-block is set to “for”, the following code template is produced:

```c
{ int __i; for (__i = 0; __i < conName; __i++) { for_galaxy_body; } }
```
Unlike the if-then-else construct, there is a restriction on the block at the inside super-block boundary: the input or output sample rate should be dynamic. In other words, the inside ports at the super-block boundary all have a dynamic sample rate since it is unknown that how many times the super-block would be executed. In this example, the input port of the DownCounter block has a dynamic sample rate.

Let’s look inside the block definition of the DownCounter block (Figure 5-41). As shown in the “input” port section of Figure 5-41, a port with a dynamic sample rate is specified with the “num” field equal to 0. And, in the setup section, the sample rate of the “input” port is set to be a fractional rate where the denominator is given by the “count” state value. And the “count” state value is copied from the “conName” state of the super-block at run time. It means that the number of executions of the DownCounter block to consume one input sample varies depending on the “conName” state value of the super-block. If a block has a dynamic port, static scheduling for the super-block is not possible. But, we perform static scheduling at compile-time assuming that the dynamic sample rate is unity. This is why we need a “public” section in this block definition. This section means that this block is regarded as an SDF block inside the “for” construct when the static scheduling is performed.

```c
defstar {
    name { DownCounter }
    domain { CGC }
    input {
        name { input } type { int } num { 0 }
    }
    output {
        name { output } type { int }
    }
    state {
        name { count } type { int }
        default { "1" } desc { the sample count }
        attributes { A_NONCONSTANT|A_SETTABLE }
    }
    state {
        name { ix } type { int } default { "0" }
        desc { monitoring the sample count }
        attributes { A_NONCONSTANT|A_NONSETTABLE }
    }
    state {
        name { ival } type { int } default { "0" }
        desc { current output value }
        attributes { A_NONCONSTANT|A_NONSETTABLE }
    }
}

public {
    /* virtual */ int isSDFinContext() const
    { return TRUE; }
}

setup {
    input.setSDFParams(1,0, int(count));
}

codeblock (block) {
    if ($ref(ix) == 0) {
        $ref(ival) = $ref(input);
        if ($ref(ival) != $ref(count) - 1) {
            printf("wrong sample rate: for
            construct");
            exit(1);
        }
        $ref(ix) = 1;
    }
    $ref(output) = $ref(ival)--; 
    if ($ref(ival) < 0) $ref(ix) = 0;
} go {
    addCode(block);
}
```

Figure 5-41 Block definition of DownCounter block

Since the input port of the DownCounter block has a fractional sample rate, we need to define a phase variable in the function body definition. The “ix” state is used as the phase variable in this block. Since the state is invisible from the user, the attribute of the “ix” state is set to A_NONSETTABLE. If a state is set to A_NONSETTABLE, it is invisible in the user interface. We define another A_NONSETTABLE state called “ival” to remember the current output value.

When you use a “for” construct, you may want to create your own block that has a dynamic port. Then, use the DownCounter definition as a template and modify it to your own purpose.

### 5.8.3 Nested Dynamic Constructs

You can nest the dynamic constructs at will. Figure 5-42 shows a simple example in which a “for” construct contains an “if-then-else” construct. Since the piggybacking relation is valid inside the super-block scope, two constructs do not interfere each other.

![Figure 5-42 A simple demo of nested dynamic constructs](image)

**Figure 5-42** A simple demo of nested dynamic constructs

### 5.9 Interactive Simulation with Tcl/tk Blocks

108
It would be more interesting if you can monitor the progress of simulation or adjust the simulation inputs during simulation. Such interactivity is provided by Tcl/tk block in PeaCE (inherited by Ptolemy). Tcl is an interpreted "tool command language" designed by John Ousterhout while at UC Berkeley. Tk is an associated X window toolkit. Both have been integrated into Ptolemy. This section explains how to use these blocks. If you are familiar with the Tcl language, you can create such blocks from scratch.

The principal use of Tcl/Tk is to customize the user interface of the simulation. Blocks can be created that interact with the user in specialized ways, by creating customized displays or by soliciting graphical inputs. Figure 5-43 shows an example, “eye”, that uses Tcl/Tk blocks, which is in the “communications” demo library. Tk blocks start their names with “Tk” prefix so that you can easily identify which blocks are Tk blocks. In the example of Figure 5-43, there are two Tk blocks: TkSlider and TkPlot. If you use Tk blocks, you should use “TclTk_Target” instead of “default-CGC”. When you run the graph, there comes up a Tk control window, called “Control panel”. Note that a slider control, “Scale”, is inserted inside the control panel. The TkSlider block puts such a slider in the Tk control panel. The TkPlot opens an additional Tk window that displays the output in an interactive fashion. Note that the number of iterations is set to a large number, so that you have enough time to move the slider to notice the display dynamically changing.

Figure 5-43 An example graph using interactive display and simulation control
There are various kinds of Tk blocks for interactive display and simulation control in the “TclTk” block library. Some examples are TkButtons, TkCheckButton for simulation control and TkShowValues, TkXYPlot for interactive display.

In this manual, we do not include how to write your own Tcl/Tk block. It can be found in the Ptolemy classic manual from the Ptolemy classic web site since we copied their implementation.

5.10 HOF (High-Order Function) representation

If you need to draw a repeated structure multiple times in a block diagram, it gets tedious to instantiate all the blocks separately and draw connections. You can save your effort if you use HOF (High-Order-Function) blocks. A HOF block is not a computation block but a graph generator. It is beyond the scope of this document why it is named to HOF: a high order function is a function that returns a function.

Figure 5-44 shows an example that is a truncated version of Fourier series of a square wave: 10 sinusoids are added to approximate a square wave. Instead of drawing 10 sets of sine generators and Gain block, we can use two HOF blocks, SrGr and MapGr blocks to express the regularity of the graph structure.

Figure 5-44 An example that uses two HOF blocks and bus connections

The number of blocks to be instantiated by those HOF blocks is determined by the number of connections with their neighbors. Note that the output connection of the MapGr block to the Add block is not a normal connection arc. Instead, it is a “bus” connection. When you draw this line, you have to use “bus-line” command under the “Library menu”. A bus line has a parameter called “BusWidth”. In this example, the BusWidth is set to
“number_of_terms” to refer to the state of the top-level application graph. Since the “number_of_terms” is set to 10 as shown in the Figure, the MapGr block has 10 output connections. So, 10 Gain blocks are instantiated and their outputs are connected to the multi-input port of the Add block. Similarly, the SrcGr block is bus-connected to the MapGr block with 10 connections: 10 “sinegen” blocks are created and connected to 10 Gain blocks, one by one.

A SrcGr block or a MapGr block has a parameter, called “parameter_map”, of a stringArray type. The parameter_map parameter is used to set parameter values in the replacement blocks. The parameter_map is a string array, a list of strings. The strings are in pairs, where the pairs are separated by spaces, and there are four acceptable forms for each pair:

- name value
- name(value)
- name = value
- name(value) = value

There should be no spaces between name and (number), and the name cannot contain spaces, =, or (. In all cases, name is the name of a parameter in the replacement block. In the first and third cases, the value is applied to all instances of the replacement block. In the second and fourth cases, it is applied only to the instance specified by the instance number, (which starts with 1). The second and fourth cases just introduce an optional equal sign, for readability. If the = is used, there must be spaces around it. The value can be any usual PeaCE expression for giving the value of a parameter. If this expression has spaces in it, however, then the value should appear in quotation marks so that the whole expression is kept together. If the string instance_number appears anywhere in value, it will be replaced with the instance number of the replacement block. Note that it need not be a separate token. For example, the value xxxxinstance_numberyyyy will become xxxx1yyyy for the first instance, xxxx2yyyy for the second, etc. After all appearances of the string instance_number have been replaced, value is evaluated using the usual expression evaluator for initializing the String Array states.

If you click the SrcGr block, you can see the parameter menu in which the parameter_map state is set to “frequency = {frequencyFormula}” in this example. Note that the sinegen block has the “frequency” parameter and the SrcGr block sets the parameter of each sinegen block using the right side value. The right side value is a string enclosed by {} or “ “. It refers to the state of the parent block. The top-level graph defines the “frequencyFormula” parameter which is set to “2*PI*(2*instance_number-1)/period” in this example. Note that the “period” parameter is also defined in the top level graph and set to 200 as shown in the left side of Figure 5-44. As a result, the frequency of the first sinegen block becomes 2*PI*(2*1 – 1)/200 = 0.01PI. That of the second sinegen block becomes 0.03PI.

There are some bus manipulation blocks in “library/SPDF/Control”. The Nop block is used to collect a group of individual lines as a bus line or to distribute a bus line into a group of individual lines. The BusSplit block splits a bus line into two narrower bus lines.

### 5.11 Fixed-point simulation

When you implement an application, you design the algorithm using float type variables. Later, however, you may want to change the floating type variables to the integer(or fixed-point) type variables. The Fixed-point simulation helps you determine how many bits each variable requires.
The fixed-point simulation consists of three steps: range estimation, word length optimization and fixed-point range estimation.

The range estimation is executed by “RangeEstimation” target in CGC. It monitors the values of all port and state variables during simulation and displays how many bits each variable requires in text file. For example, open “schematic/Peace/Demo/CGC/Basic/butterfly” (Figure 5-45). Change the target to “RangeEstimation”.

After run it, you can see the “range.txt” file as shown in Figure 5-46. The “range.txt” file includes statistical information of variables. Another file called wordlength.txt is produced in $HOME/PEACE_SYSTEMS/CGC/ for word length optimization in the next step.
The second step is the word length optimization step that assigns the bit width for each variable under the constraint of the given SNR(signal to noise) ratio. Now let’s set “desired SQNR” to 30.0 (Figure 5-47). The result will be saved in optimal.txt that is the “optimization report file”.

Figure 5-46 range.txt

Figure 5-47 Setting “desired SQNR” to 30.0
The last step is “FixedEstimation” that estimates the ranges of fixed-point variables computed by the previous step (Figure 5-48). The estimated ranges are saved in width.txt file. This step finalizes the range of each variable and shows the simulation result when the schematic is implemented with fixed-point type variables.

Figure 5-48 FixedEstimation target

Figure 5-49 shows how much different is the fixed-point simulation from floating point one.

Figure 5-49 (a) Result of fixed-point simulation (b) Result of floating-point simulation
Note that PeaCE do not support automatic type conversion from floating to fixed-point variables in code synthesis. We just perform fixed-point simulation for estimation and determination of bit widths of the variables. Therefore you should manually create fixed-point version of blocks or use shift blocks along with integer blocks.

5.12 Matlab simulation

The Matlab is a commercial tool for signal processing, image processing, digital/analog mixed simulation and so on. If you have Matlab installed in your computer, you can use it with the PeaCE. Currently we support Matlab version 6.

In order to use Matlab, use icons in library/SPDF/Matlab where there are Matlab and MatlabDisplay. The “Matlab” block can have multiple inputs (including zero input) and multiple outputs (including zero output) and the “MatlabDisplay” is used for displaying data. The “MatlabDisplay” block shows a Matlab graph which will be destroyed when you stroke any key on the ptcl terminal.

The “Matlab” block has “matlabScript”, “inputNames”, and “outputNames” states. In the “matlabScript” you can write Matlab script like Matlab m file. The “inputNames” and “outputNames”) indicate interface variables between block ports and matlab script. Each name of “inputNames”(“outputNames”) matches an input(output) port. For example, open a schematic of “schematic/Peace/Demo/CGC/Matlab/MatlabTest1”. You can see Figure 5-50.

As shown in Figure 5-50, you can use matlab in default-CGC target. In order to compile, link and run the schematic, you should set “compileCommand” to “mex” and “compileOptions” to “-f /usr/local/matlab6p5/bin/engopts.sh” where “/usr/local/matlab6p5” indicates matlab path.

After running the schematic, you can see the matlab graph first (Figure 5-51).
And then you are required to press the Enter key on the ptcl xterm (Figure 5-52).

Open another example of “schematic/Peace/Demo/CGC/matlab/matlabHeart” which represents multiple inputs (Figure 5-53). In this schematic, the “MatlabDisplay” are connected with Sin and Matlab blocks. The output connected with Sin block corresponds to “y1” and that with Matlab to “y2” since we create Sin block before Matlab. If we create Matlab block first then we change the connection order.
After running the schematic, you can see the following heart diagram (Figure 5-54). Remember to press a key on the ptcl terminal to close the matlab display window.
### An overview of CGC Domain demonstrations

#### 5.13.1 Basic

<table>
<thead>
<tr>
<th>Demonstration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Butterfly</td>
<td>Use sines and cosines to compute a curve known as the butterfly curve, invented by T. Fay. The curve is plotted in polar form.</td>
</tr>
<tr>
<td>Chaos</td>
<td>This is a simple demonstration of chaos, in which the phase-space plot of the famous Henon map is given.</td>
</tr>
<tr>
<td>ChaosBits</td>
<td>Chaotic Markov map with quantizer to generate random bit sequence.</td>
</tr>
<tr>
<td>CommandLine</td>
<td>This demo is a slight modification of the nonlinear demo. It uses the pragma mechanism to indicate the parameters that are to be made settable from the command-line.</td>
</tr>
<tr>
<td>Fourier_series</td>
<td>This demonstrates Fourier series. This demonstration approximates a square wave by a finite number of sinusoids.</td>
</tr>
<tr>
<td>Integrator</td>
<td>Demonstrate the features of the integrator star, such as limiting, leakage, and resetting.</td>
</tr>
<tr>
<td>Nonlinear</td>
<td>This simple system plots four nonlinear functions over the range 1.0 to 1.99. The four functions are exponential, natural logarithm, square root, and reciprocal.</td>
</tr>
<tr>
<td>PseudeRandom</td>
<td>Generate pseudo-random sequences.</td>
</tr>
<tr>
<td>Quantize</td>
<td>Demonstrate the use of the Quantizer star.</td>
</tr>
<tr>
<td>SinMod</td>
<td>Modulate a sinusoid by multiplying by another sinusoid.</td>
</tr>
<tr>
<td>SPDFGainTest</td>
<td>This demonstrates spdf Gain test using spdf model, we can run the same universe with different parameter values multiple times in a single run using piggybacking. Piggyback updates the parameter of a block at run-time.</td>
</tr>
<tr>
<td>XYPlot</td>
<td>Demonstrate the TkXYPlot star.</td>
</tr>
</tbody>
</table>
### 5.13.2 Communication

<table>
<thead>
<tr>
<th>CodeDecode</th>
<th>Encode and decode a 16-QAM signal using differential encoding for the quadrant and Gray coding for the point within the quadrant.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constellation</td>
<td>A 16-QAM signal is sent through a baseband equivalent channel that simulates the following impairments: frequency offset, phase jitter and white Gaussian noise.</td>
</tr>
<tr>
<td>DTMFCodec</td>
<td>Dual-Tone Modulated Frequency Demo. Generate touch tones and decode the based on the Goertzel Algorithm.</td>
</tr>
<tr>
<td>Eye</td>
<td>Plot an eye diagram for a binary antipodal signal with a raised-cosine pulse shape and user controlled noise.</td>
</tr>
<tr>
<td>Modem</td>
<td>Baseband model of a 16-QAM modem.</td>
</tr>
<tr>
<td>PLLDemo</td>
<td>Simulate a fourth-power optical phase-locked loop with laser phase noise and additive Gaussian white noise operating on a complex baseband envelope model of the signal.</td>
</tr>
<tr>
<td>Pulses</td>
<td>Generate raised cosine and square-root raised cosine pulses and demonstrate matched filtering with the square-root raised cosine pulse.</td>
</tr>
<tr>
<td>QAM</td>
<td>Produce a 16-point quadrature amplitude modulated(QAM) signal and displays the eye diagram for the in-phase part, the constellation, and the modulated transmitted signal.</td>
</tr>
<tr>
<td>QAM4withDFE</td>
<td>This is a model of a digital communication system that uses quadrature amplitude modulation(QAM) and a fractionally spaced decision feedback equalizer.</td>
</tr>
<tr>
<td>Xmit2Rec</td>
<td>Simple 2-level PAM communication system (matched filtering at the receiver).</td>
</tr>
<tr>
<td>Xmit4Rec</td>
<td>Simple 4-level PAM communication system (no filtering at the receiver).</td>
</tr>
</tbody>
</table>

### 5.13.3 H.263

<table>
<thead>
<tr>
<th>H263Encoder</th>
<th>This demonstrates H.263 encoder. This is designed according to Baseline profile which using I-slice and P-slice. It supports half-pel Motion Estimation, unrestricted motion vector, 4 motion vector. And it supports all mandatory and optional 3GPP codes. Picture size is QCIF: 176*144. There are AC/DC prediction.</th>
</tr>
</thead>
<tbody>
<tr>
<td>H263Decoder</td>
<td>This demonstrates H.263 decoder. This is designed according to Baseline profile which using I-slice and P-slice. It supports half-pel Motion Estimation, unrestricted motion vector, 4 motion vector. And it supports all mandatory and optional 3GPP codes. Picture size is QCIF: 176*144. There is a deblocking filter.</td>
</tr>
<tr>
<td>H263FREncoder</td>
<td>This demonstrates H.263 Fractional Rate decoder. FR reduces data memory size and long latency.</td>
</tr>
<tr>
<td>H263FRDecoder</td>
<td>This demonstrates H.263 FR decoder.</td>
</tr>
<tr>
<td>H263FRSTDecoder</td>
<td>This demonstrates H.263 decoder which reads input as streaming.</td>
</tr>
<tr>
<td>H263FRDIVx</td>
<td>This demonstrates H.263 FR decoder which reads input as streaming.</td>
</tr>
</tbody>
</table>
5.13.4 H.264

**H264Decoder**

This demonstrates H.264 decoder. This is designed according to Baseline profile which supports intra and inter-coding (using I-slice and P-slice) and entropy coding with context-adaptive variable-length codes (CAVLC). And it is single-slice mode that 1 slice is 1 frame. Picture format is QCIF:176*144. This block is optimized at Inverse Transform and Chroma Inter Prediction.

5.13.5 Image

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlendImage</td>
<td>Combine two images and display the result.</td>
</tr>
<tr>
<td>BwDither</td>
<td>Demonstrate four different forms of black and white dithering: error diffusion, clustered dither, dispersed dither, and use custom mask.</td>
</tr>
<tr>
<td>ColorImage</td>
<td>Convert an RGB(red-green-blue) format color image to YUV(luminance-hue-saturation) format and back, and then display it on the workstation screen.</td>
</tr>
<tr>
<td>CompareMedian</td>
<td>Median filter an image to reduce artifacts due to interleaved scanning in video sequences.</td>
</tr>
<tr>
<td>ContrastEnhance</td>
<td>Constrast enhancement by histogram modification.</td>
</tr>
<tr>
<td>DCTImage</td>
<td>Perform discrete cosine transform (DCT) coding of an image sequence.</td>
</tr>
<tr>
<td>DPCMImage</td>
<td>Perform differential pulse code modulation (DPCM) on an image sequence.</td>
</tr>
<tr>
<td>EdgeDetect</td>
<td>Demonstrate four different forms of edge detection: Sobel, Roberts, Presitt, and Frei-Chen.</td>
</tr>
<tr>
<td>FullVQ</td>
<td>Full search vector quantization using codebook generated by fullVQCodebk.</td>
</tr>
<tr>
<td>FullVQCodec</td>
<td>Generate a codebook for full search vector quantization.</td>
</tr>
<tr>
<td>MRVQ</td>
<td>Mean-removed vector quantization.</td>
</tr>
<tr>
<td>MRVQCodebk</td>
<td>Generate codebooks for mean-removed vector quantization using independent quantizer structure.</td>
</tr>
<tr>
<td>MRVQMeanCB</td>
<td>Generate codebook for mean-removed vector quantization.</td>
</tr>
<tr>
<td>MRVQShapeCB</td>
<td>Generate the shape codebook for mean-removed quantization using alternate structure. This universe uses the codebook generated by MRVQMeanCB.</td>
</tr>
<tr>
<td>SGVQ</td>
<td>Shape-gain vector quantization using codebook from SGVQCodebk.</td>
</tr>
<tr>
<td>SGVQCodebk</td>
<td>Generate codebooks for shape-gain vector quantization.</td>
</tr>
</tbody>
</table>

5.13.6 Matlab

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatlabHeart</td>
<td>This represents multiple inputs – sin and matlab block. After seeing the matlab graph, you should stroke keyboard on the ptcl terminal to close the graph and go ahead.</td>
</tr>
<tr>
<td>MatlabTest1</td>
<td>This is simple test demonstration which plots ( \text{abs(sin(x)) + cos(x)}. )</td>
</tr>
</tbody>
</table>
### 5.13.7 Matrix

<table>
<thead>
<tr>
<th>Star</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitDelays</td>
<td>Illustrate the use of initializable delays with the matrix class.</td>
</tr>
<tr>
<td>KalmanFilter</td>
<td>This demo show Kalman filter. The purpose of a Kalman filter is to estimate the state of a system from measurements which contain random errors.</td>
</tr>
<tr>
<td>MatrixTest1</td>
<td>Demonstrate the use of the Matrix stars that have one input. These include the operations inverse, transpose, and multiply by a scalar gain for all matrix types. Also conjugate and Hermitian transpose are available for the complex matrix type.</td>
</tr>
<tr>
<td>MatrixTest2</td>
<td>Demonstrate the use of some simple Matrix stars with two inputs. These include multiply, add, and subtract.</td>
</tr>
</tbody>
</table>

### 5.13.8 MP3

<table>
<thead>
<tr>
<th>Star</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP3</td>
<td>This demonstrates MP3 player. This is designed according to MPEG-I Layer 3 specification. It dose lossy compression and Huffman coding.</td>
</tr>
</tbody>
</table>

### 5.13.9 Multirate

<table>
<thead>
<tr>
<th>Star</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analytic</td>
<td>Use a FIRCx star filter to reduce the sample rate of a sinusoid by a factor of 8/5, and at the same time produce a complex approximately analytic signal (one that has no negative frequency components.)</td>
</tr>
<tr>
<td>CD2Dat</td>
<td>Convert two sine waves sampled at compact disc sampling rate(44.1kHz) to the digital audio tape rate(48kHz) data. The conversion is performed in multiple FIR stages. And it uses the CreateSDFStar target.</td>
</tr>
<tr>
<td>CD2DatFR</td>
<td>Convert from the compact disc sampling rate(44.1kHz) to the digital audio tape rate(48kHz) data. The conversion is performed in multiple FRFIR(Fractional Rate) stages.</td>
</tr>
<tr>
<td>DownSample</td>
<td>Convert form the digital audio tape sampling rate (48kHz) to the compact disc sampling rate (44.1kHz). The conversion is performed in multiple stages for better performance.</td>
</tr>
<tr>
<td>FilterBankNonUniform</td>
<td>Implement a simple split of the frequency domains into two non-uniform frequency bands.</td>
</tr>
<tr>
<td>Interp</td>
<td>Use an FIR filter to upsample by a factor of 8 and linearly interpolate between samples.</td>
</tr>
<tr>
<td>InterFR</td>
<td>Use an FRFIR filter to upsample by a factor of 8 and linearly interpolate between samples.</td>
</tr>
<tr>
<td>Loop</td>
<td>This demo demonstrates the code size reduction achieved with a loop-generating scheduling algorithm.</td>
</tr>
<tr>
<td>UpSample</td>
<td>This simple up-sample demo tests static buffering. Each invocation of the XMgraph star reads its input from a fixed buffer location since the buffer between the UpSample star and the XMgraph star is static.</td>
</tr>
</tbody>
</table>
### 5.13.10 Signal Processing

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptFilter</td>
<td>An LMS adaptive filter converges so that its transfer function matches that of a fixed FIR filter.</td>
</tr>
<tr>
<td>AllPole</td>
<td>Two realizations of an all-pole filter are shown to be equivalent. One uses an FIR filter in a feedback path, the other uses the BlockAllPole star.</td>
</tr>
<tr>
<td>AnimatedLMS</td>
<td>An LMS adaptive filter is configured as in the adaptFilter demo, but this time the filter taps are displayed as they adapt.</td>
</tr>
<tr>
<td>Cep</td>
<td>Given the coefficients of any polynomial, this demo uses the cepstrum to find a minimum-phase polynomial. Thus, given the coefficients of the denominator polynomial of an unstable filter, this demo will compute the coefficients of a stable denominator polynomial that has the same magnitude frequency response.</td>
</tr>
<tr>
<td>Convolve</td>
<td>Convolve two rectangular pulses in order to demonstrate the Convolve star.</td>
</tr>
<tr>
<td>DFT</td>
<td>Compute a discrete Fourier transform of a finite signal using the FFT star. The magnitude and phase(unwrapped) are plotted.</td>
</tr>
<tr>
<td>Doppler</td>
<td>A sine wave is subjected to four successive amount of Doppler shift. The Doppler shift is accomplished by the phaseShift galaxy, which forms an analytic signal(using a Hilbert transform) that modulates a complex exponential.</td>
</tr>
<tr>
<td>DTFT</td>
<td>Demonstrate the DTFT star, showing how it is different from the FFTCx star. Specifically, the range, number, and spacing of frequency samples is arbitrary.</td>
</tr>
<tr>
<td>DTMFCodec</td>
<td>Generate and decode touch tones.</td>
</tr>
<tr>
<td>FreqSample</td>
<td>This system designs FIR filters using the frequency sampling method. Samples of the frequency response are converted into FIR filter coefficients.</td>
</tr>
<tr>
<td>IIRDemo</td>
<td>Two equivalence implementations of IIR filtering. One of the implementations uses the IIR star. This demo is not present in Ptolemy0.6.</td>
</tr>
<tr>
<td>Lattice</td>
<td>Demonstrate the use of lattice filters to synthesize an auto-regressive (AR) random process.</td>
</tr>
<tr>
<td>LatticeDesign</td>
<td>Use of Levinson-Durbin algorithm to design a lattice filter with a specified transfer function.</td>
</tr>
<tr>
<td>LevinsonDurbin</td>
<td>Use of Levinson-Durbin algorithm to estimate the parameters of an AR process.</td>
</tr>
<tr>
<td>LinearPrediction</td>
<td>Perform linear prediction on a test signal consisting of three sinusoids in colored, Gaussian noise. Two mechanisms (Burg’s algorithm and an LMS adaptive filter) for linear prediction are compared.</td>
</tr>
<tr>
<td>LmsFreqDetect</td>
<td>Illustrate the use of the LMS algorithm to estimate the dominant sinusoidal frequency in the input signal.</td>
</tr>
<tr>
<td>OverLapAddFFT</td>
<td>Convolution is implemented in the frequency domain using overlap and add.</td>
</tr>
<tr>
<td>PhasedArray</td>
<td>Simulate a plane wave approaching a phased array with four sensors. The plane wave approaches from angles starting from head on and slowly rotating 360 degrees. The response of the antenna is plotted as a function of direction of arrival in polar form.</td>
</tr>
<tr>
<td>PhasedArray_tk</td>
<td>This simulates a plane wave approaching a phased array of arbitrary positioned sensors. This demonstrates TkSlider by creating a vertical array of radar sensors that can be move in the horizontal plane. Note that small calibration to performance of the sensor array. This simulation demonstrates the importance of sensor calibration to performance of the sensor array.</td>
</tr>
</tbody>
</table>
### PowerSpectrum

Compare three methods for estimating a power spectrum of a signal with three sinusoids plus colored noise. The three methods are the periodogram method, the autocorrelation method, and Bug’s method.

### Window

Generate and display four window functions and the magnitude of their Fourier transforms. The windows displayed are the Hanning, Hamming, Blackman, and steep Blackman.

### 5.13.11 SPDF

<table>
<thead>
<tr>
<th>For-if-test</th>
<th>This is a simple demonstration of cascaded for-if demonstration using piggyback. Set the state[conName]=for of ifgal galzxy.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForTest</td>
<td>This is a simple demonstration of for construction. Set the galaxy state[type]=for. DownCounter star produces varying number of tokens.</td>
</tr>
<tr>
<td>IfTest</td>
<td>This is a simple if-then-else demonstration. Set the galaxy states:type=if, conName=xxx, conValue=yyy. And created a universe state xxx for piggybacking.</td>
</tr>
<tr>
<td>Picture</td>
<td>This generates a random walk with evenly spaced points.</td>
</tr>
<tr>
<td>SPDFGainTest</td>
<td>This demonstrates spdf Gain test using spdf model, we can run the same universe with different parameter values multiple times in a single run using piggybacking. Piggyback updates the parameter of a block at run-time.</td>
</tr>
<tr>
<td>Timing</td>
<td>This demonstrates ASK receiver with approximate MMSE timing recovery.</td>
</tr>
</tbody>
</table>
Chapter 6. SPDF Model : VHDL Code Generation

Author: Hyunuk Jung, Hoeseok Yang, and Soonhoi Ha

This chapter explains the hardware code generation from the SPDF task graph. The generated hardware code from PeaCE is RTL (Register Transfer Level) description in VHDL. So you can simulate and synthesize the generated code.

It is relatively easy to generate hardware code just for simulation from a computation model. But it is a very different story to generate hardware code for synthesis, especially to generate an efficient hardware in terms of area and performance. We have tried to generate an efficient hardware code so that the result can be applied to practical design. Although there still remain some overheads compared with efficient manual design, we believe that this tool will be very useful for fast prototyping of applications that are not performance or area critical.

This tool does not provide everything for free. You should start from efficient library blocks. We will start our explanation with a simple example using predefined library blocks. Then, we explain how to make your own library block (VHDLStar). Later you may catch the advanced features through some examples. FRDF (Fractional Rate Dataflow) model and memory access interface may help you to design efficient library blocks in real cases.

6.1 Before You Begin

Before reading this chapter, you are assumed to be familiar with C code generation in PeaCE. You should also be familiar with defining a block for C code generation if you want to make your own block for VHDL code generation. Defining a block for VHDL Code Generation (VHDLStar) also requires VHDL coding skill.

In order to simulate the generated VHDL code, ModelSim VHDL simulator should be installed in the machine in which PeaCE is installed. To use ModelSim, please make sure that the ModelSim commands such as vlib, vmap, vcom, and vsim are in your path.

6.2 Limitations

- Piggyback is not supported yet.
- Dynamic constructs are not supported yet.
- Supported data types are only integer, fixed, and message.
- In fact, the float data type should be converted to the fixed-point in the generated HDL. So, you should specify the bit width of each port. Otherwise, the default bit width would be applied.
6.3 Using VHDL Domain with a Simple Example

Following is a simple example in VHDL domain using predefined blocks. Through this example you can understand how to make a simple application and run it.

6.3.1 Create a new schematic file

First, create a new schematic file, “simple”, and select the VHDL domain (Figure 6-1). Then the ModelSim-VHDL target will be automatically selected since ModelSim-VHDL is the only supported target in VHDL domain. You can see target parameters listed in the parameter-window and may change the parameter values (Table 6-1). However, you do not have to change the values for now. The default values are sufficient.

![New schematic file creation](image)
### Table 6-1 Target parameters of ModelSim-VHDL Target

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>NULL</td>
<td>Host machine to compile or assemble code on. Default is the server machine.</td>
</tr>
<tr>
<td>Directory</td>
<td>PEACE_SYSTEMS/CGC</td>
<td>Where to save the generated code</td>
</tr>
<tr>
<td>display?</td>
<td>YES</td>
<td>Display the generated code if YES</td>
</tr>
<tr>
<td>run?</td>
<td>YES</td>
<td>Run the compiled code if YES</td>
</tr>
<tr>
<td>write schedule?</td>
<td>NO</td>
<td>Write the schedule of the given block diagram if YES</td>
</tr>
<tr>
<td>Address Width</td>
<td>32</td>
<td>The bit width of address signal in case memory access is needed.</td>
</tr>
<tr>
<td>Clock Period(ns)</td>
<td>20</td>
<td>HW clock cycle period : 20 ns =&gt; 50Mhz</td>
</tr>
<tr>
<td>LibAnalyze</td>
<td>YES</td>
<td>Compile(analyze) the library code if YES</td>
</tr>
<tr>
<td>Analyze</td>
<td>YES</td>
<td>Compile(analyze) the generated code if YES</td>
</tr>
<tr>
<td>Width Decision Mode</td>
<td>DEFAULT</td>
<td>DEFAULT : width is set by user in HAE FILE : width is read from FILE</td>
</tr>
<tr>
<td>Default Width (bit)</td>
<td>32</td>
<td>Default signal bit width</td>
</tr>
<tr>
<td>Default IWL</td>
<td>15</td>
<td>Default integer word length of signal</td>
</tr>
<tr>
<td>Width File</td>
<td>$HOME/PEACE_SYSTEMS/CGC/width.txt</td>
<td>Width file path in case the width decision mode is FILE.</td>
</tr>
<tr>
<td>Resource File</td>
<td>$HOME/PEACE_SYSTEMS/VHDL/resource.txt</td>
<td>Resource and Schedule File : User must make this file for VHDL stand-alone application. In case of cosimulation and cosynthesis, this file is automatically generated from sched.xml file</td>
</tr>
<tr>
<td>Interactive mode</td>
<td>NO</td>
<td>NO : Modelsim is invoked and run automatically in command mode. YES : Modelsim is invoked in window mode and waits for user input. You can view waveforms and signal values in this mode.</td>
</tr>
</tbody>
</table>

#### 6.3.2 Draw an SPDF graph with library blocks

As a simple example, we will make a function, \( y = \text{abs} (\sin(x)) \).

In order to implement the function, we need the following library blocks:

- \( x \) value generator : Ramp,
- Sin function : Sin,
- Abs function : Abs,
- Result Viewer : Xgraph.
All these blocks are already implemented and you can find these predefined blocks in the library tab. First, click the library tab, then, the library browse window will appear. And select SPDF/src directory to find *Ramp* block (Figure 6-2). In this example, you should select Ramp block with float type. The float type of port is represented as blue color.

Remember that the port color represents the data type as follows.

- ANYTYPE : red
- FLOAT : blue (implemented into fixed-point type only in VHDL domain)
- INT : orange
- MESSAGE : green
- COMPLEX : white (not supported in VHDL)
- FIX : violet (not supported in VHDL)

![Figure 6-2 Select Ramp block in Src directory](image)
And you can find \textit{Sin} and \textit{Abs} block in \textit{SPDF/Math} directory (Figure 6-3) and the color of their ports are also blue.

At last select \textit{Xgraph} block in \textit{SPDF/Sink} directory (Figure 6-4). (Xgraph VHDL block is not synthesizable library. It is used just for simulation.) You also have to draw arcs between all the blocks to complete the graph.
6.3.3 Set block parameters

Now you have to specify the parameter values of blocks. To set the parameters, click the design tab.

- Design tab: setting the parameters of target and blocks
- Lib tab: browsing the library blocks and adding new blocks

First select the Ramp star (Figure 6-5).

![Design tab](image)

**Figure 6-5 Setting the Ramp block states**

The Ramp star has two main states. One is value, and the other is step. Step is a constant and value increases by step at each execution. For example if step is 1 and the initial value of value is 0, the output will be 0, 1, 2, 3, … , and so forth.

Since the generated VHDL code uses fixed-point representation, you should also specify the signal width and radix point. Usually you can set the values of WIDTH and IWL that mean signal width and integer word length respectively. Since every data signal has one sign bit, the total signal width is IWL + FWL (fractional word length) + 1. When you assign 10 to WIDTH and 5 to IWL, the value of FWL becomes 4.
The parameters of \textit{Sin} block are more complicated (Figure 6-6). Since the output signal of \textit{Ramp} is to be transferred into the input of \textit{Sin}, you’d better match the width and the radix point for direct connection, even though the VHDL code generator extends the sign-bit and pads zero bits for unmatched signals automatically. As shown in the previous figure, the width of \textit{Ramp} output signal is 10 and the IWL is 5. This means the FWL is 4. So, the InputWidth of \textit{Sin} will be 10 and InputFWL 4. The \textit{Sin} block consists of an input multiplier and a ROM table. The size of ROM table is \(2^{\text{indexWidth}} \times \text{OutputWidth}\). In other word, IndexWidth is the address width of ROM and OutputWidth is the width of an entry of ROM. So the precision of output value depends on these two parameters. In this example, 8 is assigned to both parameters. And the IWL of the \textit{Sin} output signal is always 1 since the range is always between 1 and –1. Since \textit{Xgraph} automatically calculates the width and the radix point, you do not have to specify its parameters.

### 6.3.4 Make a resource File

VHDL example needs a resource file. This resource file contains the information of resource allocation, mapping and schedule.

You should specify the path of the resource file after selecting the background first (Figure 6-7). When no block is selected (select background), the target parameters are listed in the left parameter-window. Among these parameters, you can find \textit{ResourceFile}, which represents the path of the resource file. In the server machine, you should create and edit this file using text editor such as \textit{vi-editor}. In this example, the file path is set to \$HOME/simple.rs.
In the resource file, you should write the number of each resource using `setresourcenum` command. After that, the schedule information is specified using `resmap` command and instance names. You can get the instance name of each block by selecting `view/name` option after saving this schematic file. (In case schematic has not saved, the names do not have the correct value.) In this example every block is executed in one cycle (Figure 6-8).

![Showing star instance names](image)

**Figure 6-7 Setting a resource file path**

```
# resource table
setresourcenum Ramp 1
setresourcenum Sin 1
setresourcenum Abs 1
setresourcenum Xgraph 1

# resource mapping & schedule information
# (starname, resource number, start, duration)
resmap Ramp10 0 0 1
resmap SinI5 0 1 1
resmap AbsI2 0 2 1
resmap XgraphI8 0 3 1
```

**Figure 6-8 Resource file**
6.3.5 Run the example

When you have done everything explained above, you can run this example. Click the run button, and you can see run panel popped up. After write iteration number in the run panel (in this example, 100), click the go button. A VHDL code will be generated and ModelSim will compile the code and begin to simulate it. After the simulation finishes, the Xgraph will show you the simulation result.

Figure 6-9 Run the example
6.4 VHDL Block Definition

VHDL domain automatically generates RTL-level hardware description in VHDL from the SPDF block diagram. It generates the controller of the application and the glue logics between blocks and controller based on the schedule information and the graph topology. But each block should be described manually in VHDL. Let us look inside a block in VHDL domain. As a simple example, the RampInt block is described below.

defstar {
    name { RampInt }
    domain { VHDL }
    desc { Output the sum of the inputs, as a integer value. }
    version { @(#)VHDLRampInt.pl 1.4 05/29/00 }
    author { Hyunuk Jung }
    location { VHDL main library }
    output { name { output } type { int } }
    state { name { step } type { int } default { 1 }
        desc { Increment from one sample to the next }
        state { name { value } type { int } default { 0 }
            desc { Initial value output by Ramp. }
            attributes { A_SETTABLE | A_NONCONSTANT }
        }
        state { name { WIDTH } type { int } default { 8 }
            desc { bit width of output and value signal of Ramp }
            attributes { A_GENERIC | A_WIDTHVARIABLE }
        }
    }
    begin {
        setWidthIWL(output, "$WIDTH", "$WIDTH - 1");
        setWidthIWL(value, "$WIDTH", "$WIDTH - 1");
        setWidthIWL(step, "$WIDTH", "$WIDTH - 1", PST_PASSIVE);
    }
    go { addCode(main); }
    codeblock (main) {
        output <= value_in;
        value_out <= value_in + step_in;
    }
}
This VHDLRampInt.pl file is very similar to CGCRampInt.pl file in CGC domain. If you are familiar with “.pl” file, you can easily define a hardware library block. In the simple case, you only have to
1. change the domain name (CGC → VHDL),
2. specify the width of ports and states
3. and describe the code of main body in VHDL.

Difference between CGCRampInt.pl and VHDLRampInt.pl is highlighted below.

<table>
<thead>
<tr>
<th>VHDLRampInt.pl</th>
<th>CGCRampInt.pl</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong> { RampInt }</td>
<td>name { RampInt }</td>
</tr>
<tr>
<td><strong>domain</strong> { VHDL }</td>
<td>domain { CGC }</td>
</tr>
<tr>
<td><strong>location</strong> { VHDL main library }</td>
<td>location { CGC main library }</td>
</tr>
<tr>
<td><strong>output</strong> { name { output } type { int } }</td>
<td><strong>output</strong> { name { output } type { int } }</td>
</tr>
<tr>
<td><strong>state</strong> { name { step } type { int } default { 1 } }</td>
<td><strong>state</strong> { name { step } type { int } default { 1 } }</td>
</tr>
<tr>
<td>desc { Increment from one sample to the next }</td>
<td>desc { Increment from one sample to the next }</td>
</tr>
<tr>
<td><strong>state</strong> { name { value } type { int } default { 0 } }</td>
<td><strong>state</strong> { name { value } type { int } default { 0 } }</td>
</tr>
<tr>
<td>desc { Initial value output by Ramp. }</td>
<td>desc { Initial value output by Ramp. }</td>
</tr>
<tr>
<td>attributes { A_SETTABLE</td>
<td>A_NONCONSTANT }</td>
</tr>
<tr>
<td><strong>state</strong> { name { WIDTH} type {int} default {8} }</td>
<td><strong>state</strong> { name { WIDTH} type {int} default {8} }</td>
</tr>
<tr>
<td>desc { bit width of output and value signal of Ramp }</td>
<td>desc { bit width of output and value signal of Ramp }</td>
</tr>
<tr>
<td>attributes { A_GENERIC</td>
<td>A_WIDTHVARIABLE }</td>
</tr>
<tr>
<td><strong>begin</strong> {</td>
<td><strong>begin</strong> {</td>
</tr>
<tr>
<td>setWidthIWL(output, &quot;$WIDTH&quot;, &quot;$WIDTH-1&quot;);</td>
<td>setWidthIWL(output, &quot;$WIDTH&quot;, &quot;$WIDTH-1&quot;);</td>
</tr>
<tr>
<td>setWidthIWL(value, &quot;$WIDTH&quot;, &quot;$WIDTH-1&quot;);</td>
<td>setWidthIWL(value, &quot;$WIDTH&quot;, &quot;$WIDTH-1&quot;);</td>
</tr>
<tr>
<td>setWidthIWL(step, &quot;$WIDTH&quot;, &quot;$WIDTH - 1&quot;, PST_PASSIVE);</td>
<td>setWidthIWL(step, &quot;$WIDTH&quot;, &quot;$WIDTH - 1&quot;, PST_PASSIVE);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td><strong>go</strong> { addCode(main); }</td>
<td><strong>go</strong> { addCode(std); }</td>
</tr>
<tr>
<td><strong>codeblock</strong> (main) {</td>
<td><strong>codeblock</strong> (std) {</td>
</tr>
<tr>
<td>output &lt;= value_in;</td>
<td>$ref(output) = $ref(value);</td>
</tr>
<tr>
<td>value_out &lt;= value_in + step_in;</td>
<td>$ref(value) += $val(step);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
6.4.1 Block definition and the generated code: RampInt

entity RampInt is
  generic (
    WIDTH : INTEGER
  );
  port (
    output : OUT STD_LOGIC_VECTOR(WIDTH -1 downto 0);
    step_in : in std_logic_vector(WIDTH - 1 downto 0);
    value_in : in std_logic_vector(WIDTH - 1 downto 0);
    value_out : out std_logic_vector(WIDTH - 1 downto 0)
  );
end RampInt;

architecture behavior of RampInt is
begin
  output <= value_in;
  value_out <= value_in + step_in;
end behavior;

This generated VHDL code is a structural description in RTL-level and a library block is mapped to an entity and an architecture description in the generated VHDL code.

→ Generated VHDL code structure
entity description
  generic declarations
  port declarations
  architecture descriptions
  declarations part
  architecture body

entity name ← name of the block (star)
generic declarations ← states which have A_GENERIC attribute
input port declarations ← inputs and non-HIDDEN states
output port declarations ← outputs and non-HIDDEN-and-NONCONSTANT states
declarations in architecture ← addDeclaration() in initCode{ }
architecture body ← addCode() in go{ }

HIDDEN states : parameters for code generation at compile-time
Non-HIDDEN states: variables or registers in the generated code

You can see how the generated code is related with the block definition below:

<table>
<thead>
<tr>
<th>VHDL Star File: &quot;pl&quot; file</th>
<th>Generated VHDL code: entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>defstar {</td>
<td></td>
</tr>
<tr>
<td>name { Block1 }</td>
<td>entity Block1 is</td>
</tr>
<tr>
<td>domain { VHDL }</td>
<td>port (</td>
</tr>
<tr>
<td>input { name {input} type {float} }</td>
<td>output : out std_logic_vector(15 downto 0);</td>
</tr>
<tr>
<td>output { name {output} type {float} }</td>
<td>input : in std_logic_vector(15 downto 0);</td>
</tr>
<tr>
<td>initCode { addDeclaration(decl); }</td>
<td>);</td>
</tr>
<tr>
<td>go { addCode(main); }</td>
<td>end Block1;</td>
</tr>
<tr>
<td>codeblock(decl) {</td>
<td>architecture behavior of Block1 is</td>
</tr>
<tr>
<td>declarations....</td>
<td>declarations....</td>
</tr>
<tr>
<td>}</td>
<td>begin</td>
</tr>
<tr>
<td>codeblock(main) {</td>
<td>main codes....</td>
</tr>
<tr>
<td>main codes....</td>
<td>end behavior;</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

### 6.4.2 Setting the width of ports and states

setWidthIWL() function in begin{} block

setWidthIWL(a port or a state, width value, IWL value)

IWL: integer word length used to adjust the radix points between signals in fixed-points implementations

<table>
<thead>
<tr>
<th>VHDL Star Example File: &quot;pl&quot; file</th>
<th>Generated VHDL code: entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>defstar {</td>
<td>entity Block1 is</td>
</tr>
<tr>
<td>name { Block1 }</td>
<td>port (</td>
</tr>
<tr>
<td>domain { VHDL }</td>
<td>output : out std_logic_vector(7 downto 0);</td>
</tr>
<tr>
<td>input { name {input} type {float} }</td>
<td>input : in std_logic_vector(7 downto 0);</td>
</tr>
<tr>
<td>output { name {output} type {float} }</td>
<td>);</td>
</tr>
<tr>
<td>begin {</td>
<td>end Block1;</td>
</tr>
<tr>
<td>setWidthIWL(input, 8, 7);</td>
<td>architecture behavior of Block1 is</td>
</tr>
<tr>
<td>setWidthIWL(output, 8, 7);</td>
<td>declarations....</td>
</tr>
<tr>
<td>}</td>
<td>begin</td>
</tr>
<tr>
<td>.....</td>
<td>main codes....</td>
</tr>
<tr>
<td>}</td>
<td>end behavior;</td>
</tr>
</tbody>
</table>
Width & IWL

- **Width**: total signal width
- **IWL**: Integer word length
- **FWL**: Fractional word length
- **Width** = **IWL** + **FWL** + 1 (sign bit)

![Diagram showing width and integer/ fractional word lengths]

**Width Setting by generic variable**

```vhdl
defstate {
    name { WIDTH } type { int } default { 8 }
    attributes { A_GENERIC | A_WIDTHVARIABLE }
}
begin {
    setWidthIWL(input, "$WIDTH", "$WIDTH-1");
    setWidthIWL(output, "$WIDTH", "$WIDTH-1");
}
```

**entity Block1 is**

generic ( WIDTH : integer );
port (  
    output : out std_logic_vector(WIDTH-1 downto 0);
    input : in std_logic_vector(WIDTH-1 downto 0);
);
end Block1;

**Width can be set by**

1. **constant value**
   - `setWidthIWL(input, 8, 7);`
2. **a width variable**: a state of which attribute is **A_WIDTHVARIABLE**
   - `setWidthIWL(input, WIDTH, IWL);`
   ```vhdl
   defstate {
       name { WIDTH } type { int } default { 8 }
       attributes { A_GENERIC | A_WIDTHVARIABLE }
   }
   ```
3. **expression** (constant value, width variable and operations `<+,-,*>`)  
   - `setWidthIWL(input, "$WIDTH", "$WIDTH-1");`
   
   `expression`: a string that can be evaluated later
### 6.4.3 Type of block

We classify the blocks into four types as follows:
- **A**: Combinational logic
- **B**: Single-cycle sequential logic
- **C**: Multi-cycle sequential logic with fixed execution time
- **D**: Multi-cycle sequential logic with variable execution time.

<table>
<thead>
<tr>
<th>Type A: Combinational logic</th>
<th>Type B: Single-Cycle sequential logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHDL Star with NO Internal States</td>
<td>VHDL Star with Internal States</td>
</tr>
</tbody>
</table>

This logic is separated into combinational logic and state and implemented as Mealy machine.

![Block Implementation of type A and B](image)

**Figure 6-10 Block Implementation of type A and B**

Depending on the block types, we create different glue logics between blocks and control logics associated with the blocks.

A combinational logic can be connected directly in hardware implementation. The input signals are propagated to the output ports naturally. The central controller only have to consider its execution time or propagation delay in order to decide the execution timing of other nodes.

For implementing a single-cycle sequential logic, we separate it into a combinational logic and state registers. This implementation is like a Mealy machine. Clock and reset signal should be provided to this register and the central controller should provide the enable signal in order to latch the next value. This enable signal is generated when the execution of this block finishes.
In the third case of multi-cycle sequential logic with fixed execution time, we should provide **start** signal. In this case, the state registers are located in the code body of the block. The start signal is generated at the scheduled time of the node by the central controller. Since we know the execution time of the node, we can generate the output update signal to latch the output after its execution. The last case deals with a block with variable execution time. Then the library block should inform the controller when to latch the output signal using “done” signal.

The type of each block is decided by a block-writer. He or she must specify its internal states and necessary signals such as “start” or “done”. In case “start” signal is not specified, the type of a block is A or B, otherwise C or D. If a block has neither “start” signal nor internal state, its type is A. Following figure explains this type decision scenario easily.
To declare having no internal state.
constructor {     noInternalState();     }

To declare start & done signal
initCode {
    declarePort("start",VHDL_START,VHDL_STD_LOGIC,VHDL_INPUT);
    declarePort("done,VHDL_DONE, VHDL_STD_LOGIC,VHDL_OUTPUT);
}

6.4.4 Timing Model of a HW Library Block

Dataflow semantic assumes the strict execution while a hardware block may react to any event arrival (or value change) in its ports. To preserve the SPDF execution semantics of the generate hardware, we enforce that the generated HW block can start its execution after all its inputs are valid and finish its execution after all its outputs are valid. Then we may use the principle of “correct-by-construction”.

In addition PeaCE has some restrictions during VHDL code generation. It assumes that a library block can start its execution only after previous execution ends. – a library block can finish its execution only after all its outputs are valid and all its internal operations end. – strict execution including internal states. So the execution length of a block equals to (earliest start time of successive block – its start time). Automatic pipelining is not supported in this release.
6.5 Scheduling and Code Generation for HW Synthesis

6.5.1 Schedule Information File

In VHDL domain, no scheduler is executed to determine the execution order of blocks. Instead it takes as an input a schedule information file that has been generated automatically or manually. If you want to draw and execute a new application, you should make its own schedule information file. In case of cosimulation or cosynthesis, this schedule information file is automatically generated from *sched.xml* using *xml2rs* tool.

For the “Butterfly” example, resource allocation, a schedule information, and the schedule information file is shown in Figure 6-12, Figure 6-13, and Figure 6-14 respectively.

![Figure 6-12 Resource allocation & binding (Butterfly Example)](image1)

![Figure 6-13 Timing diagram of the schedule of Butterfly example](image2)
In this scheduling example, 2 resources are used for “Gain” block and the other blocks have only one resource each. As a result, the resources of MCos, Multiplier are shared. In the schedule information file, each block should have its resource mapping, start timing, and execution time. The time unit is clock cycle. When you draw a new application, you cannot know the name of each star. In this case, you can run the application and HAE generates the “pt_temp.pt” file that contains the ptcl script. You can get the name of each block from this file. Using these names, make the schedule information file, and run the application again. During the codesign process, the schedule information file will be automatically generated in the partitioning step.

The schedule information consists of two parts. One is resource allocation table and the other is mapping/schedule information. Resource allocation table is simply a set of pairs, {resource type name, number of its resources}. It tells the number of allocated resources. Mapping/schedule information defines the timing information of each instance of nodes. It also defines the allocated resource to the instances. It may be grouped by a loop hierarchically. Following BNF represents the structure more concisely.

```plaintext
<schedule information> ::=<allocation table><mapping_schedule information>
<allocation table> ::= set of <allocation item>
<allocation item> ::= <resource type name> <number of resources>
<mapping_schedule information> ::= set of <mapping_schedule item>
<mapping_schedule item> ::= 
```

Figure 6-14 Schedule Information File : Butterfly Example
In this version, pipelining is not implemented. So the pipeline stage field in loop specification should be always set to 1.

### 6.5.2 Generated VHDL code structure

Generated VHDL code is divided into two parts.

1. Library block code
2. Top-level block code

Figure 6-15 shows a simple example and its generated code.

Top-level block code is divided into three parts.
1. Top module entity declaration
2. Top module architecture body
3. Top module configuration

Top module architecture body is divided into two parts
6.6 Running a simple Demo: Butterfly_ModelSim

To run a VHDL demo, ModelSim VHDL simulator should be installed in the server. Other VHDL simulators are not yet supported. If ModelSim is installed in your server, you can compile and run the generated VHDL code. Otherwise, you can only generate a VHDL code.

1. Open a Demo schematic file in HAE
   ./schematic/PeaCE/Demo/VHDL/Butterfly_ModelSim (Figure 6-16)

2. Run this Demo (Figure 6-17)
   By default, VHDL simulator is executed in the command mode. If you want to view wave signals, turn on the option, Interactive mode, in the target parameter list. This will make the simulator executed in window mode and you can type any command you want.
If we run the demo, a VHDL code will be generated and compiled. We invoke the ModelSim simulator and simulate it. Even though you may use other VHDL simulators, the current version is assumed to use the ModelSim.

(3) Simulation result
In this Demo, the simulation result is shown on the Xgraph window (Figure 6-18). In fact, the result of running C-code is more accurate. Fixed-point implementation of VHDL code shows some quantization noise in the output.
The generated codes are located in PEACE_SYSTEMS/VHDL that is set by “directory” target parameter. Generated Codes are

- Butterfly_ModelSim.vhd : main vhdl file
- Butterfly_ModelSim_SIM.vhd : vhdl file for simulation
- Butterfly_ModelSim.do : simulation script for ModelSim

6.7 Running a Demo : JPEG Encoder

This example is a JPEG encoder. In this specification, we used message data type, which is implemented as struct type in c code and record type in VHDL code. You can see the green ports with message type in Figure 6-19. In JPEG example 8x8 matrix or array of 64 integers is used as a data sample with message type. For simple implementation and simulation, we used FLI-interface and C code to build ReadBMP and HuffmanEncode library block. Therefore, they cannot be used for synthesis, they are only for simulation. Except them, all other blocks are described in RTL and so synthesizable.

![Figure 6-19 JPEG encoder SPDF representation in PeaCE](image)

Figure 6-19 specifies the JPEG Encoding algorithm. Each arc between blocks carries 8x8 matrix data sample. The function of each block is described below

- ReadBMPBlock : Read 8x8 matrix from the specified file and output the data samples
- PreShiftBlock : subtract 128 from each value of 8x8 matrix
- ChenDCT : Chen Discrete Cosine Transform algorithm for 8x8 matrix
- BoundDCT : limit the output value between –1023 and 1023
- JPGQuantizeBlock : divide each value of input matrix by the Quantize matrix
In this section, we will give you two JPEG encoder examples. One is the JPEG encoder algorithm using 2-dimensional ChenDCT block, which acts as 8x8 block DCT transformer. (section 6.7.1) In another version, we will decompose ChenDCT block into 2 instances of 1-dimensional DCT block and illustrate resource sharing (section 6.7.2.2) and using multiple resources (section 6.7.2.3). (section 6.7.2)

All blocks used in this examples are in block library of Hae.(Lib→Application→JPEG)

### 6.7.1 JPEG Encoder using ChenDCT

- Create a new schematic file, “jpeg” (or your desired name) using “new” button and select VHDL domain, ModelSim-VHDL target.

![Image of starting JPEG Example with new VHDL schematic](image)

- Go to the Library tab and select the appropriate blocks.
Figure 6-21 Getting library blocks of JPEG example

- Draw jpeg encoder SPDF representation, as below

Figure 6-22 Completed graph of JPEG encoder algorithm
- For writing the schedule file, check out the star instance names by setting “Name” field of “View” tab. (You should save this schematic file before checking out the star instance name.)

![Image of schematic file showing star instance names]

**Figure 6-23 Viewing Each Instance Name for Editing Schedule(rs) File**

- Write the schedule files as below. (The lines start with ‘#’ are treated as comment.)

```
# resource table
# setresourcenum [star_name] [number_of_resources]
setresourcenum ReadBMPBlock 1
setresourcenum PreShiftBlock 1
setresourcenum ChenDCT 1
setresourcenum BoundDCT 1
setresourcenum JPGQuantizeBlock 1
setresourcenum ZigzagBlock 1
setresourcenum HuffmanEncodeBlock 1

# resource mapping & schedule information
# resmap [instance_name] [resource_id] [start_time] [duration]
resmap ReadBMPBlock0 0 3 2  
resmap PreShiftBlock2 0 5 1  
resmap ChenDCT15 0 6 1000  
resmap BoundDCT18 0 1006 1  
resmap JPGQuantizeBlock11 0 1007 600  
resmap ZigzagBlock16 0 1607 1  
resmap HuffmanEncodeBlock14 0 1608 5
```

**Figure 6-24 An Example of Schedule File of JPEG Encoder**
• Set the destination directory and the schedule file path.

Go back to the design tab. Set destination directory and the schedule file path.

Figure 6-25 Setting Target Parameters: Destination Directory and Schedule File Path

• setting parameters
  ✓ In ReadBMPBlock, you should set width, height, and inputFile.
  Click ReadBMPBlock block, and go to the design tab. Then you will see three parameters (width, height, inputFile). Insert appropriate parameter value to each blank.
  (In our case, width, height, inputFile are “200”, “150”, “$PEACE/lib/demo/hw/lee.bmp”, respectively.)

  ✓ In HuffmanEncodeBlock, you should set width, height, outputFile.
  Click HuffmanEncodeBlock block, and go to the design tab. Then you will see three parameters (width, height, outputFile). Insert appropriate parameter value to each blank.
  (In our case, width, height, outputFile are “200”, “150”, “lee.jpg(or your desired name)”, respectively.)
Set parameters

Figure 6-26 Setting Block Parameters : Width, Height and File Name

- Set the default “Run Count” Value

Now, let’s set the default “Run Count” value. In our SPDF graph, a 8x8 block is processed in an iteration. So we must execute the SPDF graph many times to process a full size image. By the given width and height value, we can compute the number of iterations enough to process full image.

In this example, we treat 200x150 image. So, the number of 8x8 blocks in this image is

\[
\left\lceil \frac{\text{width}}{8} \right\rceil \times \left\lceil \frac{\text{height}}{8} \right\rceil = \left\lceil \frac{200}{8} \right\rceil \times \left\lceil \frac{150}{8} \right\rceil = 25 \times 19 = 475
\]

Now, go to “Edit Run Count” field in “Run” tab. And set the default “Run Count”.

151
Check out the result.

Now, check out the result. Click ‘Run’ icon.

Then, you can see the run panel, as shown below.
click ‘GO’! Then, you can get the generated code, and output image. pctl calls external VHDL simulator (we use ModelSim for this) and fetches the simulated result. (It will take a few seconds for 200x150 image)
6.7.2 JPEG Encoder using 1-dimensional DCT

In 6.7.1, we illustrate a JPEG Encoder example using 2-dimensional Chen DCT block. In this section, we will illustrate another JPEG Encoder example whose Chen DCT part is now decomposed into 2 one-dimensional DCT instances. We think that it is a good example to explain resource sharing and multiple-resources instantiation for one instance.

Let's draw an SPDF graph as shown in Figure 6-31. (You can also get this schematic file from Hae\schematic\Peace\Demo\VHDL) You can get all needed blocks from Library/Application/JPEG.

![Figure 6-31 2-Dimensional DCT algorithm specification using 1-Dimensional DCT Blocks](image)

The highlighted box represents the subgraph corresponding to the ChenDCT block. The functionality of each block is described below.

- **Block2Int**: decompose 8x8 block to 64 integers
- **DCT**: DCT 8 integers
- **TransposeMem**: transpose 8x8 block
- **Int2Block**: compose 64 integers to 1 8x8 block

Block2Int gets a data sample of 8x8 block as input and decomposes it into 64 integer data samples. This block outputs 64 integers into the DCT block. The DCT block consumes 8 integers in order. So the first DCT block fires (or is executed) 8 times per iteration and each firing handles 1-row of 8x8 block. Then the TransposMem block permutes the sequence of 8x8 block. So the following DCT block does 1-column DCT of 8x8 block at each firing. After 8-times firing of the DCT block, the TransposeMem block restores the sequence of 8x8 block. Int2Block packages 64 integers (output of TransposeMem) to 8x8 block.

Pay attention to the rates attached to the ports of each star. They represent the number of samples produced or consumed per one invocation of each block. So, the number of firing of each block is not identical. (It is contrary to previous jpeg encoder representation.). The number of firings of the blocks becomes
Now, we will treat three examples that have the same schematic but different schedule(resource) files.

- **Instance-resource 1:1 mapping**: each instance of DCT have a resource (6.7.2.1)
- **Resource sharing**: 2 instances of DCT share a resource (6.7.2.2)
- **Multiple resource**: an instance of DCT have two resources (6.7.2.3)

### 6.7.2.1 Instance-resource 1:1 mapping

Figure 6-32 shows an example of resource file of JPEG Encoder using 1-dimensional DCT

```markdown
# resource table
# setresourcenum [star_name] [number_of_resources]
setresourcenum ReadBMPBlock 1
setresourcenum PreShiftBlock 1
setresourcenum Block2Int 1
setresourcenum DCT 2
setresourcenum TransposeMem 2
setresourcenum Int2Block 1
setresourcenum BoundDCT 1
setresourcenum JPGQuantizeBlock 1
setresourcenum ZigzagBlock 1
setresourcenum HuffmanEncodeBlock 1

# resource mapping & schedule information
# resmap [instance_name] [resource_id] [start_time] [duration]
# loop [number_of_iteration] [start_time] [duration] [pipeline_info]
resmap ReadBMPBlockI0 0 3 2
resmap PreShiftBlockI2 0 5 1
resmap Block2IntI28 0 6 1
loop 8 7 10 1 {
  resmap DCTI19 0 10
}
resmap TransposeMemI16 0 87 1
loop 8 88 10 1 {
  resmap DCTI22 1 10
}
resmap TransposeMemI25 1 168 1
resmap Int2BlockI31 0 169 1
resmap BoundDCTI18 0 170 1
resmap JPGQuantizeBlockI11 0 171 600
resmap ZigzagBlockI16 0 771 1
resmap HuffmanEncodeBlockI14 0 772 5
```

Figure 6-32 Schedule example : 1-to-1 mapping of DCT Block

You can see that the `loop` command in this file is used to express the number of firings of a block or a set of blocks. Pay special attention to the “resource_id” field which is highlighted by green color. 2 DCT instances have separate resources. And look at the first argument of loop (It is the number of iteration). Since Block2Int and
TransposeMem produce 64 integers as output, the subsequent DCT blocks should be executed 8 times. Therefore the number of iteration becomes 8.

If you have finished writing the schedule file, set the schedule file path on the design tab. (You can get the same resource file at $PEACE/lib/demo/hw/jpeg_2d_dct.rs) And execute it as the same way to execute previous JPEG encoder example. You should get the same result. Figure 6-33 is a snapshot of generated VHDL code for above instance-resource 1-to-1 mapping example. This is the part of instance-resource mapping.

![Figure 6-33 Generated VHDL code: 1-to-1 mapping](image-url)
Take a look at the code. The above one is related to the first DCT instance and the below one is for the second DCT instance. You can see that two instances are bound to separate resources. DCT_res0 is dedicated to DCTI19 and DCT_res1 is dedicated to DCTI22.

### 6.7.2.2 Resource sharing

Figure 6-34 shows an example of resource file of JPEG Encoder using only one resource of 1-dimensional DCT for 2 instances of DCT blocks. As you see, most of resource file is the same as Figure 6-32.

```
# resource table
# setresourcenum [star_name] [number_of_resources]
setresourcenum ReadBMPBlock 1
setresourcenum PreShiftBlock 1
setresourcenum Block2Int 1
setresourcenum DCT 1
setresourcenum TransposeMem 2
setresourcenum Int2Block 1
setresourcenum BoundDCT 1
setresourcenum JPGQuantizeBlock 1
setresourcenum ZigzagBlock 1
setresourcenum HuffmanEncodeBlock 1

# resource mapping & schedule information
# resmap [instance_name] [resource_id] [start_time] [duration]
resmap ReadBMPBlockI0 0 3 2
resmap PreShiftBlockI2 0 5 1
resmap Block2IntI28 0 6 1
loop 8 7 10 1 {
    resmap DCTI19 0 0 10
}
resmap TransposeMemI16 0 87 1
loop 8 88 10 1 {
    resmap DCTI22 0 0 10
}
resmap TransposeMemI25 1 168 1
resmap Int2BlockI31 0 169 1
resmap BoundDCTI8 0 170 1
resmap JPGQuantizeBlockI11 0 171 600
resmap ZigzagBlockI16 0 771 1
resmap HuffmanEncodeBlockI14 0 772 5
```

Figure 6-34 Schedule Example : Resource Sharing of DCT Block
In the resource(schedule) file. You can see that the resource number of DCT is set to 1. And the resource id of 2 DCT instances are set to 0 in common. This means that 2 DCT instances use a same resource. This is feasible only if these instances are not overlapped in the schedule.

If you have finished writing the schedule file, set the schedule file path on design tab. (You can get the same resource file at SPEACE/lib/demo/hw/jpeg_2d_dct_shared.rs) And execute it as the same way to execute previous JPEG encoder example. You should get the same result again. Figure 6-35 is a snapshot of generated VHDL code for DCT shared JPEG encoder example. This is the part of instance-resource mapping.

Figure 6-35 Generated VHDL code : Resource sharing of DCT block
Take a look at the code. The above one is related to the first DCT instance and the below one is for the second DCT instance. You can see that two instances are bound to the same resource. DCTI19 and DCTI22 have DCT_res0 as DCT resource in common.

### 6.7.2.3 Multiple resource

Figure 6-36 shows an example of resource file of JPEG Encoder using 4 1-D DCT resources for 2 instances of DCT blocks. As you see, the ‘number of iteration’ and ‘resource id’ fields are different from the previous cases.

```plaintext
# resource table
# setresourcenum [star_name] [number_of_resources]
setresourcenum ReadBMPBlock 1
setresourcenum PreShiftBlock 1
setresourcenum Block2Int 1
setresourcenum DCT 4
setresourcenum TransposeMem 2
setresourcenum Int2Block 1
setresourcenum BoundDCT 1
setresourcenum JPGQuantizeBlock 1
setresourcenum ZigzagBlock 1
setresourcenum HuffmanEncodeBlock 1

# resource mapping & schedule information
# resmap [instance_name] [resource_id] [start_time] [duration]
# loop [number_of_iteration] [start_time] [duration] [pipeline_info]
resmap ReadBMPBlockI0 0 3 2
resmap PreShiftBlockI2 0 5 1
resmap Block2Int28 0 6 1
loop 4 7 10 1 {
    resmap DCTI19 0 10
    resmap DCTI19 1 10
}
resmap TransposeMemI16 0 47 1
loop 4 48 10 1 {
    resmap DCTI22 2 10
    resmap DCTI22 3 10
}
resmap TransposeMemI25 1 88 1
resmap Int2BlockI31 0 89 1
resmap BoundDCTI8 0 90 1
resmap JPGQuantizeBlockI11 0 91 600
resmap ZigzagBlockI16 0 691 1
resmap HuffmanEncodeBlockI14 0 692 5
```

Figure 6-36 Schedule Example : Multiple resource of DCT Block
Take a look at the resource number field of DCT. It is set to 4. It means that we use 4 resources for DCT instances. In the loop section you can see that resource mappings for 2 DCT instances exist. 2 resources are running concurrently. Precisely speaking, 8 firings of one resource is now changed to 4 firings of 2 DCT resources. So the number of iteration is reduced to 4 from 8. And, the total schedule length is also reduced.

Figure 6-37 is a snapshot of generated VHDL code for JPEG encoder using multiple DCT resources.
Take a look at the code. The above one is related to the first DCT instance and the below one is for the second DCT instance. You can see that the same instance is related to 2 separate resources. DCTI19 is connected to DCT_res0 and DCT_res1. And DCTI22 is connected to DCT_res2 and DCT_res3. It is a good example that boosts up the parallelism of system by concurrent invocations of resources.

6.8 Running a Demo: H.263 Decoder example

As the last demo example, open the H.263 decoder schematic in Hae/schematic/Peace/Demo/VHDL/H263FRDecoder. (Then, you can see the schematic like Figure 6-38). You can also get the appropriate schedule file at $PEACE/lib/demo/hw/H263.rs. Write that destination directory which will contain the generated codes at directory of the design tab. Also you should check that the schedule file path is set correctly in Resource file parameter of the design tab.

For simple implementation and simulation, we used FLI-interface and C code to build H263FRFrameDecode and H263StoreFrame library block. Therefore they cannot be used for synthesis, they are only for simulation. Except them, all other blocks are described in RTL and synthesizable.
Figure 6-38 looks very complicated. But if you look at the schematic carefully, you can indicate that common part (Figure 6-39) are repeated three times. This means that Y, U, and V frames are processed separately to make a full image.

![Figure 6-39 Block process of H.263 decoder](image)

Take a look at Figure 6-39. At the end of this part, Mux select one from 2 inputs. One is from DecSkipblock. And the other is from $H263DeQ$-$InvZigzagBlock$-$FixIDCTBlock$. The choice is dependent on the block parameter which is from $H263UnpackCBP$. By this value, Mux selects Skipped block or IDCT processed block as output.

![Figure 6-40 repeat for Y block](image)

As you know, we process 4 8x8 Y blocks per 1 decoded MacroBlock. So we attached a Repeat block for parameters needed for Y block dequantizing. (Figure 6-40)
Figure 6-41 Motion Compensation Block and StoreFrame Block

\[H263FRMC\] block at the center of Figure 6-41 is Motion Compensation block. This block gets Y,U,V blocks and other parameters (motion vectors, mode, type) as input. This block is implemented for FRDF. So, 1 firing of this block processes 1/(total number of 16x16 blocks) of image. For example, if we treat 176x144 image as input, the total number of blocks is 99. Therefore, after 99 firings of \(H263FRMC\), 1 output image will be popped out.

Pay attention to the delay of feedback arc between Fork and H263FRMC. The delay is set to 1, because Motion Compensation block refers one previous frame as input.

- setting parameters

Now, we have to set the parameter values to execute H.263 decoder example.
Figure 6-42 illustrates the parameters of *FRFrameDecode*, *H263FRMC*, *H263StoreFrame* blocks respectively.

- **width** : width of image
- **height** : height of image
- **inputFile** : input file path
- **outfileName** : output file name.

**run**

Execute the example by click ‘Run’ button. Notice that output image will be saved per frame and output file will be of ppm format. If you set the outfileName to outImage, the output file name is outImage{frame number}.ppm.

**check out result**

Figure 6-43 shows the result of running. You can see the generated VHDL files. To see the output file, you need image viewer program such as *ee* or *xv*.

164
Figure 6-43 Results of running: H263 Decoder
Chapter 7. Specification with DE Model

Author: Soonhoi Ha and Dohyung Kim

The DE model is widely used to simulate the dynamic system behavior as a function of time: for example, queueing systems, communication networks, and hardware simulations. In a Discrete Event (DE) model, a sample is associated with a **time stamp** indicating when it is generated. And a block processes input events in the order of arrival times. The basic scheduling strategy in the DE model is event-driven: the scheduler collects the output events from all blocks and sorts them to process one by one in the order of non-decreasing time stamp.

Since the DE domain of PeaCE is inherited from the DE domain of Ptolemy classic, both have the same simulation capabilities. Like Ptolemy PeaCE also allows a hierarchical composition of different models of computation, a DE block may encapsulate a nested block diagram with a different model of computation: for example an SPDF block diagram (for a computation task) or an FSM block diagram (for a control task). But there are a couple of differences between PeaCE and Ptolemy classic.

1. Unlike Ptolemy, PeaCE does not allow arbitrary composition of domains. The DE domain may not be nested in an SPDF and FSM domain. The DE domain should be the top-level domain as the simulation engine.
2. Unlike Ptolemy classic, the top-level DE model may intervene the internal behavior of each task. It plays the role of the simulation engine managing interaction between subgraphs (or tasks).

NOTE: The DE domain in PeaCE is used only for system simulation: It is not included in the co-design flow. Nonetheless the DE domain is useful to develop and test each task (especially a control task) since it provides a rich set of library I/O blocks for test input generation output display.

Let’s start the journey to the DE world from drawing a simple DE graph using Pre-defined Library Blocks. If you have not looked at the SPDF section, please do so before proceeding further. We omit the explanation of some terminologies in this section assuming that you are already familiar with them from the SPDF chapter.

7.1 Draw a DE graph Using Pre-defined Library Blocks.

Let us draw a simple DE graph that receives a periodic sequence of saw-tooth wave samples with exponentially distributed inter-arrival times (Poisson arrivals), samples them at a constant rate, and displays the sampled output. Figure 7-1 displays what we are now going to draw and what we obtain from the graph.

The procedure you have to perform in HAE is following:

1. Open a new facet and name it as you like (“File”-“New”). And set the domain to “DE” and the target to “default-DE”. There is a parameter in the default-DE target to select the scheduling option: “calendar queue
scheduler?”. Currently, you set it to “NO”. It means that you use the basic event-driven scheduler. The “calendar queue scheduler” uses multiple event queues to reduce the event sorting time.

(2) Bring two event generators from the block library (library/peace/DE/src): one is a Poisson block and the other is a Clock block. The “mean” parameter of the Poisson block indicates the mean of exponentially distributed inter-arrival times of events: we set it to “1.0”. The “magnitude” parameter indicates the constant value of the output events. The “Clock” block produces periodic events whose period is determined by the “interval” parameter of the block. Please refer to the document for the library blocks to understand the block function.

(3) While event generators define the timing of the events, we need to attach a functional block to define the values of the events. A functional block manipulates the event values and usually has a constant time delay or zero delay. Now, we bring a “Waveform” block from (library/peace/DE/src) to make saw-tooth wave samples with the following parameters: “value = 0 1 2 3 4 5” and “periodic = YES”. In general, a “conceptual” DE-block is modeled by a pair of a delay-type block and a functional block. A delay-type block specifies the delay of the block and the functional block defines the functionality of the block. Considering that an event generator is a delay-type block, pairing it with a “Waveform” block defines a source block that generates a sequence of saw-tooth wave samples with Poisson arrival.

(4) Bring a copy of the “Sampler” block from (library/peace/DE/func/control) and connect the “input” port of the sampler block to the output of the Waveform block and the “clock” port to the output of the Clock block.

(5) For output display, we bring a display block called “Xgraph” from (library/peace/DE/sink) and connect the output of the sampler block to its input port. The “red” color of the Xgraph block indicates that any type of porthole may be connected to. For now, we use the default parameter values for the Xgraph block.
(6) Now, “run” the application by clicking the “Run” menu and setting the “When to stop” value to 30, which indicates that the scheduler stops its operation at global time 30.0 or it stops processing the events with larger time stamps than the stop value.

Now, we believe that you understand how to draw and execute a DE graph using pre-defined blocks. It is fairly straightforward except that it is your responsibility to find out suitable blocks from the block library. If there is no suitable block for your purpose, you have to define your own. This is the topic of the next section.

7.2 DE Block Definition

Let’s make “PWM” (Pulse-Width-Modulation) block which will output as many samples as the input value. We define a parameter called “interval” to determine the time interval between output samples. Make sure that the output samples associated with a given input event are produced before receiving the next input event, by setting the “interval” parameter smaller than the clock period of the Sampler divided by the maximum input value. And, insert this block between the Add block and the Xgraph block in Figure 7-1. Since there is no “PWM” block in the block library, it has to be created.

To begin with, we will create the PWM block using the “copy-and-modify” approach. Since the PWM block has one input and one output port, we copy the Ramp block and modify it: “cp $PEACE/src/domains/de/stars/DERamp.pl ~your_name/work/DEPWM.pl”. Figure 7-2 shows the definitions of the Ramp block and the PWM block. The following sections are modified.

(1) name: Sure, you have to set the name of the block.
(2) author, location, desc, version: These are optional.
(3) input, output: You may change the names. The type of the “input” port is set to “int” meaning that the block receives integer samples from this port.
(4) state or defstate: Define two float states, “interval” and “magnitude” to specify the time interval between output samples and the sample magnitude respectively.
(5) constructor, setup: This block needs no special constructor and setup methods.
(6) go: This section defines the main function body of the block. The PWM block first reads a current sample from the “input” port by calling the “get()” method: input.get(), and assigns the sample value to a local variable, v. If this value is negative, we generate an error message and halt the simulation. The get() method returns the most recent sample value. Note that the Ramp block does not read the input sample since the input sample is used only to trigger the block execution.

The time stamp of the incoming event is set to the “arrivalTime” variable before block execution. A DE block has an internal variable, “completionTime” to record the time stamp of the current output sample. This block is designed to produce \(2 + \text{(input value)}\) output samples including two samples of value 0.0 at the start and the end of the output sequence. The time interval between two output samples is determined by the “interval” parameter. An output sample is produced by using “put(time_stamp)” method whose argument is nothing but the time stamp of the output event: output.put(completionTime).
After you define a new block, insert the block into PeaCE by “importing” the block by clicking “Tool – Import Star” in HAE. Then, the block is compiled and dynamically linked to the PeaCE kernel and a new icon is created in HAE. Figure 7-3 shows how the newly created block is inserted into the block diagram and what are the results of the graph execution.
7.2.1 Delay-type and Functional Blocks

A DE block can be viewed as an event-processor; it receives events from the outside, processes them, and generates output events after some latency. In a DE block, generating output values and computing time stamps are separable tasks. For the purpose of modularity, therefore, some DE blocks, so-called delay-type blocks, are dedicated to time management, adding a non-zero processing delay to the output timestamp, while other blocks, so-called functional blocks, omit time-management and produce output events with the same time stamp as the input events. They, however, do manipulate the value of the samples. Such separation is not obligatory but recommended. The PWM block described above is not a delay-type block since it generates an output event with the same timestamp with the input event. But, it is not purely functional either since it manipulates the time stamps of the output samples.

An example of a delay-type block is the “Delay” block that adds a constant delay to the time stamp of an event. The body of the block definition includes the following:

```c
constructor { delayType = TRUE; }
    go {
        completionTime += double(delay);
        Particle& pp = input.get();
        output.put(completionTime) = pp;
    }
```

If a block is delay-type, you are requested to say so in the “constructor” section by setting “delayType = TRUE”. This information is used by the scheduler to detect whether there is a delay-free loop or not. If there is a delay-free loop, the simulation may go into an infinite loop! Another thing you should note is “Particle” object in the `go()` body. Since the Delay block does not manipulate the sample value but only the time stamp, it just copies the input sample to the output sample. An input sample is carried through a “Particle” object in PeaCE (or in Ptolemy classic). Method call of `input.get()` returns the Particle object. If you type-cast a Particle object, you can get the
value of the sample as shown in the body of the PWM block: \( \text{int } v = \text{input.get();} \) When a Particle object is passed to the output port, you should use “=” operator rather than “<<” operator: \( \text{output.put(completionTime) = } pp; \)

### 7.2.2 Blocks With Multiple Inputs

In the DE model of computation, a block is \textit{runnable} (ready for execution), if any input port has a new event, and that event has the smallest time stamp of any pending event in the system. When the block is executed, it may need to know which input or inputs contain the events that triggered the firing. It can be done by examining the \texttt{dataNew} flag associated with an input port since the scheduler sets the flag when it delivers a new event to the block. The \texttt{dataNew} flag of the port is reset to zero after the input sample is read by \texttt{get()} method. To see how this is done, consider the Sampler block of Figure 7-1.

```plaintext
defstar{
    name {Sampler} domain {DE}
    desc { }
    input { name {input} type {anytype } }
    input { name {clock} type {anytype } }
    output { name {output} type {=input} }
    constructor {
        input.triggers();
        input.before(clock);
    }
    go {
        if (clock.dataNew ) {
            completionTime = arrivalTime;
            output.put( completionTime) = input%0;
            clock.dataNew = FALSE;
        }
    }
}
```

The Sampler block has two input portholes: input, and clock. When an event arrives at the \textit{input} port, it does nothing and simply returns. But when an event arrives at the clock port, it routes the most recently arrived input sample to the output port. In the \textit{go} section, it checks whether a new input event has arrived at the clock input by examining the \texttt{clock.dataNew} flag. Note that the most recently arrived sample is accessed by using “%” operator instead of \texttt{get()} method: \texttt{input%0} instead of \texttt{input.get()}. If we use “%” operator to access the sample, the \texttt{dataNew} flag is not reset to zero.

You can notice that there are two lines you have never seen before in the constructor section. In fact, these two lines are optional. They are needed to help the scheduler schedule the events with the same time stamp. In case there is more than one event with the same time stamp, the scheduler should decide which event to process first. Method call \texttt{input.before(clock)} indicates that the scheduler should give a higher priority to the event to the input port than the event to the clock port. If this method is not called, the behavior is not easily predicted when simultaneous events arrive at the block. On the other hand, \texttt{triggers()} method indicates which output port is triggered instantaneously by which input port. By default, all outputs are triggered, or produce events, when an event arrives at any input. Therefore \texttt{input.triggers()} method indicates that an event to the input port does not produce any zero-delay output. By default, \texttt{clock.triggers(output)} is assumed. Why do we need these complicated things? It helps you to make debugging easier. See section 7.3.1 to understand the need of these methods.
Now we summarize the methods and the operators associated with the ports in the DE domain in Table 7-1. The first four methods are associated with input ports and the last one with output ports.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>get()</td>
<td>Read the newly arrived input sample, or Particle. The dataNew flag is reset to zero.</td>
<td>int v = input.get();</td>
</tr>
<tr>
<td>%0</td>
<td>Read the most recently arrived input sample, or Particle.</td>
<td>Particle&amp; pp = input%0;</td>
</tr>
<tr>
<td>triggers()</td>
<td>Indicate which output ports will produce events with the same time stamp as the input event.</td>
<td>input.triggers(out1, out2);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>input.triggers();</td>
</tr>
<tr>
<td>before()</td>
<td>Determine the event processing order between simultaneous events that arrive at multiple input ports.</td>
<td>input.before(clock);</td>
</tr>
<tr>
<td>put(double time)</td>
<td>Send the output sample with the given time stamp.</td>
<td>output.put(completionTime) &lt;&lt; v;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>output.put(x) = pp;</td>
</tr>
</tbody>
</table>

### 7.2.3 Event Generators

Some DE blocks are event generators that do not consume any event, and hence cannot be triggered by input events. In the beginning of the simulation, these blocks are triggered by the scheduler. All such blocks are derived from the base class RepeatStar in the PeaCE (or Ptolemy classic) kernel. In a RepeatStar, a special hidden pair of input and output ports are created and connected together. This allows the block to schedule itself to execute at any desired future time(s), by emitting events with appropriate time stamps on the feedback loop port. The hidden ports are identical to normal ports, except that they are not visible in the graphical user interface. Let’s look at the body of the Clock definition below.
defstar{
    name {Clock} derivedFrom{ RepeatStar}
    domain {DE}
    desc { Generate events at regular intervals, starting at time zero.}
    output { name {output} type {float} }
    defstate{
        name {interval} type {float} default {"1.0"}
        desc { The interval of events. }
    }
    defstate{
        name {magnitude} type {float} default {"1.0"}
        desc { The value of the output particles generated. }
    }
            go {
                // Generate the output event
                // (Recall that the first event comes out at time 0).
                output.put( completionTime ) << double(magnitude);
                // and schedule the next firing
                refireAtTime (completionTime );
                // Increment the completion time
                completionTime += double(interval);
            }
}

The Clock block is derived from “RepeatStar” that has a hidden pair of feedbackIn and feedbackOut ports. To schedule itself, we use the refireAtTime() method that puts an event to its feedbackOut port. RepeatStar can also be used to emulate time-driven execution of the block: the block is not triggered by the arrival of input events, but by the arrival of feedback events only.

### 7.3 Discrete Event Simulation

Discrete event simulation is a well-known technique to simulate the system’s timing behavior. In the DE domain, events are irregularly spaced in time and system responses are generally very dynamic, all scheduling actions are performed at run-time. At run-time, the DE scheduler processes the events in chronological order until simulated time reaches the global "stop time."

The current DE domain has two schedulers that maintain a global event queue where particles currently in the system are sorted in accordance with their time stamps; the earliest event in simulated time being at the head of the queue. They are the default scheduler and the “calendar queue” scheduler. The default DE scheduler uses a single sorted list with linear searching. You may use the “calendar queue” scheduler by setting the “calendar queue scheduler?” parameter of the default DE target to YES. As the number of unprocessed events increases, the performance gain of using the calendar queue scheduler gets larger.

The basic action of the scheduler is to fetch an event from the head of the global event queue and deliver to the destination block. One thing to note is that the scheduler searches the event queue to find out whether there are any simultaneous events at the other input port of the same block, and fetches those events. Thus, for each firing, a block can consume all simultaneous events for its input portholes. After a block is executed it may generate some output events on its output ports. These events are put into the global event queue. Then the scheduler fetches another event and repeats its action until the given stopping condition is met.
7.3.1 Simultaneous events

You have to pay special attention to simultaneous events in any discrete event simulation. Any DE simulator has its own mechanism to decide which event to process first among the simultaneous events. A simplest approach is to select one randomly. Then, the system behavior becomes non-deterministic and the programmer has hard time to predict the system behavior. In general, we cannot avoid such non-determinism. But, DE simulators usually do something to reduce the probability of such problem as much as possible.

Open the “sampler” demo in Peace/Demo/De/Basic as shown in Figure 7-4. The Ramp block is a functional block with zero latency. At time 0, two Clock blocks generate events to the Ramp block and the Sampler block respectively. Which event do you think should be processed first? If the Ramp block is executed first, the Sampler block will receive two input events, one from the input port and the other from the clock port and produce the output sample that is nothing but the received input sample. But, if the Sampler block is executed first, the Sampler block misses the simultaneous event from the input port.

PeaCE solution, or Ptolemy solution, to this problem is to sort the events in the topological order among simultaneous events. The event to the Ramp block is given a higher probability than the event to the Sampler clock port because the Ramp block is topologically higher than the Sampler block. We set the topological level of all delay-type blocks to zero. On the other hand, some simulators use “delta” time concept for the functional block meaning that the non-zero but infinitesimal delay is added to the time stamp. So, the output sample from the Ramp block is assigned a higher time stamp than the event at the clock port of the Sampler block.

![Figure 7-4 Sampler demo in Peace/Demo/De/Basic](image)

If functional blocks form a loop, topological sorting is impossible. To prevent this from happening, cut the loop setting the delay parameter of any arc on the loop. In the Dataflow model, this delay element indicates the sample delay. However, in the DE model, this delay element is just to set the topological level of the destination block to zero.
If a zero-delay event-path forms a loop, we call it a *delay-free loop*. While a delay-free loop in the SPDF domain results in a deadlock of the system, a delay-free loop in the DE domain potentially causes unbounded computation. Therefore, it is advisable to detect the possible delay-free loop at compile-time while it may be a false alarm. In PeaCE, we detect the possibility of a delay-free loop by examining the “triggering” path specified in the Block definitions. If a zero-delay triggering path is found, a warning is signaled at compile time. The programmer may ignore such warning at his/her responsibility.

### 7.4 Mixed Simulation Capability

The key feature of Ptolemy is to mix heterogeneous models of computation in a hierarchical fashion. Since PeaCE is extended from Ptolemy, it has the same capability. Differently from Ptolemy that allows arbitrary mixture of models, however, we restrict the usage of the DE model as the top-level model for system-level design. If you use a super block of a different domain in a DE domain, the super block is called a Wormhole in Ptolemy’s terminology. In this section, we will explain how to mix the DE simulation with SPDF simulation and analog simulation. See the next chapter for the mixture of the DE model and the FSM model.

#### 7.4.1 Mixed Simulation With SPDF Model

Mixed simulation with SPDF model can be easily achieved by using a SPDF super-block (or Galaxy) in the DE domain. Figure 7-5 shows a simple “worm” demo that contains a SPDF super-block that adds an input sample and a Gaussian random value to produce an output. You can find this demo in the demo directory of PeaCE. In this simple example, the super-block contains only two SPDF blocks: one is Add and the other is IIDGaussian. If you look-inside this super block and check the domain, you can find that it is set to “CGC” and its target is set to “worm-CGC”.

![Figure 7-5 de/wormhole/worm demo](image)

Since the CGC domain follows the SPDF model, the inside graph is a SPDF graph. At the setup stage of the application graph, the super block creates a scheduler object and schedules the inside graph: the schedule of the inside graph becomes IIDGaussian – ADD. When the super block receives an input event from the Poisson block,
the inside graph is executed once. The output of the super-block is now delivered to the destination block with the same time stamp as the input sample. In other words, the super-block is regarded as a functional block. You have to append a delay block if you want to model the execution delay of the super-block.

In case a super-block of SPDF model has only one input sample, the operation is simple. But if a super-block has more than two inputs, you need to be cautious. Figure 7-6 shows such an example: “adder” example. The “sdf_adder” super-block contains only a single Add block that is executable only when both inputs have new samples. On the other hand, the super-block is fired from the DE scheduler when a new input event arrives at any input. There is a semantic mismatch between the outside DE model and the inside SPDF model. To resolve this mismatch, the wormhole (super-block) performs some interface logic. The default interface logic is to queue the input samples at each port until the wormhole has new events at all input ports. When all input ports have new events, the wormhole deliver one input sample per port to the inside SPDF graph. In this example, one input port receives input samples with period equal to 3 while the other with period equal to 2. Then, the input samples with period 2 are queued up.

Figure 7-6  “adder” demo

Since such data synchronization action is not evident from the programmer, we recommend you to use explicit synchronization block at the input ports to a wormhole with more than two input ports. In Figure 7-7, a Sampler block is added between the Ramp block and the “sdf_adder” super-block. The input samples with period 2 are sampled by the samples with period 3. Then, some input samples with period 2 are overwritten by the subsequent samples before being sampled. For instance, a sampled produced at time 4 is missing in the result.
In fact, mixed simulation with SPDF model is a distributed simulation between PeaCE kernel and the C process created by the CGC domain. Therefore, the target of the wormhole should be set to “worm-CGC” since this target will add the required socket interface code to the generated C code.

7.4.2 Mixed simulation with Octave Analog Simulator

In this section, we show how PeaCE can achieve mixed simulation with an analog simulator, called Octave. Octave has similar capability to MatLab for numerical computation but it is a free-ware under GPL license. Since Octave allows interactive execution by interpreting the command line queries, it is simple to use Octave program in the DE model. Figure 7-8 shows a simple example of mixed simulation.

There is a DE block, called OctaveODESolver, that is used to compose the queries and send them to Octave. At the setup stage, PeaCE forks Octave as a child process and kills it when the simulation ends. The
OctaveODESolver block writes the queries to the standard input, which are delivered to the Octave via unix pipe mechanism. The OctaveODESolver waits until it receives the outputs from the Octave via pipe.

If you click the OctaveODESolver block, you can see the parameter window as shown in Figure 7-9. There are two states you have to fill up for composing the query. In the “query” state, you write down the first order differential equation and variable initialization. The initial values of state elements are put into the “initValue” state that is float-array type. The example of Figure 7-9 shows a simple RC circuit response. The Clock and WaveForm blocks generate a square wave of amplitude 5. The OctaveODESolver block specifies the first order differential equation of the RC circuit. The result is displayed with an X graph.

Using the same graph, we can obtain a different result by changing the state values. Figure 7-10 shows a different set of states to solve van der Pol equation. This demo can be found in the demo directory.
For more detailed discussion on the query format, please refer to the Octave manual that is available on-line. There are some limitations of mixed simulation with Octave. Currently, mixed simulation is performed as a distributed simulation that takes significant portion of time for inter-process communication. Second, the current version of PeaCE supports a subset of Octave queries: queries for differential equation. It is kind of experimental capability of PeaCE. If needs arise, more complete support will be provided.
Chapter 8. FSM Model Specification

Author: Soonhoi Ha and Dohyung Kim

Finite state machine (FSM) model is useful to express the behavior of a control (or reactive) system like a bending machine, a cruise control system of a car, a traffic light signal and etc. Reactive systems collect input data from buttons, sensors and timers and send output data to actuators that change external environments - a coke and coins, speed of a car and blinked walk signal of traffic light signal.

Even though FSM is simple to use, its unstructuredness and state explosion problem due to system concurrency and memory prohibit FSM model from practical representation of complex control behavior. To describe complex control modules, the extensions should support various kinds of concurrency. Another desired feature is compositionality whether the complex module can be represented as a composition of simpler modules. Then, modules can be easily reused to construct a large system. Moreover, because of their complexity, it is desired to have a static analysis method to check ambiguous behavior. Finally for fast prototyping, automatic hardware (or software) synthesis is needed from the extended model. To meet these requirements, PeaCE devises a new extension of FSM model, called fFSM Model. It extends the expression capabilities by concurrency, hierarchy, and state variable.

This chapter explains how to draw an fFSM graph in PeaCE and how to use it in the system level modeling.

NOTE: An FSM graph is not used alone but used as a sub-task in another outer graph that provides input events to the FSM graph and displays the output. The outer domain can be either DE (for simulation) or BP (for code generation).

8.1 Basic FSM Definition

State transition graph is a directed connected graph composed of states and transitions. A state is drawn as a circle and a transition is a directed arc that connects two states. Each state indicates a unique memory location that describes specific status of the system and a state with bolder line is the initial system state. And a transition has a condition that must be fulfilled to execute actions in the form of 'condition/actions' above the line and make state transition from a source state to a destination state. State transition graph also has event definitions for input, output, internal event and variable state.

Here shows a simple state transition graph (Figure 8-1).
There are two states – ‘start’ and ‘stop’ – and ‘start’ state is the initial state. And there are two transitions which have a condition ‘toggle==1’ and no action. It means that it start at ‘start’ state and each time it gets ‘toggle’ event, it changes the current state to the other state.

To draw a simple graph, you can start with opening a new schematic and select the FSM domain and default-FSM target. And a state icon is located in library-Peace-FSM of Lib tab as shown in Figure 8-2.

You select a state block and move it to new schematic twice (Figure 8-2). And then select one of state block and fill states( we will use the term “parameter” instead of “state” in the FSM domain to avoid the confusion between the behavioral state and the block parameter) of the state block. ‘name’ is the unique name for the state and ‘init’ indicate if the state is the initial state or not with ‘TRUE’ and ‘FALSE’ value (Figure 8-3). Other states will be discussed later.
After that, draw transition lines between two states. A transition has a source state and a destination state. If you move the mouse pointer close to the state, the green circle occurs and first click makes the state as a source state. And you move the mouse pointer to the middle of a source state and a destination state and click the button again to draw smooth curve. And third click on the green circle of the destination state makes a transition line between them (Figure 8-4).

Then select a transition line and fill the condition and actions (Figure 8-5).
A transition is associated with two parameters: condition and actions. If the specified condition is meet, the actions in the transition are executed and the current state is changed from the source state to the destination state.

Parameter syntax

- **Condition**: C condition expression. All events except output events can be used in the expression.
- **Action**: [{variable_name=expression}], Variable can be any event but an input event

If you define ‘toggle’ event as shown in Figure 8-6, it completes the design of the simple state transition graph. Finally you enroll this FSM galaxy to library block using ‘Tool-Register Galaxy’ menu. Input events defined like Figure 8-6 are shown as input ports of FSM galaxy and output events as output ports. Because you define one ‘toggle’ input event, the library block will have one input port.
In the state transition graph, there exist four kinds of events: They are input event, output event, internal event and variable state. Input and output events define the external ports of an FSM super-block. An internal event defines an event for internal communication. Variable state defines a global storage. We explain the usage of internal event and variable state later. And each event has the definition of "name", "type" and "scope".

- **Name**: unique in the FSM domain and referred in condition and actions description
- **Type**: determine the event type that is one of \{"PURE", "VALUE"\}, default value is "PURE". "PURE" event indicates that occurrence of the event not the value of the event triggers the FSM machine. On the other hand, the value of an event is important for a "VALUE" type event.
- **Scope**: define the range of value that an event can have. The scope of a “PURE” event is \{0,1\}.

And events are defined as super-block (galaxy) parameters for the top level FSM domain as shown in Figure 8-6.

- **InputNames** : names of input events
- **InputTypes** : types of input events
- **InputScopes** : range of input event values
- **OutputNames** : names of output events
- **OutputTypes** : types of output events
- **OutputScopes** : range of output event values
- **InternalNames** : names of internal events
- **InternalTypes** : types of internal events
- **InternalScopes** : range of internal event values
- **VariableNames** : names of variable states
- **VariableTypes** : types of variable states
- **VariableScopes** : range of variable state values
- **VariableInits** : initial values of variable states

And the syntax of those parameters becomes

- **Name definition**: \[{name}\]*
- **Type definition**: \[{PURE}|{VALUE}]*
- **Scope definition**: \[{0,1}|{min value, max value}] *

We make the following rules to make each event name unique in FSM domain because each event must be able to be referred uniquely in conditions and actions.

- Each event name must be unique within the sub-graph. A same name can be used in multiple subgraphs.
- Each FSM galaxy name must be unique globally.
- In a child FSM, an input event name must be one of input events or internal events of the parent FSM in the hierarchical FSM graph. In a child FSM, an output event name must be one of output events or internal events of the parent FSM.

The example shows an example FSM graph and galaxy parameters located at the left bottom corner. If you look at the galaxy parameters in Figure 8-7, it helps you to understand the syntax of event descriptions.
8.2 fFSM: FSM extension

To describe complex control modules, an FSM extension should support concurrency. Another desired feature is compositionality whether the complex module can be represented as a composition of simpler modules. Then, modules can be easily reused to construct a large system. Moreover, because of their complexity, subtle design errors are difficult to find and the equivalence check of an implementation is not easy to achieve. Therefore, it is desired to have a static analysis method to check ambiguous behavior. Finally for fast prototyping, automatic hardware (or software) synthesis is needed from the extended model.

In this chapter, we explain how those requirements can be satisfied with a proposed FSM model. The proposed FSM model is called flexible FSM model meaning that there are more than one way of expressing concurrency. It extends the expression capabilities by concurrency, hierarchy, and state variable while it maintains the formal property of the base FSM model. Because of formality and the structured nature of fFSM model, we can apply a static analysis method to find ambiguous behavior and synthesize software/hardware automatically. Moreover, it has a special syntax to express memory in a compact form. In a formal FSM model, if a system has a memory element, the number of states explodes as the number of possible values does. We introduce a new concept of “variable states” to overcome this difficulty.
8.2.1 Concurrency

fFSM expresses concurrency as illustrated in Figure 8-8. Bold lined states indicate initial states and a dotted-line divides two concurrent FSMs. Two states A and C are concurrently active at the same time. And transitions may occur simultaneously if both inputs “a” and “c” are active. No direct transition is allowed between concurrent states. However, two concurrent states can affect each other by exchanging internal events to be explained below. This constraint is for formality and simplicity.

![Figure 8-8 Concurrency expression](image)

The example shows an fFSM example showing concurrency in PeaCE. Each concurrent FSM has its own initial state and is not connected to each other.

![Figure 8-9 Concurrent FSMs](image)

8.2.2 Hierarchy

Hierarchy means that a state can have another FSM inside. When the outer state is active, the internal FSM becomes active and initial state is ready to execute. And as soon as the outer state is inactive, the internal FSM also becomes inactive. We avoid the use of inter-hierarchy transitions, unlike Statechart. Figure 8-10 shows an example fFSM graph that shows hierarchy.
If the current state is A and input event “a” occurs, the next state becomes state B and internal FSM becomes active, which makes the state C active. When input event “b” occurs, the internal FSM exits to go to state A. But it has an ambiguity problem. Suppose that the current states are B and C and input events “b” and “c” occur simultaneously. Which transition should be executed first? There are a number of methods to determine the priority between transitions. We assign an outer transition higher priority in PeaCE.
To draw a hierarchical FSM, first you design a FSM subgraph like Figure 8-11 and save it. And at the parent FSM, fill ‘internal FSM’ state in the state block as the file name which you just created as shown in Figure 8-12. And if you want to see the internal FSM of a state, just double-click the mouse button on the state in which you are interested.

8.2.3 Internal event

Internal event is used for communication between concurrent states and/or hierarchical states. Because it may cause ambiguity, we adopt the delta-delay model. All input events exist during delta-delay phase only. If there is any new internal event, it becomes valid at the next phase after delta-delay and all previous events set to be invalid.

Figure 8-13 shows how we can avoid ambiguity by applying the delta-delay model of execution. If a current state is start and input event “a” and “e” occur, an internal event “b” becomes valid and the current state is changed to state A. We cannot determine which transition is executed if the current state is A and events “e” and “b” are active. But we assume that event “e” is deactivated after delta-delay and the next state will become B.

If you want to use an internal event between concurrent FSMs, first you define an internal event in the galaxy parameter. And then if you want to send the signal to other FSM, you can write an action script like ‘internal_event = value’. And the receiving FSM uses the internal event in the condition definition. Let’s see an
example (Figure 8-14). We define ‘tostart’ internal event as PURE type. And at ‘reset’ state, the ‘tostart’ internal event is sent to ‘stop’ state, which makes a state transition to ‘start’ state.

**Figure 8-14 Internal event example**

If you want to use an internal event across hierarchical FSMs, you define an internal event at the parent FSM. And at the child FSM, you define the internal event either as the input event if you want to receive the signal from the parent, or as output event if you want to send the signal to the parent. To avoid ambiguous transition, you would better to have only one writer for the internal event.

In the example, ‘stop’ state has a child FSM (Figure 8-15). And the parent FSM defines ‘tostart’ internal event. And the child FSM defines ‘tostart’ output event to send the signal to the parent FSM. So at the ‘stop’ state, if the FSM receives ‘reset’ event, the parent FSM makes a state transition to ‘start’ state by the ‘tostart’ internal event.
8.2.4 Variable state

A variable state provides a memory to FSM model. It is considered as a local variable and expressed as a concurrent FSM. The usage of a variable state is similar to an internal event. While an internal event disappears after delta-delay, the variable state always exists. A variable state is formally equivalent to a concurrent FSM except that it can be examined and updated by other concurrent FSMs. If it is used as a condition, it is called a state reference. If it is used as an output, it implies a state transition.

Figure 8-16 Variable state expression
Figure 8-16 implements a time-out FSM using a variable state. An input event “time” is an external clock and an output signal ‘timeout’ is the result output which indicates that actual timeout occurs. If an outer block wants to set a timeout, it sends “start” signal to the timeout FSM with a required timeout value. After the timeout FSM gets the input, it sets the variable state “remain” as the value of the input event “start”, changes the state to “wait”. It decreases the “remain” state at each “time” input until the “remain” value becomes zero. Then a transition from “wait” to “init” occurs and a “timeout” signal is supplied to the outer block. If we specify this example by using previous FSM models, we must create different FSMs with different time-out values.

Figure 8-17 shows the timeout example using variable states. First we define ‘remain’ variable state to store remaining time that has a scope from 0 to 10 and is initialized by value ‘0’. And if the upper FSM sends ‘timeset’ internal event, the lower FSM receives the event and stores it in the ‘remain’ variable state. And each time the lower FSM gets ‘time’ input event, it decreases the ‘remain’ variable state until the ‘remain’ variable state become 0. If the ‘remain’ variable state becomes 0, then it sends ‘timeout’ internal event to the upper FSM.

8.3 FSM Simulation

FSM model cannot be executed alone but is simulated with DE domain. After registering an FSM galaxy, you use the FSM block in the DE domain to make it a hierarchical block diagram. The DE blocks provides input events to the FSM block and receives the output samples. To understand how it works, open the ReflexGame demo in
PeaCE as illustrated as Figure 8-18. Note that this demo can also be found in Ptolemy class but with different specification method.

Run the demo, then you will see a nice user interface come up with three buttons inside. The TclScript block at the input of the FSM graph reads a Tcl script (specified as a block parameter) to construct such a nice user interface. If you click a button, the block generates an input to the FSM block. The Synchronize block generates clock events to the FSM. The TclScript block at the output port displays the result. It is a neat game to measure your reaction speed!

![Figure 8-18 Reflex Game FSM demo](image)

If you look inside the game_FSM block, you will see a concurrent and hierarchical fFSM graph. Even though the internal behavior of the game_FSM block is rather complicated, it can be easily represented in fFSM model using concurrency, hierarchy, and variable states. The target of the FSM graph is 'default-FSM' that is a simulation target with no settable parameter.

Figure 8-19 shows top-level fFSM graph specification of the reflex game demo. As shown in the left table, you can define input events, output events, internal events and variable states. The upper fFSM graph of the right schematic controls ON/OFF state of the game and the lower fFSM graph implements time-out fFSM using a variable state. Two fFSM graphs in Figure 8-19 are simultaneously active by concurrency.

In the GameOn state, an hierarchical fFSM graph is defined to handle interactions of the game which is illustrated in Figure 8-20. Events used in the child fFSM graph can be inherited from the parent fFSM graph or be newly defined for its own usage. When they are inherited, you write only event names of the parent fFSM graph. Otherwise, you should define all properties of the event (Ex. 'rand' variable state). The upper fFSM graph handles interactions and the lower fFSM graph generates random time delay to test reaction speed.
Figure 8-19 Top level fFSM graph specification of reflex game demo

Figure 8-20 Child fFSM graph specification in the GameOn state
8.4 Talking with Computation Module

While an FSM graph specifies the control behavior of a system, we use SPDF subgraphs or DE blocks to specify the computation module of the system. For example, in Figure 8-21, Ramp block in the DE domain models the computation module of the system. The Ramp block generates a sequence of increasing numbers to make a counter. The FSM graph receives the user input and commands the Ramp block to reset the counter value or to stop counting. So interaction mechanism between the FSM graph and the computation module is needed. This section explains how to make an FSM graph talk with computation module. For simplicity, we assume that a DE block represents a computation module. Note that the DE computation block encapsulates an SPDF graph inside to model a complicated computation module of the system.

![Simple Counter Demo](image)

**Figure 8-21 Simple Counter Demo**

Interaction between a DE block(function module) and an FSM graph(control module) is divided into two categories: one is *synchronous interaction* and the other is *asynchronous interaction*.

In a synchronous interaction, a control module and a computation module communicate with each other by exchanging data samples through an arc between two modules. The computation module may send a sample to set a flag of a control register and the controller becomes active on the arrival of the flag. Or, the control module may send a data sample that activates the computation module to process the sample. The Simple Counter Demo does not possess this kind of interaction.

Using asynchronous interactions, a control module plays the role of a supervisory module to manage the state of a computation module. We define three states of a computation module by its current activity: *active*, *suspend* and *stop* as illustrated at Figure 8-22, where a computation module is represented as an SPDF graph. If there is no synchronous input interaction from the FSM module to a computation module, the computation module goes into the active state from the start by default. When the control module enters into a certain state, it may want to suspend the computation module and resume it later. When the computation module goes into the suspend state, it stops its execution and just discards the incoming samples from the outside environment.
Another situation of asynchronous interaction occurs when the control module wants to change parameters of computation module asynchronously. Suppose a computation module plays an encoded audio. If the user wants to lower the volume, that action is delivered to the control module and finally to the computation module by asynchronously changing the "gain" parameter of the internal dataflow node that amplifies the sound samples as shown in Figure 8-23.

Though there are many possible methods to describe asynchronous interactions, we choose to use a "script" language inside a state node in an FSM. For asynchronous interaction, we use a script language inside a state node in an FSM. Each FSM state has the ‘script’ parameter. If the ‘script’ parameter is defined, the script is executed after the state is entered.

To use a script in the state, you must specify a node name of a destination computation module first, which make possible to indicate a specific node in a script. In DE domain, every block except wormholes has the ‘nodename’ state as shown in the left of Figure 8-24.
But if the computation block is a wormhole that contains a CGC graph, you must add a ‘nodename’ state as string type to the super-block(or Galaxy). Then you can specify the block name.

After specifying the unique name ‘ramp’, you can define the script in the state. Table 8-1 shows the syntax of the scripts. The first three manage the states of the computation modules and the last script describes the asynchronous parameter updates of the named block. The difference between “suspend” and “stop” command is that the suspended block resumes its action starting from where it was suspended while the stopped block resumes from the beginning when resumed.

**Table 8-1 Syntax of script language**

<table>
<thead>
<tr>
<th>Scripts</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run n_name</td>
<td>Resume the n_name block</td>
</tr>
<tr>
<td>Suspend n_name</td>
<td>Suspend the n_name block</td>
</tr>
<tr>
<td>stop n_name</td>
<td>Stop the n_name block</td>
</tr>
<tr>
<td>Set n_name</td>
<td>Update the parameter with value in the n_name block</td>
</tr>
</tbody>
</table>

Figure 8-25 shows an example of script language usage. In the ‘start’ state, ‘run ramp’ is specified. And in the ‘stop’ state, ‘stop ramp’ is used. When each ‘toggle’ input data is arrived, a state transition is occurred and each script of destination state is called. So we can change the run time status of the ramp block in the DE domain.
run status, the Ramp block increases the counter value. In stop status, the Ramp block initializes the counter value and is stopped. You can use ‘suspend’ script as the same way.

![Diagram](image)

**Figure 8-25 Script language usage example**

In the start state, there is a hierarchical FSM, called ‘resetFSM’, as shown in Figure 8-26.

![Diagram](image)

**Figure 8-26 resetFSM**

The reset state has ‘set ramp value 0’ script which means that it change ‘value’ state of ‘ramp’ node to ‘0’. So each time FSM receives ‘reset’ input data at ‘start’ state, FSM send the signal to reset the counter value.
8.4.1 Simple Counter example

In this section, let’s make the Simple counter demo as shown in Figure 8-21. Upper DE blocks models a increasing counter and the TclScript block creates a user interface with ‘toggle’ and ‘reset’ buttons to generate input events to the FSM block. The FSM block gets two inputs and supervises the execution of Ramp block.

(1) First you open a new facet and name it as you like. Set the domain to DE and the target to “default-DE”.

(2) Draw the block diagram using the library blocks. You can find the Clock block, the TclScript, and the Ramp block from “/library/peace/DE/Src”, the TkShowValue block from “library/peace/DE/Sink”. And then you set the ‘nodename’ state of the Ramp block as “ramp” for asynchronous interaction with the FSM graph. And make as below the ‘button.tcl’ file which is a tcl script file that creates two buttons. The file is set to the ‘tcl_file’ state of the TclScript block.

```tcl
set s $ptkControlPanel.SimpleCounter
if {![winfo exists $s]} {
    toplevel $s
    wm title $s "FSM Demo"
    wm iconname $s "FSM Demo"
} else {
    wm deiconify $s
    raise $s
}
set buttons $s.buttons
catch {destroy $buttons}
frame $buttons -relief groove -bd 3
pack $buttons -padx 3 -pady 3 -expand 1 -fill both
label $buttons.label -text "You can push:"
pack $buttons.label -side top
button $buttons.toggle -text toggle -bd 4 -padx 3 -pady 3 -command "setOutput_$starID 1 1"
button $buttons.reset -text reset -bd 4 -padx 3 -pady 3 -command "setOutput_$starID 2 1"
pack $buttons.toggle $buttons.reset -side left -expand 1 -fill both -padx 3 -pady 3
tkwait visibility $buttons
```

(3) Create the FSM subgraph as explained above. Figure 8-27 shows the resultant FSM graph and the state parameters. Note that the “start” state nests an internal FSM graph for “reset” event handling.
Figure 8-27 Parent FSM and related parameters

The internal FSM and the associated state parameters are displayed in Figure 8-28.
If you run the design, you can see the two buttons and the run panel that contains a display box of increasing numbers (Figure 8-29). You can change the execution by pushing buttons.
8.4.2 Add a SPDF super-block

Now we add a SPDF super-block as another computation module in the system. You will see that if you want to change the state value of CGC galaxy, you must do a few more things than specifying the “nodename” state of the block.

![Figure 8-30 Adding a buffer super-block](image)

We add a new buffer super-block (a CGC graph inside) which buffers the counter value, multiplies a gain value and sends the result data to display as shown in Figure 8-30. And we suppose that we want to change the gain value.

1. First we add ‘nodename’ state to the super-block.

2. Because we cannot directly access the gain value of the block, we must make a path from the super-block to the block. So we add a ‘globalstate’ state as a string array. The state indicates which states are used for the parameter update. And we add a super-block parameter that has the same type of the gain value. We name it as ‘gainvalue’ and the type is float.

3. Next we set the ‘globalstate’ of the super-block and the ‘gain’ state of the Gain block to ‘gainvalue’. Left Figure shows the super-block parameters and right Figure the block parameter (Figure 8-31).

201
And then if you change scripts of FSM state as follows, you can see that the gain value for multiplication is changed each time you push ‘toggle’ button (Figure 8-32).
8.5 Code Generation

This section shows how to synthesize a C code or a VHDL code from the FSM graph. To generate a C code
‘FSMCGCTarget’ should be taken: It generates C code from the fFSM graph, compiles, and executes it. For
VHDL code generation ‘FSMVHDLTarget’ or ‘FSMSimVHDLTarget’ should be taken. The latter generates a
VHDL code from the fFSM graph and interface codes for Synopsis VHDL simulator. The target parameters for
each target are summarized in the sub-sections.

Note: current version of PeaCE only support C code generation in ‘FSMCGCTarget’ and VHDL code generation
in ‘FSMVHDLTarget’. But these two targets cannot be cosimulated with other targets. Instead, current version of
PeaCE supports another C code generation in ‘FSMTaskTarget’ that is simulated with BP domain. It will be
explained in Section 9.

8.5.1 FSMCGCTarget

The target parameters of ‘FSMCGCTarget’ are listed below with brief explanation.

- host (default “”): host name which compiles and executes the code
- directory (default “$HOME/PTOLEMY_SYSTEMS/FSM”): directory where the file locates
- file (default “”): name of the file to save the generated code
- printStruct? (default “YES”): print the fFSM structure text file if “YES”
- display? (default “YES”): display the generated C code if “YES”
- compile? (default “YES”): compile the generated C code if “YES”
- run? (default “YES”): run the executable after compilation if “YES”
- compileCommand (default “g++”): indicate the compiler command PeaCE uses
- compileOptions (default “”): indicate the compiler options
- linkOptions (default “”): indicate the link options during compilation

Code b) shows the C code structure generated from an FSM subgraph. Each concurrent state makes a ‘switch()
{ case ‘}’ phase, each transition makes an ‘if then else’ sentence, and each action in transition constructs a C
sentence. We distinguish normal transitions and internal transitions, which makes more efficient execution.
Because of delta-delay execution semantics, an input event is only valid during the first phase and any internal
event will be valid from the second phase. So first we check normal transitions and if there exists any internal
event, we iterate to check internal transitions until there is no more events as code a).

read input events;
loop init code;
process normal transitions of fFSMs;
transition wrapup code;
while(if there is any internal event) {
  process internal transitions of fFSMs;
  transition wrapup code;
}
send output events
a) fFSM main loop code
switch (for each concurrent FSM) {
    case for each state:
        if (for each condition) {
            do_actions;
        } else if (another condition) {
            ...
        } else {
            execute internal fFSM; // if exists
        }
    case ...
}

b) Each FSM code

8.5.2 FSMVHDLTarget
- host (default "") : host name which compiles and executes the code
- directory (default "$HOME/PTOLEMY_SYSTEMS/FSM") : directory where the file locates
- file (default "") : name of the file to save the generated code
- printStruct? (default “YES”) : print the fFSM structure text file if “YES”
- display? (default “YES”) : display the generated C code if “YES”
- clockPeriod(ns) (default 20) : indicate the default clock period for VHDL simulation
- Environ (default “/usr/tools/synopsis”) : the location at which synopsis tools is installed
- Architecture (default “sparcOS5”) : the host architecture on which the tools are executed
- Analyze (default “YES”) : indicate whether analyze process will be executed or not

8.5.3 FSMSimVHDLTarget
- host (default "") : host name which compiles and executes the code
- directory (default "$HOME/PTOLEMY_SYSTEMS/FSM") : directory where the file locates
- file (default "") : name of the file to save the generated code
- printStruct? (default “YES”) : print the fFSM structure text file if “YES”
- display? (default “YES”) : display the generated C file if “YES”
- clockPeriod(ns) (default 20) : indicate the default clock period for VHDL simulation
- Environ (default “/usr/tools/synopsis”) : the location at which synopsis tools is installed
- Architecture (default “sparcOS5”) : the host architecture on which the tools are executed
- Analyze (default “YES”) : indicate whether analyze process will be executed or not
- Simulation (default “YES”) : run Synopsis VHDL Simulator if “YES"
Interactive mode (default “NO”) : run the simulation in the interactive mode if “YES”
Debug mode (default “NO”) : run the simulation in the debug mode if “YES”

![Main fFSM block diagram](image)

Figure 8-33 shows the main fFSM block diagram. Because main fFSM block consists of the network of concurrent FSM blocks, it has the definition of input and output events for external interface. And there exist concurrent fFSMs and internal event logic between fFSMs. Below code shows the basic VHDL structure generated from an FSM.

```vhdl
{FSM name}_SEQ_LOGIC:
    process(clk, reset)
    begin
        if ( reset='1' ) then
            current state <= initial state;
        elseif rising_edge(clk) then
            ## if the FSM has a parent FSM ##
            if (parent FSM's current state=parent state) then
                current state <= next state;
            else
                current state <= idle;
            endif;
            ## if the FSM does not have a parent FSM ###
            current state <= next state;
        endif;
    end process {FSM name}_SEQ_LOGIC;

{FSM name}_COM_LOGIC:
    process(current state, [{Input event}, ...])
    begin
        case (current state) is
            -- for each state
            when {state name} =>
                -- for each transition
                if ({{condition}}) then
```

205
next state <= next state name;
-- for each action
output event <= expression;
....
elself .. other conditions
endif
when {state name} => other state
when others =>
current state <= initial state;
end process {FSM name}_COM_LOGIC;
Chapter 9. Task level specification model (BP Domain)

Author: Soonhoi Ha and Dohyung Kim

The SPDF model and fFSM model is used to describe the internal behavior of a computation and a control task respectively. We use another model, called Task Model, to describe the system behavior at the task level. Thus we are able to represent multi-tasking applications. In PeaCE, we can define a super block that encapsulates a block diagram with a different model of computation to make a hierarchical block diagram. At the top-level, the Task model represents a composition of multiple tasks: each task is modeled as a SPDF model or fFSM model. Since we use the Task model in this fashion, we call the domain of Task model as the BP (Backplane) domain.

NOTE: The BP domain is a code-generation domain, So a DE block (or a DE graph) can not be used in the BP domain at the current implementation. The BP domain is used mainly for HW/SW codesign of the system though you may use it only for functional simulation of the system. To use it as a codesign backplane, open a new project instead of a new facet. The next chapter will describe the overall codesign flow.

9.1 Task model in PeaCE

Tasks in the system have diverse activation conditions and port semantics that should be clearly specified in the task-level specification model at the top-level. Table 9-1 summarizes the classification of supported task types and port properties. We support three types of tasks: periodic tasks triggered by time, sporadic tasks triggered by external IO, and function tasks triggered by data. And we also define various port semantics by combining port type, data size, and data rate. For example, the data port semantic for SDF model is classified as a static-rate static-size queue and the data port semantic for FSM model as a dynamic-rate static-size queue.

<table>
<thead>
<tr>
<th>Task type</th>
<th>Periodic</th>
<th>Triggered by time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sporadic</td>
<td>Triggered by external IO</td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>Triggered by data</td>
</tr>
<tr>
<td>Port type</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Queue</td>
<td>Destructive read/non-destructive write</td>
</tr>
<tr>
<td></td>
<td>Buffer</td>
<td>Non-destructive read/destructive write</td>
</tr>
<tr>
<td>Data size</td>
<td>Static</td>
<td>Size is determined at compile time</td>
</tr>
<tr>
<td></td>
<td>Variable</td>
<td>Size is determined at run time</td>
</tr>
<tr>
<td>Data rate</td>
<td>Static</td>
<td>Rate is determined at compile time</td>
</tr>
<tr>
<td></td>
<td>Dynamic</td>
<td>Rate is determined at run time</td>
</tr>
</tbody>
</table>

Table 9-1 Classification of task types and port properties
Note that the internal behavior of a task is represented as the SPDF model or the fFSM model. On the other hand, the task itself has diverse execution semantics as shown in Table 9-1. Then, the problem is how to connect the internal SDF and FSM models to the outer task-model.

We use a task wrapper approach where a task wrapper is created at the boundary of hierarchical node (or a task node) that translates the outer execution semantics to the internal execution semantics. For example, if a SPDF task is defined as a periodic task with a static-size static-rate buffer port, arrival of input data is ignored while the arrived data is stored in the port buffer. Instead, the current values stored in the input port buffers are delivered to the inside at periodic wake-up. Such translation is the role of the task wrapper and is performed in the kernel of PeaCE, being invisible to the user: Never mind how it works.

Figure 9-1 overviews the sequence of task wrapper creation using a DivX player example. The DivX player example consists of one control(FSM) task(control FSM) and four computation(SPDF) tasks(User interface, control FSM, AVI reader, H.263 decoder, and MP3 decoder).

Figure 9-1 (a) Task level definition without applying task wrapper (b) control ports appended by control path analysis (c) control ports appended by task types
(1) At first, the task type of each is specified by the user. The “user interface task” is a sporadic task that is triggered by the user input and the “AVI reader” task is a periodic task that is invoked periodically. The other tasks are set as function, which is the default task type so can be omitted.

(2) The task wrapper defines port semantics for each data port of SPDF and fFSM models. In this example, data ports between the user interface task and the control FSM task are defined as dynamic-rate static-size buffer-type.

(3) It appends control ports to the SPDF or fFSM task by analyzing the scripts in the FSM model (see Chapter 8 for the detailed explanation of the fFSM model and scripts) and also defines a port semantic for each type of control port. In Figure 9-1 (b), two control connections are created between the control task and the divx mode to exchange the control command and the status signal.

(4) At last, it appends an additional control port associated with the task type. Through this port, the supervisory task gives commands to change the execution status of the task. We summarize the port semantics for the task wrapper in Table 9-2.

<table>
<thead>
<tr>
<th>Port type</th>
<th>Port semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data port of SDF model</td>
<td>Static-rate static-size queue</td>
</tr>
<tr>
<td>Data port of FSM model</td>
<td>Dynamic-rate static-size buffer</td>
</tr>
<tr>
<td>Scheduling control</td>
<td>Static-rate static-size queue</td>
</tr>
<tr>
<td>Parameter delivery control</td>
<td>Static-rate static-size buffer</td>
</tr>
<tr>
<td>Exception handling control</td>
<td>Dynamic-rate static-size queue</td>
</tr>
<tr>
<td>Control for periodic task</td>
<td>Static-rate static-size queue</td>
</tr>
<tr>
<td>Control for sporadic task</td>
<td>Static-rate static-size queue</td>
</tr>
</tbody>
</table>

Sometimes we meet a need to specify a dynamic-rate variable-size (DRVS) port type in the SPDF task. In the MP3 player task, for example, the required data size of one encoded frame cannot be determined until the MP3 decoder task starts decoding and detects it at runtime. In Figure 9-2, the sample rate \( x \) is determined at runtime by the MP3 decoder itself. The Task model in PeaCE allows dynamic-size ports to specify this case.

![Figure 9-2 Dynamic-rate variable-size port type](image)

NOTE: the port semantics and task semantics only define the behavior not the implementation of the task-model.

9.2 How to design a task-model in PeaCE

This chapter tries to explain the basic notions of task-model by using a simple example. After reading this chapter, you should understand how to define a functional task and a periodic task in the task-model, how to define a task-model using pre-defined tasks and what meaning the target parameters in the task-model have.
As explained in the previous section, task-model supports three types of tasks: functional task, periodic task and sporadic task. Functional task is invoked by data, periodic task by time and sporadic task by interrupt. It is simple to define a functional task. After you describe the algorithm in the CGC domain, you can make it as a functional task by setting the target of the graph "Task-Model" as shown in Figure 9-3. You can place an Xgraph from library/Peace/SPDF/Sink, a super-block input port from library/Peace/Port and make a connection between them. Then you register the task as Galaxy to use it in task-model.

Figure 9-3 Defining a function task

To define a task as a periodic task, what you have to do is only defining a new galaxy parameter named “period” as INT type and setting a positive period value. However, if you set the value of the “period” parameters to –1, the task is handled as a functional task again. Let’s make another task as shown in Figure 9-4; It is a periodic task with a Ramp block (library/Peace/SPDF/Src) and a galaxy output port (library/Peace/Port). And the “period” parameter is set to 1, which indicates that the task is invoked periodically at every one-time unit.

Then we are ready to design first example in BP domain (Figure 9-5) for the task-model which is composed of two tasks. One task with Ramp block sends an increasing integer value periodically (Figure 9-4) and the other task with Xgraph block displays the value when each data arrives (Figure 9-3). You move two tasks from saved libarary to the schematic and make a connection between them.
“period” Parameter is added

BP domain and default-BP target

Figure 9-4 Defining a period task

Figure 9-5 Defining a BP graph with two tasks
To use the Task-model, you should select the BP (backplane) domain and the default-BP target. The default-BP target is in charge of generating codes, compiling the codes and running the executable for functional simulation. In this chapter, we explain default-BP target for functional simulation and other targets in BP domain will be explained in the PeaCE design flow chapter and later chapters in more details. The default-BP target supports the following target parameters:

<table>
<thead>
<tr>
<th>Target parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display?</td>
<td>show a header file which has the data structure of all component tasks</td>
</tr>
<tr>
<td>compile?</td>
<td>compile the generated codes</td>
</tr>
<tr>
<td>run?</td>
<td>run the compiled executable</td>
</tr>
<tr>
<td>debug?</td>
<td>run the compiled executable with gdb debugger</td>
</tr>
<tr>
<td>profile?</td>
<td>show the execution profile of each task</td>
</tr>
<tr>
<td>profile block?</td>
<td>show the execution profile of each block in tasks</td>
</tr>
<tr>
<td>compile options</td>
<td>if it is specified, all compile options in tasks are ignored</td>
</tr>
<tr>
<td>link options</td>
<td>if it is specified, all link options in tasks are ignored</td>
</tr>
</tbody>
</table>

Now run the application. Then you will obtain the codes for each task, a header file with data structure of all tasks, a simulation engine for functional simulation and makefile (Table 9-3). And they are compiled together and executed as shown in Figure 9-6.

Figure 9-6 Results after running the application of Figure 9-5

Table 9-3 Generated files from Figure 9-5

<table>
<thead>
<tr>
<th>Generated codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task110.c, test_sync12.c</td>
<td>task codes</td>
</tr>
<tr>
<td>taskAddressMap.h</td>
<td>header file with task data structures</td>
</tr>
<tr>
<td>makefile_test_BP_0</td>
<td>Makefile</td>
</tr>
<tr>
<td>task.h, sem.h, sem.c, taskSynthesisAPI.c</td>
<td>simulation engine for functional simulation</td>
</tr>
</tbody>
</table>
If you set both “profile?” and “profile block?” flags to YES and execute the task-model, then “log.c, log.h” files are appended to the simulation engine. Then you will get the profile results on the execution times of each task and blocks, as shown in Figure X. It can be used to analyze the complexity of the algorithm and optimize the algorithm.

![Figure 9-7 Profiling results from Figure 9-5](image)

### 9.3 Sporadic task in the Task-model (EXPERIMENTAL)

Sporadic task is an aperiodic task which is triggered by interrupt. For example, a user input from the user interface and a call request from network generates such an interrupt. So a sporadic task should wait for occurrence of an interrupt. Although interrupt and wait routines are usually implemented using OS primitives, how to model such interrupt and wait routines in function simulation without operating system is not clear. So it is now at the experimental stage supporting sporadic tasks in functional simulation.

To define a sporadic task, you should write an interface SPDF block based on the pre-defined form to support sporadic execution as shown in Figure 9-8. Blocking/releasing mechanism is implemented by semaphore OS APIs as shown in Table 9-4. In the `initCode` section, it creates a thread which is blocked on a Linux system call to receive data from external IO. In the main body of the block (`go` section), “wait_interrupt” OS API is called, which releases the task after the thread receives data from the system call. Then it sends the data to the output port. At the `wrapup` section, it generates the wrap-up code to detach the thread.

**Table 9-4 OS APIs implemented in PeaCE for functional simulation of sporadic tasks**

<table>
<thead>
<tr>
<th>Names</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem_t</td>
<td>semaphore variable</td>
</tr>
<tr>
<td>sem_init(sem_t *sem, int initial_number)</td>
<td>initialize a semaphore with an initial number</td>
</tr>
<tr>
<td>sem_v(sem_t *sem)</td>
<td>release a semaphore</td>
</tr>
<tr>
<td>sem_p(sem_t *sem)</td>
<td>acquire a semaphore</td>
</tr>
</tbody>
</table>
Once you write an interface block which supports sporadic execution as the source block of a task, it is handled as a sporadic task.

### 9.4 SPDF extension in Task-model

PeaCE provides two SPDF extensions in the Task-model. One extension is defining a **variable-size dynamic-rate** port to enable a block to consume or produce variable-size and dynamic-rate data. The SPDF model assumes that the number of samples to be consumed is already known at compile-time or at best before execution of the block at run-time (like dynamic construct). But there are cases that we do not know how many samples are needed beforehand in some media applications such as MP3 player. Therefore we need a mechanism of blocking read in the Task model. The variable-size dynamic rate port implements the blocking read mechanism in PeaCE. This type of port is allowed only at the boundary of the task.
The other extension is exception handling. An SPDF task may need to exit during execution if a certain exit condition is met. An error generated during the execution can be an exit condition. If such an exit condition is met, it may need to signal the exception to the top-level BP domain.

In this section, we explain these two extensions with simple examples.

### 9.4.1 Variable-size dynamic-rate port

To use a variable-size dynamic-rate port in a block, you should define the port as MESSAGE type and set some properties at setup function as shown in Figure 9-9. Put “VARIABLE” as the argument of setMessageName method, which indicates that this port is a variable-size dynamic-rate port. The argument of setMessageSize method indicates the maximum buffer size of the port.

```plaintext
input {
    name {input}
    type {MESSAGE}
}
setup {
    input. setMessageName("VARIABLE");
    input. setMessageSize(128); // ring buffer size
}
```

![Figure 9-9 Definition of a variable-size dynamic rate port](image)

After defining the port, you can use the following four OS API functions (Table 9-5) to access the port.

<table>
<thead>
<tr>
<th>OS API functions</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>available($ref(port_name))</td>
<td>wait data arrival and then return available data size</td>
</tr>
<tr>
<td>available_nonblock($ref(port_name))</td>
<td>return available data size immediately</td>
</tr>
<tr>
<td>read_port($ref(port_name), void *data, int size)</td>
<td>wait data with the ‘size’ and copy the data from the buffer to ‘data’ variable</td>
</tr>
<tr>
<td>write_port($ref(port_name), void *data, int size)</td>
<td>write data with the ‘size’ from ‘data’ variable to the buffer</td>
</tr>
</tbody>
</table>

![Table 9-5 OS APIs to access a variable-size dynamic rate port](image)

Figure 9-10 shows a simple example of variable-size dynamic-rate port.

1. Open a new facet and set the domain to BP and the target to default-BP.
2. Reuse two SPDF tasks of Figure 9-5. Instantiate two periodic tasks with the same period (we set the period as 1) to generate Ramp inputs and a display task as a functional task.
(3) Define two new tasks of which each contains only one block inside for simplicity. STPacketize block reads two values and packetizes them into an AddPacket structure as shown in Figure 9-11 (a). Then AddVariable block defines a variable-size dynamic-rate port although it reads two values from the port statically and sends the summation of values (Figure 9-11 b). Then sink task displays the value.

(4) Run the application.

Figure 9-10 An example of using a variable size dynamic rate port
Figure 9-11 Definitions of (a) CGCSTPacketize block and (b) CGCA ddVariable block

Figure 9-12 Execution result
9.4.2 Exception handling

To produce an exception in a block, you can use exit_application() OS API in the main body of the block as shown in Figure 9-13. The template code reads data from the input port and sends data to the network using the write system call. During the execution, if the size of sent data is not equal to the requested size, it means that there is a problem in the network. In that case, it calls the exit_application() function and the task which has the block is terminated with an exception signal.

Figure 9-14 shows an fFSM example that has a concurrent FSM to handle the exception condition. The exception signal generated from the “exit-application” API is delivered to the control FSM task. In the control FSM task, the signal can be handled by “get” script shown below.

<table>
<thead>
<tr>
<th>Script definition</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>get task_name exit internal_event_name</td>
<td>Catch an exception from task_name task and set internal_event_name event active</td>
</tr>
</tbody>
</table>

```c
codeblock (goCode ) {
  int write_size, data_size;

  read_port($ref(input), $ starSymbol (buf ), 8);
  data_size = *( int *)($ starSymbol (buf )+4);
  read_port($ref(input), $ starSymbol (buf )+8, data_size);

  write_size = write($ref( sId), $ starSymbol (buf ), data_size+8);
  if (write_size!=data_size+8) {
    exit_application(); // error condition
    return;
  }
}
```

Figure 9-13 Code template to exit the task at an error condition
9.5 Define task group to control multiple tasks simultaneously

Sometime you may want to control multiple tasks simultaneously in a single state transition in the control task. For example, you may want to change the operation mode of the system, which would terminate the active tasks at the same time. Then you need to define a task group. To define a task group, you create a new schematic for the the task group and set the domain to “BP” and target to “parent”. And place the group of tasks on the schematic. Finally, define the “nodename” galaxy parameter as STRING as shown in Figure 9-15. Figure 9-15 is composed of two tasks shown at the bottom. Then, we register it as a new galaxy using “register galaxy” command. The schematic at the right side shows the registered task group. Then the fFSM can control the task group using the “nodename” of the task group.
Figure 9-15 Task group definition

9.6 Task-model design example

Now you are ready to design a neat example having two task groups and a control FSM in the BP domain. The control FSM task will select one task group between two task groups periodically. Each task group is composed of two tasks. One task has a Ramp block and the other task has a BPXMGraph. Two task groups have different “increment” parameter values in Ramp block. The control FSM task delivers the current value of one Ramp block to that of the other Ramp block when it switches the task group.

(1) We create a source task of the first task group. It consists of a ramp block, a type conversion (float to int), and a galaxy port. We define a period for periodic execution, a nodename to get control signals from FSM, and value1 and globalstate to exchange the current value to the other source task as galaxy parameters. Remind that these are needed for asynchronous interaction between the control fFSM graph and the SPDF graph (see Chapter 8 for detailed explanation).
Then, we set the step state of ramp block as ‘4’ and the value state as ‘value1’. The source blocks of two task groups will have different step values, so we can notice the different slopes of graph.

Let’s define the sink task of the first task group. It is composed of a type conversion block and a BPXMgraph block as shown below.

By registering those two tasks as galaxies in the menu, we make the first task group and add the nodename parameter ‘src1’ to be controlled by the control FSM.
Similarly make the second task group. The second source task is made to have the following galaxy parameters and the step value as ‘1’ in the ramp block as shown below.

By registering the second source task as a galaxy in the menu and getting the sink task from the library, we make the second task group and add the nodename parameter as ‘src2’ to be controlled by the control FSM.

Now make another source task that will generate a periodic event to the control FSM. Set the period to ‘5’ and the level of the const block to ‘1’ as shown below. The task sends the value ‘1’.
The next step is to create the control FSM graph as shown below. Note that ‘processorId’ to be ‘0’ in FSMTaskTarget.

The control FSM gets a trigger signal and activates one task group selectively. When the ‘src1’ state becomes active, script ‘{suspend src1} {deliver ramp1 value1 ramp2 value2} {run src2}’ of ‘src1’ block is executed, which suspends the task group ‘src1’, delivers the value of ‘value1’ state of the ‘ramp1’ task to ‘value2’ state of the ‘ramp2’ task and resume the ‘src2’ task group. In turn, when the ‘src2’ state becomes active, script ‘{suspend src2} {deliver ramp2 value2 ramp1 value1} {run src1}’ of ‘src2’ block is executed, which suspends the task group ‘src2’, delivers the value of ‘value2’ state of the ‘ramp2’ task to ‘value1’ state of the ‘ramp1’ task, and resume the ‘src1’ task group. We can make the following schematic by putting them altogether.
When we execute the application ‘BP_FSM_0’ at $HOME/PEACE_SYSTEMS/CGC, we can acquire the following X graphs.
Chapter 10. PeaCE design flow overview

Author: Soonhoi Ha, Dohyung Kim, and Youngmin Yie

This chapter shows the overview of the overall HW/SW codesign flow in PeaCE environment. The following chapters will explain each design steps in more details.

Figure 10-1 shows the codesign flow of PeaCE, of which the key feature is reconfigurability. The rectangular boxes describe the design steps while the dark rounded boxes indicate the input and output xml files of the design steps. Each design step is modularized so that various algorithms or other CAD tools can be integrated with a minimal effort of wrapper design that translates the xml files. The numberings indicates the sequence of the design steps associated with design taps in the user interface.

Let’s follow the design procedure with a DivX player application.
10.1 Algorithm specification and functional simulation

First, create a new project.

- Select “File→New Project” (not File→New) in the File menu
- Type “DivX” as a project name

Then you will see the newly created project window (Figure 10-2). Note that the design tabs are specified at the top of the window. These design tabs guides you from algorithm specification to cosynthesis. For each design tab, the domain and the target are set automatically.

![Figure 10-2 New project window](Image)

Initially the “sim” design tab is chosen for algorithm specification in the BP domain. Instead of making a new application, copy an existing DivX player application to your new project window.

- Open the ‘schematic/Peace/Demo/BP/DivX_MAD’ schematic
- Copy the schematic to the sim tab of the project schematic
  - Copy all tasks in the ‘DivX_MAD’ schematic to the clipboard by Ctrl+C
  - Paste the selected tasks to the project window by Ctrl+V
- Set the target parameters for the functional simulation (Figure 10-3)
  - compile options  →  ‘-DFPM_INTEL -DCC -DDISPLAY’
  - link options  →  ‘-lm -lx11 -lxext -lpthread -L/usr/X11R6/lib’
Figure 10-3 Design tab for functional simulation

The target parameters for functional simulation are described in Table 10-1.

Table 10-1 Target parameters for functional simulation

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display?</td>
<td>Display generated tasks and an interface file or not</td>
</tr>
<tr>
<td>compile?</td>
<td>Compile the generated code or not</td>
</tr>
<tr>
<td>run?</td>
<td>Run the executable file or not</td>
</tr>
<tr>
<td>debug?</td>
<td>Execute ‘gdb’ program when running the executable file or not</td>
</tr>
<tr>
<td>profile?</td>
<td>Profile the execution time of tasks or not</td>
</tr>
<tr>
<td>compile options</td>
<td>Assign compile options for functional simulation</td>
</tr>
<tr>
<td>Link options</td>
<td>Assign link options for functional simulation</td>
</tr>
</tbody>
</table>

- Set the run count to 1000 in the bottom of the sim tab.
- Click the “Run” button at the bottom of the window. Then,
  - It will generate the task codes, a simulation engine and an interface file:
  - It will compile all and make an executable file, named ‘DIVX_0’.
If the target parameter “run?” were set to “YES”, the executable would be run automatically. Otherwise you should run the executable file at the server for yourself.

- $HOME/PEACE_SYSTEMS/DIVX/func/DIVX_0

It means that PeaCE just generates the code and functional simulation is done separately from PeaCE. If there is any error, you can debug using a conventional debugging program to the generated code. If you set the “run count” with a huge value, then run the executable separately. Or you should wait too long!

You will see the result of functional simulation as follows.

You will see the result of functional simulation as follows.

Figure 10-5 Functional simulation outputs
10.2 Candidate architecture description: arch tab

We specify the candidate architectures to be explored. In this step, we list all processing elements, processor cores and hardware IPs, that are available in the design library. Or we select the platform to be used. The current release does not allow the user to select a specific platform.

If you click the “arch” design tab, you will have an empty schematic where you will draw an abstract view of the target architecture that includes all candidate processing elements. The architecture description at this stage is only to inform the HW/SW partitioning algorithm of which processing elements are available.

To make a simple example, get CPU, ASIC and Bus blocks from library/Architecture in Lib tab. Note that the “processorId” parameter of each block is automatically set when you place it to the schematic. This parameter is used throughout the design procedure to refer the corresponding processing element.

- Set parameters of CPU block:
  - processor: ‘arm926ej-s’ optLevel : ‘2’
- Set parameters of ASIC block and make connections between them
  - processor : FPGA

![Diagram showing an abstract view of the target architecture](image)

(a) (b)

**Figure 10-6 Set parameters of (a) CPU and (b) ASIC blocks**

NOTE: In this release, a separate CPU block should be drawn for each candidate processor: for example, there are four candidate processors available, four CPU blocks should be drawn. If two instances of the same CPU is used, you still need to draw two separate CPU block. It is kind of awkward restriction: sorry for inconvenience. The next release will improve the architecture description capability significantly.
Finally you should click the ‘set architecture’ button at the bottom of the window. Then the architecture information is delivered to the server.

![Image of 'set architecture' button]

**Figure 10-7** Make sure to press the "set architecture" button

### 10.3 Performance estimation for the base platform: estimation tab

After the system behavior and the candidate architecture are separately represented, mapping of the function blocks to the processing elements will be performed based on the performance information (execution time, memory access requirements, area, power, etc.) of each function block on each processing element. In PeaCE this performance information of each function block is managed in a database.

Then a question arises: who measures the performance of each function block? The answer is the author of the block. If a hardware block is created, the author should give all performance information to the database. But for a software block, it is not feasible to estimate the performance before the architecture is determined. Also the software performance depends on the compiler option and the input test vectors.

Therefore PeaCE provides a way to estimate the **software performance** of each function block on each candidate processor core on-the-fly if it is not available in the design library. In this step, we use an ISS (instruction set simulator) of the processor core. And through simulation, PeaCE extracts the information on the CPU execution cycle with unit memory access time and the memory access counts of each block.
NOTE: This design step needs ISS of each candidate processor core. The user may skip this step if all performance estimates of function blocks are already recorded in the library.

NOTE: PeaCE is just a design framework based on library blocks. We recognize that the blocks are the intellectual properties of the authors. So while PeaCE is an open-source program, the block library with performance information can be commercialized.

In this design tab, what you have to do is just to set the “EstProcId” target parameter and to click the “Run” button at the left bottom corner of the window(Figure 10-8). The EstProcId parameter indicates which candidate processing element is the target processor of performance estimation. The default value is 0.

Then, the simulation environment for profiling blocks will be made automatically and compiled to make an executable.

Then run the executable file at the server side

   - $HOME/PEACE_SYSTEMS/DIVX/esti/DIVX_0

This executable is different from that of functional simulation even though it has the same name. If this executable generates the performance profile information of each blocks and updates the database. The execution trace and the profiling results are screen-dumped in .
10.4 Generation of interfacing XML files: analysis tab

This step is to translate the graphical specification into three sets of textual file, which describe the graph topology, block-performance information, and the timing constraints of tasks. Such translation modularizes PeaCE so that
the following design steps do not depend on the PeaCE kernel and PeaCE user interface. A third party tool can be used in PeaCE seamlessly if it can read these textual files.

What you have to do in this design tab is to just click the ‘Run’ button at the left bottom corner of the window(Figure 10-10).

Then, the following three xml files will be created at $HOME/PEACE_SYSTEMS/DIVX directory. The syntax and its meaning of each file will be explained later.

- DIVX.xml - graph topology
- DIVX_TimeCost.xml – performance information of blocks on the processing elements, which is extracted from the database.
- DIVX_mode.xml – timing constraint information for each task

Figure 10-11 shows the dumped screens of the three textual files after graph analysis.
10.5 HW/SW partitioning: partition tab

We perform HW/SW partitioning and component selection at the same time in this design tab. The output is written in a text file that describes the mapping and scheduling of function blocks onto each processing element.

Note that even though PeaCE provides an HW/SW partitioning tool, the user may use another tool thanks to the file interface mechanism. How to use the other tool is explained in a later chapter.

PeaCE provides two partitioning options: automatic and manual. For automatic partitioning, just select ‘Run’ button in this design tab. Then the results will be displayed textually and graphically. The textual output is saved in DIVX_sched.xml file(Figure 10-12) at the $HOME/PEACE_SYSTEMS/DIVX directory and the graphical output is displayed in a Gantt chart form(Figure 10-13).

NOTE: The current implementation of the HW/SW partitioning algorithm can neither handle the piggybacking capability nor dynamic constructs of SPDF model. But it can handle FRDF model, resource sharing between multiple tasks, and static loop structures.
Figure 10-12 (a) HW/SW partitioning result in a textual form and (b) the resultant DIVX_sched.xml file

Figure 10-13 Scheduling result in Gantt chart
To perform manual partitioning, select ‘manual partitioning’ button at the bottom of the window. Then you will obtain a pop-up window that shows a mapping table between each function block to a processing element. If a block may not be assigned to a processing element, the check-box is disabled to prevent you from making an invalid assignment. The table shows the performance information of each block to each processing element to aid the partitioning decision. This information is extracted automatically from the block database.

Now select the appropriate check box if you move the default assignment of the blocks. After manual selection is completed, push ‘run’ button at the left bottom corner of the partitioning schematic to fix the partitioning decision. Then you will get the scheduling results in a textual form and a graphical form as shown in

Figure 10-14 Manual partitioning mode
10.6 Design space exploration of communication: dse tab

After partitioning is made, the next design step is to explore the communication architecture based on the memory trace information from each processing element and the scheduling information. Thus this design step is divided into two sub-steps: memory trace generation and communication architecture exploration. So we explain these sub-steps separately. Now you are getting closer to the low level of design: you need the basic knowledge of software/hardware interface logics to understand the procedure clearly. Detailed explanation can be found later.

NOTE: This step needs a time-accurate simulator for each processing element for HW/SW cosimulation. In this release, we use ARMulator for ARM processor core and ModelSim for hardware logics. If you want to use other components you should define your own target, which is beyond the scope of the user manual. If you are interested in this task, please contact us directly: then we will help you. Later we will prepare the programmer’s manual.

10.6.1 Memory trace generation

Following the scheduling result(DIVX_sched.xml), PeaCE generates the partitioned codes for the processing elements and co-simulates the system to generate the memory traces from the processing elements. PeaCE provides a co-simulation tool based on the virtual synchronization technique, which is fast and time-accurate. At this step, it is assumed that communication architecture is not determined yet.

In this step, you should choose the TraceGen target. And you may need to define the memory map for the code and data sections. There is a default memory map in PeaCE: $HOME/PEACE_SYSTEMS/default.map as shown in Figure 10-16. It defines code and data memory region for each processor as the following format. And it also
defines shared address region for data communication between tasks. Finally architecture dependent regions should be defined for the final synthesis step which will be explained later section.

\[ \text{<start address> \ <size> \ <type> \ [<processor name>] } \]

Now click the ‘Run’ button at the left bottom corner of the window. Then,

- It will generate the cosimulation environment for trace generation, and
- generate the memory map $\text{HOME/PEACE_SYSTEMS/DIVX/DIVX.map}$.

Run the executable file at the server side: $\text{HOME/PEACE_SYSTEMS/DIVX/dse/DIVX_0}$. Then the memory traces for each block will be generated and stored at $\text{HOME/PEACE_SYSTEMS/DIVX/trc}$. Note that this step actually run the distributed cosimulation with component simulators running concurrently as shown in Figure 10-17: ARMulator for ARM processor and ModelSim for ASIC. It means that this step cannot be run without those component simulators.
After completion of the cosimulation, you will be able to see the text files that contain the memory traces for the blocks (Figure 10-18(a)). Note that the memory traces contain the information of time, address, value, etc, as shown in Figure 10-18(b). Since the communication architecture is not determined yet, the relative (not absolute) timing is useful.
10.6.2 Design space exploration for bus architecture

Using the memory traces and schedule information, you can explore the bus architectures. The final bus architecture is recorded in a text file. In this sub-step, you should select the DSE target in the dse tab.

NOTE: As of now the communication architecture we explore in PeaCE is quite restricted in its topology. Later we will allow other communication architectures to be explored. This function is one of the main improvements we are planning in the next release.

If you want to make an initial architecture without traces, set NO for the findOptimal target parameter and click the ‘Run’ button. Then, it will generate a default initial architecture XML file at $HOME/PEACE_SYSTEMS/DIVX/INIT.Divx_initArch.xml. The initial bus architecture is a single bus.

If you performed cosimulation for trace generation in section 10.6.1, change the findOptimal target parameter to YES and click the ‘Run’ button once more. Then, it will invoke a separate process to explore the optimal bus architecture using the memory trace information and the scheduling information. The output result will be saved in an communication architecture XML file ($HOME/PEACE_SYSTEMS/Divx/Divx_arch.xml in this example). Note that you should have traces files at the $HOME/PEACE_SYSTEMS/Divx/trc directory to explore communication architectures.
10.7 Timing cosimulation for HW/SW verification: cosim tab

After the bus architecture is determined, we want to verify the final architecture before synthesis. This step performs time-accurate HW/SW cosimulation for verification. We may use Seamless CVE in this step or our cosimulation tool used in Step 6 but using accurate modeling of bus architecture and OS.

10.7.1 Using PeaCE cosimulation tool

We have developed our own HW/SW cosimulation tool based on the virtual synchronization technique, where component simulators need not be time-synchronized during cosimulation. Instead we obtain the memory traces from the component simulators and perform trace-driven simulation using the communication architecture model and the OS model. While the cosimulation speed is improved two orders of magnitude compared with Seamless CVE, the timing accuracy depends on the modeling of communication architecture and the OS. Since the technical detail is beyond the scope of this manual, just enjoy the speedy cosimulation for now. If you want to use a different OS and different bus from what PeaCE provides, you need to make your own target. If it is the case, please contact us directly to let us help you.

To use PeaCE cosimulation tool, select the ‘Cosimulation’ target and click the ‘Run’ button at the left bottom corner of the window. Then, it will generate the cosimulation environment for performance verification and make an executable at the server: $HOME/PEACE_SYSTEMS/DIVX/cosim/DIVX_0. If you run this executable, you will get the similar results as Figure 10-17 since it basically executes the same cosimulation tool.
10.7.2 Using Seamless CVE for verification

Mentor Graphics Seamless CVE is the most popular design tool for HW/SW verification. One of the main objectives of PeaCE is to integrate a third-party design tool into the design flow of PeaCE seamlessly. To confirm the viability of this objective, this release of PeaCE includes this capability of running Seamless CVE in PeaCE environment. The nice thing of this approach is that the user need not write the interface code between processing components and any wrap-up-code between PeaCE and Seamless CVE. Integration between PeaCE and Seamless CVE is automated. For sure, it was not an easy job from our side! We will support more cosimulation tools in the future to make PeaCE more reconfigurable.

NOTE: In this release, PeaCE doesn’t have complete DivX example for verification using Seamless CVE, because the example use MAD mp3 decoder that embed GNU assembly codes for speed-up. But, we don’t use GNU assembler for verification using Seamless CVE, so we use DIVX_cve example that doesn’t have mp3 decoder instead of DivX example. For the progress of this example, see the chapter 15, Cosimulation.
To use Seamless CVE, select the **Seamless CVE** target and click the ‘Run’ button at the left bottom corner of the window. Then it will generate the cosimulation environment for Seamless CVE, which includes the following files (Figure 10-23).

- **top.vhd**: top schematic for hardware
- **parameters.vhd**: channel information
- **makefile**
Execute Seamless CVE cosimulation environment by running the following file. Then, you will see Seamless CVE running as shown in Figure 10-24

- $HOME/PEACE_SYSTEMS/DIVX/seamless/platform.cve

![Figure 10-24 Screen capture of Seamless CVE triggered by PeaCE](image)

### 10.8 Cosynthesis for prototyping board: cosyn tab (NOT IMPLEMENTED)

This step is to generate the codes for the processing elements in a prototyping board. For a processor core, we generate a C code and a VHDL code for FPGA hardware implementation. This is the final step of the proposed design flow in PeaCE. Since the main job in this step is to generate the interface code that is very target dependent, you need to make your own target for a specific prototyping board. Please contact us directly to let us help you if you are interested.

Generating interface code for a prototyping board is similar to that for Seamless CVE or other cosimulation tool. For our specific prototyping board, called Huins board, we made a specific target, called ‘Huins-Board’ target. If you select the target and click the ‘Run’ button at the left bottom corner of the window, it will generate SW, HW and interface codes for the Huins-Board. The generated files include

- `top.vhd`: top schematic for hardware
- `parameters.vhd`: channel information
- `makefile`

244
Chapter 11. Interface Files

Author: Choonseung Lee, Seongnam Kwon, Kiseun Kwon, Sungchan Kim, Soonhoi Ha

In order to integrate other design tools into the PeaCE environment, we define several interface files between design steps in the PeaCE codesign flow. A design tool can be developed independently and used in PeaCE seamlessly if it communicates with PeaCE through these interface files. And each PeaCE design tool in the codesign flow can be used as a point design tool.

11.1 Graph topology description file: Clustered.xml

11.1.1 Usage

The graph topology description file is obtained from the graph analysis step in the PeaCE codesign flow. For a project `designName`, the file is named as `designName.xml`. This file is used in the partitioning step and in the design space exploration step.

In general an application consists of fFSM tasks and SPDF tasks in the BP domain using the Task model as explained in Chapter 9. The subsequent design steps after functional simulation consider only SPDF tasks while each fFSM task is manually assigned to a processing element. Therefore, the graph analysis step collects the internal task structure of all SPDF tasks in a single graph topology description file.

11.1.2 Syntax description

The hierarchical structure of the tags and attributes used in graph topology description files is summarized in Figure 11-1. The entire architecture starts with the top-level ‘node’ element whose name is set to the “project” name: the top-level ‘node’ is called the project node. A ‘node’ element may have a ‘graph’ element that in turn includes ‘node’ and ‘arc’ elements to make a hierarchical description. Nodes that have no ‘graph’ element inside are atomic nodes. Otherwise, they are called macro-blocks. The descendant child node elements of the project node are associated with the SPDF tasks. The hierarchical structure of ‘node’ elements is displayed in Figure 11-2. Inside the atomic ‘node’ element, there are child elements that specify the following node properties: ‘className’, ‘fullName’, ‘repetition’, ‘gate’ and ‘state’. Connection between nodes is specified with an ‘arc’ element that contains three child elements: ‘from’, ‘to’ and ‘numInitial’.
Figure 11-1 Tags and attributes used in graph topology description files

```
<?xml version="1.0"?>
<node name=' '>
<graph name=' '>
<node name=' '>
<graph name=' '>
<node name=' '>
<className>…</className>
<fullName>…</fullName>
<repetition>…</repetition>
<gate name=' ' type=' '>
<rate>…</rate>
<size>…</size>
</gate>
...
<state name=' '>
<value>…</value>
</state>
...
</node>
...
<arc>
<from node=' ' gate=' '></from>
<to node=' ' gate=' '></to>
<numlnitial>…</numlnitial>
</arc>
...
</node>
</graph>
</node>
```

Figure 11-2 The hierarchical structure of element 'node'

246
A macroblock node has a child element called “repetition” to represent the repetition count of the inside graph. About the repetition count, refer to chapter 5 for details. Suppose that an SPDF graph has three nodes and the schedule becomes 2(AB)C. In this case, A and B are looped, and they are included in a macroblock node in the graph topology description file as illustrated in Figure 11-3. Note that the loop structure is automatically generated and the clustered node has its name ‘bagXX’ where XX is a unique identifier. Therefore the graph description file is NOT the same as the original SPDF graph BUT the clustered graph (for looping) of the original SPDF graph.

```
<node name='bag1'>
    <fullName>…</fullName>
    <repetition>2</repetition>
    <graph name='bag1'>
        <node name='A'>
            ….
        </node>
        <node name='B'>
            ...
        </node>
    </graph>
</node>
```

Figure 11-3 The macroblock example of graph topology description file

### 11.1.3 Description of node

- **name**: The instance name of block, for example, DivX, AviReaderI0, AviParserI4, and so on.

An atomic node has the following child elements as described in Table 11-1

<table>
<thead>
<tr>
<th>Tag name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>The class name of block, not an instance name</td>
</tr>
<tr>
<td></td>
<td>Ex. AviParser, ConstInt, and so on</td>
</tr>
<tr>
<td>fullName</td>
<td>The unique name of the block instance including the hierarchical structure</td>
</tr>
<tr>
<td></td>
<td>Ex. DivX.AviReaderI0.CGCAviReader.AviParserI4</td>
</tr>
<tr>
<td>repetition</td>
<td>The repetition number of the node</td>
</tr>
<tr>
<td>gate</td>
<td>Input or output port description</td>
</tr>
<tr>
<td>state</td>
<td>State(parameter) description</td>
</tr>
</tbody>
</table>
11.1.4 Description of graph

- name: The same to that of the element ‘node’.

11.1.5 Description of gate

- name: The port name of the block
- type: The direction of the port, input or output

The tag ‘gate’ has two child elements, ‘rate’ and ‘size’ as described in Table 11-2

Table 11-2 Child elements of 'gate' element

<table>
<thead>
<tr>
<th>Tag name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rate</td>
<td>The I/O rate of port. The default is fractional rate. Ex. 2/2, 1/1, 1/99, and so on</td>
</tr>
<tr>
<td>size</td>
<td>The data size of port. If this value is -1, this port has a variable data size</td>
</tr>
</tbody>
</table>

11.1.6 Description of state

- name: The state name of the block

This element means the state name of block, and state value is represented in a child element, ‘value’.

11.1.7 Description of arc

The arc information is described with the tag ‘arc’. An ‘arc’ element has three child elements: ‘from’, ‘to’ and ‘numInitial’. These elements are described in Table 11-3.

Table 11-3 Child elements of ‘arc’ element

<table>
<thead>
<tr>
<th>Tag name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>The source gate of the arc node: The sender block name of this arc, this should be a full name, for example, DivX.AviReader10.CGCAviReader.AviParser14 gate: The port name of the sender block, for example, video_out</td>
</tr>
<tr>
<td>to</td>
<td>The destination gate of the arc node: similar to that of ‘from’ gate: similar to that of ‘from’</td>
</tr>
<tr>
<td>numInitial</td>
<td>This represents an initial delay of the arc</td>
</tr>
</tbody>
</table>
11.1.8 An illustrative example

The DivX player application has almost all properties of graph topology description file, but it is too complex and large to be used as an example. So we make a toy example to illustrate the graph topology description file, and Figure 11-4 shows the example.

Figure 11-4. A toy example for describing graph topology description file

The project name of this example is “cluster_test”, and has two tasks, “task1” and “task2”. In the analysis tab of Hae, a graph topology description file, named as “cluster_test.xml”, will be generated if “Run” the application (Figure 11-5). For simplicity, the nodes in task1 are omitted. Note that the block B (UpSample) and C (MRBlackHoleInt) have the multirate 2:3. This makes a local schedule between two blocks as 3B2C. This is represented as a looped structure: in Figure 11-5, the node named ‘bag0’ is used to describe this structure.

```xml
<?xml version="1.0"?>
<node name="cluster_test">
  <graph name="cluster_test">
    <node name="task1I0">
      ...
    </node>
  </graph>
</node>

<node name="task2I2">
  <node name="bag0">
    <fullName>cluster_test.task2I2.CGTask2.bag0</fullName>
    <repetition>3</repetition>
    <graph name="bag0">
      <node name="pt_CGCReceive0">
        <className>Receive</className>
        <fullName>cluster_test.task2I2.CGTask2.CGTask2.pt_CGCReceive0</fullName>
      </node>
    </graph>
  </node>
</node>
```
Figure 11-5 Example of generated graph topology description file

11.1.9 BNF description of the syntax

11.1.10 Regular expression of token definition

{digit} ::= [0-9]
{number} ::= (-)?{digit}+
{string} ::= [^<>]+
11.1.11 BNF description of the syntax

cluster_file ::= "<?xml version="1.0"?>" top_node

top_node ::= "<node name="" string ">">" top_graph "</node>"
top_graph ::= "<graph name="" string ">">" task_node arc_op "</graph>"

task_node ::= null | "<node name="" string ">">" graph "</node>" task_node

task_node ::= null | node node_op

node_op ::= null | node node_op

arc_op ::= null | arc arc_op

graph ::= "<graph name="" string ">">" node_op arc_op "</graph>"

node ::= "<node name="" string ">">" className fullName repetition gate_op state_op "</node>"
     | "<node name="" string ">">" fullName repetition graph "</node>"
className ::= "<className>" string "</className>"

node_op ::= null | node node_op

gate_op ::= null | gate gate_op

state_op ::= null | state state_op

gate ::= "<gate name="" string ">" type="" string ">">" rate size "</gate>"

state ::= "<state name="" string ">">" value "</state>"

value ::= "<value>" string "</value>"

to ::= "<to node="" string ">" gate="" string ">">" </to>"

to ::= "<to node="" string ">">" gate="" string ">">" </to>"

to ::= "<to node="" string ">">" gate="" string ">">" </to>"

numInitial ::= "<numInitial>" number "</numInitial>"
11.2 Block performance description file: TimeCost.xml

11.2.1 Usage

The block performance description file is obtained from the graph analysis step in the PeaCE codesign flow. For a project \textit{designName}, the file is named as \textit{designName\_TimeCost.xml}. This file is used in the partitioning step. Block performance information is obtained from the database. If the performance value seems not correct, check the database of block performance.

In the partitioning step, performance information of each block is needed per processing element, so the graph analysis step collects the performance information of all blocks for all candidate processing elements in a single block performance description file.

11.2.2 Syntax description

The hierarchical structure of the tags and attributes used in the block performance description files is summarized in Figure 11-6. The entire architecture starts with the ‘table’ tag. The ‘table’ element has ‘processor’ child elements and each ‘processor’ element have ‘node’ child elements.

```
<?xml version="1.0"?>
<table>
  <processor name=' ' type=' '>
    <node name=' '>
      <fullName>…</fullName>
      <className>…</className>
      <time>…</time>
      <area>…</area>
      <power>…</power>
    </node>
    ...
  </processor>
  ...
</table>
```

Figure 11-6 The tags and attributes used in the block performance description files
11.2.3 Description of processor

This element describes the processing element name and its type (software or hardware element type) used in the HW/SW partitioning step. Remind that the processing element candidates are set in the architecture description step.

- name: The name of the processing element, for example, *arm920T, arm720T, FPGA*, and so on.
- type: The type of the processing element. There are two types – software and hardware or “SW” and “HW”.

11.2.4 Description of node

This element describes the performance of the blocks on its parent processing element.

- name: The instance name of block, for example, *AviParserI4, RepeatI0*, and so on.

Elements enclosed by the tag ‘node’ for describing performance information are summarized in Table 11-4.

<table>
<thead>
<tr>
<th>Tag name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fullName</td>
<td>The full name of the block instance</td>
</tr>
<tr>
<td></td>
<td>Ex. DivX.AviReaderI0.CGCAviReader.AviParserI4</td>
</tr>
<tr>
<td>className</td>
<td>The class name of the block</td>
</tr>
<tr>
<td></td>
<td>Ex. AviParser, ConstIInt, and so on</td>
</tr>
<tr>
<td>time</td>
<td>The computation time of block in the target processing element. If block type is “HW,” time element is just obtained from the block performance database. If this block type is “SW,” however, this value is obtained by the following equation.</td>
</tr>
<tr>
<td></td>
<td>Time = (Pure CPU execution time) + (memory read count)<em>(memory read penalty) + (memory write count)</em>(memory write penalty)</td>
</tr>
<tr>
<td></td>
<td>Performance information on “Pure CPU execution time” and “memory read/write counts” are obtained from the database.</td>
</tr>
<tr>
<td>area</td>
<td>Area of the block in case of “HW” type processing element.</td>
</tr>
<tr>
<td>power</td>
<td>Power consumption of the block.</td>
</tr>
</tbody>
</table>

In Table 11-4, later three elements – time, area and power are obtained from the performance information in the block performance database. If the information is not available in the database, the value will be “-1”.

254
11.2.5 An illustrative example: DivX player example

Before generating a block performance description file, candidate processing elements should be selected at first. Figure 11-7 shows the example of processing elements in the architecture selection step in PeaCE. In this example, “arm922T” and “FPGA” are selected as software and hardware processing elements respectively.

![Figure 11-7 Example of processing element selection](image)

*Figure 11-7 Example of processing element selection*

![Figure 11-8 The schematic of DivX player example](image)

*Figure 11-8 The schematic of DivX player example*
Figure 11-8 shows the schematic of DivX player example, for simplicity, some block are omitted. Figure 11-9 shows an example of block performance description file – \textit{DIVX\_TimeCost.xml}. In the example file, two processing elements – arm922T and FPGA are described as software and hardware type respectively. In the processing element ‘FPGA’, only ‘FixCBPIDCTBlock’ has a valid value because this block only has valid performance information in VHDL block performance database.

```xml
<?xml version="1.0"?>
<table>
  <processor name='arm922T' type='SW'>
    <fullName> DivX.AviReaderI0.CGCAviReader.AviParserI4</fullName>
    <className>AviParser</className>
    <area>-1</area>
    <power>0</power>
  </node>
  <processor name='FPGA' type='HW'>
    <fullName> DivX.AviReaderI0.CGCAviReader.pt_CGCSendM0</fullName>
    <className>SendM</className>
    <time>80</time>
    <area>-1</area>
    <power>0</power>
  </node>
  ...
</processor>
<processor name='FPGA' type='HW'>
  <fullName> DivX.AviReaderI0.CGCAviReader.AviParserI4</fullName>
  <className>AviParser</className>
  <time>-1</time>
  <area>-1</area>
  <power>-1</power>
</node>
...
<node name='FixCBPIDCTBlockI22'>
  <fullName> DivX.H263FRDivxI3.CGCH263FRDivx.FRDecForDecl0.BlkDecl13.FixCBPIDCTBlockI22</fullName>
  <className>FixCBPIDCTBlock</className>
  <time>200</time>
  <area>22000</area>
  <power>0</power>
</node>
```
11.2.6 BNF description of the syntax

11.2.7 Regular expression of token definition

\{digit\} ::= [0-9]
\{number\} ::= (-)?\{digit\}+
\{string\} ::= [^<>]+
\{null\} ::= \φ

11.2.8 BNF description of the syntax

```plaintext
Time\_cost ::= "<?xml version="1.0"?>" table

table ::= "<table>" processor "</table>"

processor ::= null | "<processor name="" string "" type="" string ">

node ::= null | "<node name="" string ">" performance "</node>"

performance ::= fullName className time area power

fullName ::= "<fullName>" string "</fullName>"

className ::= "<className>" string "</className>"

time ::= "<time>" number "</time>"

area ::= "<area>" number "</area>"

power ::= "<power>" number "</power>"
```
11.3 Mode description file: mode.xml

11.3.1 Usage

The mode description file is obtained from the graph analysis step in the PeaCE codesign flow. For a project `designName`, the file is named as `designName_mode.xml`. This file is used in the partitioning step and in the design space exploration step.

A mode of an embedded system is defined as a set of application tasks running concurrently at a certain system operating condition. For example a PMP (Portable Multimedia Player) can be executed as a DivX player or as a MP3 player at user’s request. A PCS phones can be used as a digital camera or as a MP3 player. And each task may have different system constraints depending on the operation mode. By definition, a system runs in one mode where multiple tasks are running concurrently.

When we optimize the system architecture, we have to consider all modes of operation. If we optimize the architecture for one mode, our decision may be sub-optimal for other modes. Therefore the partitioning algorithm used in PeaCE considers the resource sharing possibility between different modes of operation.

In the proposed system specification in PeaCE, the fFSM task describes the execution status of the SPDF tasks. Therefore, we can identify which task is active at any given time. In the graph analysis step, we analyze the action scripts of the FSM models to identify all possible modes of operation and store that information in this mode description file. It also describes the timing constraints of tasks in each mode of system. Currently supported constraints are the period and the deadline of each task.

11.3.2 Syntax and description

The hierarchical structure of the tags and attributes used in mode description files is summarized in Figure 11-10. The mode starts with the ‘system’ tag. Inside the ‘system’ element, there is the ‘mode’ element in which has the ‘task’ element. The ‘period’ and ‘deadline’ elements are the child elements of the ‘task’ element. The information for period and deadline is obtained from the task parameters which you can define by using ‘add parameters’ menu inside block in ‘Task-Model’ domain.
11.3.3 Description of mode

- name: Name of the mode given from the system status machine

11.3.4 Description of task

- name: Name of task instance
- period: Period of task in nano-second unit. For example 50000000
- deadline: Deadline of task in nano-second unit.

11.3.5 An illustrative example: MMMT Terminal

Figure 11-11 shows a multi-mode multi-task application, called MMMT(multi-mode multimedia terminal). It has three task groups (videophone, MP3 player and VOD player). Each task group has a hierarchical structure and consists of multiple tasks. An FSM task controls dynamic scheduling of task groups and delivers the task parameters received from the user interface task. Two tasks of connection handling for videophone are connected to the control FSM task.

The figure also shows the internal fFSM specification of the control task. Since there are three task groups, it seems that there are three modes of operation. But it may not be true. The number of modes is determined by the FSM description. Therefore PeaCE analyzes the control FSM graph to define a SSM(System Status Machine). How to obtain the SSM is beyond the scope of this manual. MMMT has seven different operational modes as shown in Figure 11-12. It can be automatically constructed by analyzing the control FSM graph. Each state in the SSM indicates an operational mode in the MMMT example. Transition between states shows which control signal changes operational mode of the system.
Figure 11-11 fFSM specification for the control task of the MMMT system
We define the control FSM task as P1, AviParser task as P2, H.263Decoder as P3 and MP3Decoder task as P4. Active task set of S1 is \{p1\}, that of S2 is \{P1, P2, P3, P4\} and that of S3 is \{p1\}. In the S3 state, P2, P3, P4 tasks are at the suspended state, so S1 and S3 lie in different states. Using the System Status Machine, we may perform static analysis of some system properties like liveness and safety, scheduling analysis of each state in SSM, and architecture synthesis for multi-mode multitask. From the SSM, the mode description file is generated: designNam_mode.xml.

### 11.3.6 BNF description of the syntax

### 11.3.7 Regular expression of token definition

\[
\begin{align*}
\{\text{digit}\} & ::= [0-9] \\
\{\text{number}\} & ::= (-)?\{\text{digit}\}+ \\
\{\text{string}\} & ::= [^<>]+ \\
\{\text{null}\} & ::= \phi
\end{align*}
\]
11.3.8 BNF description of the syntax

```
mode ::= "<?xml version="1.0"?>" system

system ::= "<system>" mode "</system>"
mode ::= null | "<mode name="" string ">" task "</mode>" mode
task ::= null | "<task name="" string ">" parameters "</task>" task

parameters ::= period deadline
period ::= "<period>" number "</period>"
deadline ::= "<deadline>" number "</deadline>"
```

11.4 Partition and schedule description file: sched.xml

11.4.1 Usage

The partition and schedule description file is obtained from the partition step in the PeaCE codesign flow. For a project `designName`, the file is named as `designName_sched.xml`. This file is used in the design space exploration step, the cosimulation step and the consynthesis step.

After HW/SW partitioning, blocks are mapped to the processing elements, and scheduled statically. In this file, mapping and scheduling information of blocks is described.

11.4.2 Syntax description

The hierarchical structure of the tags and attributes used in partition and schedule description files is summarized in Figure 11-13. This file starts with the ‘schedule’ tag. The ‘schedule’ element include ‘processor’ elements, and each ‘processor’ element has ‘task’ elements. A ‘task’ element consists of ‘node’ elements mapped to the processor. Each atomic ‘node’ element has ‘className’, ‘fullName’, ‘resourceId’, ‘startTime’ and ‘endTime’ child elements. A ‘node’ element may be a macroblock that has other nodes inside. A macroblock is defined in the graph topology description file (See Section 11.1.2).
11.4.3 Description of schedule

The ‘schedule’ element includes ‘processor’ elements. Two tags – version and algorithm are used to describe the partitioning algorithm used in partitioning step, as summarized in Table 11-5.

<table>
<thead>
<tr>
<th>Tag name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>The version of the partitioning algorithm used in the partitioning step: Ex. 0.2</td>
</tr>
</tbody>
</table>
The ‘makespan’ describes the sum of the scheduled execution time of all tasks. In Figure 11-14, the makespan becomes 14.

![Figure 11-14 Computation of makespan](image)

### 11.4.4 Description of processor

- **name**: The name of a processing element, for example, *arm920T, arm720T, FPGA* and so on.
- **type**: The type of the processing element. There are two kinds of type: SW or HW.

This element has ‘makespan’ and ‘task’ child elements.

### 11.4.5 Description of task

- **name**: The name of a task instance, for example, *AviReaderI0, H263FRDivxI3*, and so on.
- **type**: The type of the element as summarized in Table 11-6. The table is defined in $(PEACE)/src/domains/bp/loopedScheduler/LoopedNode.h". This should be fixed to “-51” for the element ‘task’.

### Table 11-6 Types of node elements

<table>
<thead>
<tr>
<th>Type number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>SEND_NODE</td>
</tr>
<tr>
<td>-2</td>
<td>RECEIVE_NODE</td>
</tr>
<tr>
<td>-11</td>
<td>LOOP_START_NODE</td>
</tr>
<tr>
<td>-12</td>
<td>LOOP_END_NODE</td>
</tr>
<tr>
<td>-13</td>
<td>LOOP_BODY_NODE</td>
</tr>
<tr>
<td>-21</td>
<td>FOR_START_NODE</td>
</tr>
<tr>
<td>-22</td>
<td>FOR_END_NODE</td>
</tr>
<tr>
<td>-23</td>
<td>FOR_BODY_NODE</td>
</tr>
</tbody>
</table>
The ‘task’ element consists of ‘node’ elements mapped to the processing element.

### 11.4.6 Description of node

- **name**: The name of a block instance, for example, `AviParserI4`, `ConstIntI15`, and so on.

An atomic block has the following child elements: ‘className’, ‘fullName’, ‘resourceId’, ‘startTime’ and ‘endTime’. A macroblock has ‘graph’ and ‘repetition’ elements. These elements are described in Table 11-7.

<table>
<thead>
<tr>
<th>Tag name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>The class name of the block, not the instance name</td>
</tr>
<tr>
<td></td>
<td>Ex. AviParser, ConstInt, and so on</td>
</tr>
<tr>
<td>fullName</td>
<td>The full name of the block instance</td>
</tr>
<tr>
<td></td>
<td>Ex. DivX.AviReaderI0.CGCAviReader.AviParserI4</td>
</tr>
<tr>
<td>resourceId</td>
<td>RESERVED</td>
</tr>
<tr>
<td>startTime</td>
<td>The scheduled start time of the block</td>
</tr>
<tr>
<td>endTime</td>
<td>The scheduled end time of the block</td>
</tr>
<tr>
<td>repetition</td>
<td>Repetition count of the macroblock to be sequentially executed</td>
</tr>
<tr>
<td>graph</td>
<td>The name of the graph instance. It is same as in graph topology description file.</td>
</tr>
</tbody>
</table>
11.4.7 An illustrative example: DivX player

In this section, DivX player example is used and the example is described in section 11.1.8. The Generated partition and schedule description file – DIVX_sched.xml is described in Figure 11-15. In this file, the algorithm name and the version are “multi-mode multi-task cosynthesis” and “0.2” respectively. And the makespan is “5349958.”

```xml
<?xml version="1.0"?>
<schedule>
  <version>0.2</version>
  <algorithm>multi-mode multi-task cosynthesis</algorithm>
  <makespan>5349958</makespan>
  <processor name='arm922T' type='SW'>
    <makespan>5349958</makespan>
    <task name='AviReaderI0' type='-51'>
      <node name='AviParserI4' type='0'>
        <className>AviParser</className>
        <fullName>DivX.AviReaderI0.CGCAviReader.AviParserI4</fullName>
        <startTime>0</startTime>
        <endTime>13118</endTime>
      </node>
      ...
      <task name='H263FRDivxI3' type='-51'>
        <node name='bag0' type='-11'>
          <repetition>1</repetition>
          <graph name='bag0'>
            <node name='pt_CGCReceiveM0' type='0'>
              <className>ReceiveM</className>
              <fullName>DivX.H263FRDivxI3.CGCH263FRDivx.pt_CGCReceiveM0</fullName>
              <resourceId>0</resourceId>
              <startTime>13278</startTime>
              <endTime>13325</endTime>
            </node>
            <node name='ConstIntI15' type='0'>
              <className>ConstInt</className>
              <fullName>DivX.H263FRDivxI3.CGCH263FRDivx.ConstIntI15</fullName>
              <resourceId>0</resourceId>
              <startTime>13325</startTime>
              <endTime>13325</endTime>
            </node>
          </graph>
        </node>
        ...
        <node name='ConstIntl15' type='0'>
          <className>ConstInt</className>
          <fullName>DivX.H263FRDivxI3.CGCH263FRDivx.ConstIntl15</fullName>
          <resourceId>0</resourceId>
          <startTime>13325</startTime>
          <endTime>13325</endTime>
        </node>
      </node>
    </task>
  </processor>
</schedule>
```
Figure 11-15 Example of a partition and schedule description file

In the example above, no block is partitioned to the processing element ‘FPGA’ that has the empty macroblock ‘bag0’. This comes from the implementation of partitioning algorithm. Figure 11-16 shows the example of partitioning with looped structure. In Figure 11-16, node 3 is the macroblock with looped structure, and nodes 1, 2, and 3 are mapped to the processing element ‘P0’. Because node 3 has looped structure, the tags associated to ‘bag0’ should be inserted before and after node 3. Furthermore, although the processing element ‘P1’ has no mapped block, the tags associated to ‘bag0’ also should be inserted because next node 4 is not mapped yet. This is because the end time of ‘bag0’ is needed for mapping node 4 to ‘P1’. As a result, after node 4 is mapped to processing element ‘P0’, P1 has the empty macroblock ‘bag0’. 
11.4.8 BNF description of the syntax

11.4.9 Regular expression of token definition

{digit} ::= [0-9]
{number} ::= (-)?{digit}+
{real} ::= (-)?{digit}+\.{digit}+
{string} ::= [^<>]+
{null} ::= ø

11.4.10 BNF description of the syntax

sched_file ::= "<?xml version="1.0"?>" schedule

schedule ::= "<schedule>" version algorithm processor "</schedule>"
version ::= "<version>" real "</version>"
algorithm ::= "<algorithm>" string "</algorithm>"

processor ::= null
   | "<processor name="" string "" type="" string ">" makespan task "</processor>" processor
makespan ::= "<makespan>" number "</makespan>"
task ::= null
   | "<task name="" string "" type="" number ">" node "</task>" task

node ::= null
   | "<node name="" string "" type="" number ">" atomic_node "</node>" node
   | "<node name="" string "" type="" number ">" repetition graph "</node>" node
repetition ::= "<repetition>" number "</repetition>"

graph ::= "<graph name="" string ">" node "</graph>"
11.5 Architecture description file: Arch.xml

11.5.1 Usage

The architecture description file is obtained after the design space exploration step of communication architectures and used in the co-verification and co-synthesis step for generating interface codes and communication models.

NOTE: This file describes the characteristics of architecture components and how the processing elements are connected to the bus architecture. As of now we do not include the peripherals in this file. And we support only a small set of architecture components in the current release. We also cannot express other communication architectures besides bus architectures. These rather severe restrictions will be mitigated in the next release.

For each design, PeaCE generates the following two files:

   designName_initArch.xml, designName_finalArch.xml

11.5.2 Syntax description

The hierarchical structure of the tags and attributes used in architecture description files is summarized in Figure 11-17. The architecture starts with tag ‘topology’. The ‘bus’ and ‘bridge’ elements are child elements of the ‘topology’ element. Inside the ‘bus’ element, the tags for processors and memory exist, which are ‘pe’ and ‘mem’ respectively: ‘pe’ is the abbreviation of processing element, which can be microprocessor, DSP, dedicated hardware block, provided IP, and on. There are some associated attributes for each tag.
11.5.3 Description of bus

- **name**: The name of the bus instance, for example, `bus0`, `bus1`, `myBus`, and so on.
- **type1**: Indicates whether the bus is a pipelined address/data bus or not. Not used currently.
- **type2**: Bus arbitration scheme. It should be one of ‘priority’, ‘round-robin’, and ‘tdma_2level’. (*‘priority’ is only available in the current released version*)
- **dataWidth**: Bit width of data bus. For example, ‘32’ in the case of 32-bit data bus.
- **clock**: Period of bus clock in nanosecond unit. For example, ‘10’ in the case of 100MHz bus.
- **overhead**: The bus protocol overhead associated with various bus specification in terms of bus cycles. For example, there are 2 additional bus cycles after granting bus in AMAB AHB, which are the waits for asserting the HRREADY signal and drive of the start address of a transfer.

11.5.4 Description of processor

- **name**: The instance name.
- **isProc**: Indicated whether the processing element is programmable or not. For example, it should be set to ‘1’ in the case of microprocessor or DSP, and so on. Otherwise, it should be set to ‘0’.

---

**Figure 11-17 Tags and attributes used in an architecture description file**
- type2val: The priority of the processing element for the bus access to a priority bus. For example, if the processing element has the highest priority in the bus, this attribute should be set to ‘1’. Two processing elements may not have the same priority level.

There may be three child elements in the ‘pe’ element, which start with the following tags: ‘code’, ‘data’, and ‘shared’.

- code: This element contains the information on the code memory area if exists. It has the following child element inside:
  - start: The start address of the code memory region in hexadecimal
  - size: The size of the code memory in hexadecimal
  - oriStart: This attribute is required by the DSE module for full architecture exploration. If the DSE module is used as a standalone program, this attribute should be set to the same with the ‘start’ attribute.

- data: This element contains the information on the data memory area if exists. The attributes are similar to those of ‘code’.

- shared: This element is used for describing communication between two function blocks through the shared memory region. The first three elements describe the shared memory region and the additional two elements, ‘src, ‘dest’, refer to the functions blocks mapped to different processing elements.
  - start, size, oriStart: These attributes are similar to those of ‘code’.
  - src: This attribute is the name of function block that initiates communication.
  - dest: This attribute is the name of function block that is the destination of communication.

11.5.5 Description of memory

- name: The name of the instance.
- burstInit: Initialization time for burst transfer in bus clock cycles.
- latency: Access time for reading unit data from a memory. This attribute usually corresponds to memory access time.
- start: The start address of the memory. It should be provided in hexadecimal.
- size: The size of the memory. It also should be provided in hexadecimal.
- memType: Indicates which kind the memory is. Currently, this attribute should be set to ‘SYNC_SRAM’.

11.5.6 The description of bridge

When multiple buses are used, they are connected via a bus bridge. The following attributes are required for describing a bridge.

- name: The name of the instance.
- leaf0, leaf1: Since a bridge connects two buses, these attributes tell which buses are connected to the bridge by the ‘name’ attribute of the bus element.
- leaf0Type2Val, leaf1Type2Val: These are the priorities of this bridge when it plays a role of bus master in the connected buses.
11.5.7 An illustrative example: a single bus architecture

The architecture description files are generated as the result of the DSE step. The file contains an overall architecture determined by the Partitioning and DSE step. The main information provided by the architecture description files includes the bus architecture that connects processors and memories.

Figure 11-18 An illustrative example

Figure 11-18 shows an illustrative example for an H.263 encoder application. H.263 encoder consists of 7 function blocks (ME, DCT, Q, VLC, MC, IDCT, DeQ). Suppose that you already got the partition and DSE result, where the function block ‘ME’ and the function block ‘DCT’ are executed by the processor ‘HW_ME’ and ‘HW_DCT’ respectively in a single bus. The other function blocks are executed in the ‘ARM720T’ processor. Then shared memories are needed for the communications between ARM720T and HW_ME and between ARM720T and HW_DCT. Then these architectural parameters are put together into an architecture description file ‘designName_initArch.xml’ or ‘designName_finalArch.xml’. The resultant architecture description file is captured in Figure 11-19.
11.5.8 BNF description of the syntax

This section provides the BNF expression of architecture description files.

11.5.9 Regular expression of token

\[
\begin{align*}
\text{(digit)} & ::= [0-9] \\
\text{(binary)} & ::= [01] \\
\text{(natural)} & ::= (\text{digit})^+ \\
\text{(string)} & ::= [^<>]+ \\
\text{(hex)} & ::= [0-9A-Fa-f] \\
\text{(hexa_digit)} & ::= (0x[0X])\text{hex}\{\text{hex}\}\{\text{hex}\}\{\text{hex}\}\{\text{hex}\}\{\text{hex}\}\{\text{hex}\}\{\text{hex}\}
\end{align*}
\]

11.5.10 BNF description of the syntax

```
architecture_description_file ::= "<?xml version="1.0"?>" topology

topology ::= "<topology name=" string "">" bus { bus } { bridge } "</topology>"
bus ::= bus_start_tag { pe } mem { mem } bus_end_tag
bus_start_tag ::=
"<bus name='bus' natural
"' type1='" bus_type1
"' type2= '" bus_type2
"' dataWidth= '" natural
"' clock= '" natural
"' overhead= '" natural ">

bus_type1 ::= "pipelined" | "nonpipelined"
bus_type2 ::= "priority" | "round-robin" | "tdma_2level"

bus_end_tag ::= "</bus>"

pe ::= pe_start_tag [ code ] [ data ] { shared } pe_end_tag

pe_start_tag ::= 
"<pe name='" string
"' isProc='" binary
"' type2Val= '" bus_type2
"' dataWidth= '" natural
"' clock= '" natural
"' overhead= '" natural ">

pe_end_tag ::= "</pe>"

code ::= 
"<code start='" hexa_digit
"' size='" hexa_digit
"' oriStart='" hexa_digit
"'></code>"

data ::= 
"<data start='" hexa_digit
"' size='" hexa_digit
"' oriStart='" hexa_digit
"'></data>"

shared ::= 
"<data start='" hexa_digit
"' size='" hexa_digit
"' src='" string
"' dest='" string
"' oriStart='" hexa_digit
"'></data>"

mem ::= 
"<mem name='" string
"' burstInit='" natural
"' latency='" natural
"' start='" hexa_digit
"' size='" hexa_digit
"' memType='" "SYNC_SRAM"
"'>"/

bridge ::= bridge_start_tag address address bridge_end_tag
11.6 Memory Trace Files

11.6.1 Usage

The memory trace files store the memory traces generated from each function block. The memory trace files are generated in the first phase of the DSE step. After partitioning is completed, PeaCE generates the codes for the processing elements and performs the HW/SW cosimulation based on the virtual synchronization technique. The memory trace files are used in the next phase of the DSE step, communication architecture exploration. The DSE step makes use of the memory traces in order to extract the statistics information for performance estimation and in order to apply them directly to the trace-driven simulation. Since the memory trace information is valuable for architecture exploration, PeaCE save them for future use.

The generated file names are in the form of \{Task_Name\}_{Processor_Name}_{Block_Full_Name}.txt to make them unique.

11.6.2 Format of memory traces

Figure 11-20 shows a part of memory traces

<table>
<thead>
<tr>
<th>time n/s</th>
<th>r/w</th>
<th>address</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>307</td>
<td>0</td>
<td>0x000ce9b4 0x06128077</td>
<td></td>
</tr>
<tr>
<td>308</td>
<td>1</td>
<td>0x000ce9b8 0x579f54c0</td>
<td></td>
</tr>
<tr>
<td>309</td>
<td>1</td>
<td>0x000ce9bc 0x0e56ecf1</td>
<td></td>
</tr>
<tr>
<td>310</td>
<td>1</td>
<td>0x000ce9c0 0x1c04cbba</td>
<td></td>
</tr>
<tr>
<td>320</td>
<td>0</td>
<td>0x000a06a4 0x2b45a991</td>
<td></td>
</tr>
<tr>
<td>321</td>
<td>0</td>
<td>0x000a06a8 0x3145f17b</td>
<td></td>
</tr>
<tr>
<td>322</td>
<td>0</td>
<td>0x000a06ac 0x6a83e826</td>
<td></td>
</tr>
<tr>
<td>323</td>
<td>0</td>
<td>0x000a06b0 0x2bb0b765</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11-20 Memory trace format
As you can see, memory traces are composed of consecutive memory accesses sorted in their occurrence time. One memory access information consists of five fields:

- **time**: The time when the memory access occurred, which marked by the component simulator. Its absolute values are not used in the DSE. Important information is the difference of the time stamp of adjacent memory accesses.
- **n/s**: This field indicates whether the memory access is sequential or not. If the value is 0, the memory access is non-sequential transfer. By this field, you can distinguish the boundary between burst transfers. For example, in Figure 11-20, the memory accesses from time stamp 307 to time stamp 10 are made as a burst transfer. Note that the memory access time is always assumed to one cycle when generating the memory traces. Furthermore any kind of overhead is not included in the memory traces.
- **r/w**: This field indicates that the current memory access is read or write. The value of ‘0’ and ‘1’ represent ‘read’ and ‘write’ respectively.
- **address**: This field is the address of the current memory access. It is provided in hexadecimal.
- **data**: This field is the content of the memory access. It is also provided in hexadecimal.

### 11.6.3 Memory trace to actual bus transfer

Using the memory trace information of Figure 11-20, the actual transfer on AMBA AHB will be progressed as displayed in Figure 11-21. After the memory access of time stamp 310 is completed, internal execution time of 9 cycles follows from time stamp 311 to time stamp 319. So no memory accesses are presented on a bus.

![Figure 11-21 AHB transaction by memory accesses from the time 320 to the time 323 in](image)

At the very next cycle to the last of the internal execution, a processor request a bus to execute the burst transfer composed of the memory accesses from time stamp 320 to time stamp 323. If the grant for the request is delayed due to the memory access of previous bus master, the burst transfer of time stamp 320 in the memory traces is also delayed together.

276
Moreover, if the memory has an access latency of multiple bus clocks, the elapsed time of the memory transfer gets longer.

11.7 Memory map description file

11.7.1 Usage

The last interface file to see is the memory map description file, which is generated by the cosimulation step of the trace-generation mode using the default map information. A user may define his/her own memory map information directly modifying this file. This file describes the entire address space of the architecture that you are designing. The naming convention is:

{designName}.map.

NOTE: Unfortunately, the cosimulation (for all modes: trace generation, performance evaluation using both virtual synchronization and seamless CVE) does not support user’s manual modification of memory description file in the current released version. The address translation table for memory management of operating systems can not be made automatically to reflect the address map by modified memory description. This restriction corresponds even in case of single task applications without an operating system since the cache initialization including the page table construction for address translation may be required.

11.7.2 Syntax description

Memory map description file includes three kinds of address map for each processor that is selected in the partitioning step:

- **Code**: This area indicates the code memory area of programmable processors such as microprocessor, DSP, or ASIP. So dedicated hardware blocks are unlikely to have this memory region.
- **Data**: For programmable microprocessors, this region is used for local and some global variables, stack of function call, and heap for dynamic memory allocations. Of course, dedicated hardware blocks may need local data space to store local states or temporary data.
- **Shared**: This is used for communication between different processing elements. One or more shared memory regions can be associated with a processor.

```
Used code or data regions:
{start_address} {size} {region} {processor_name}

Unused code or data regions
0xffffffff 0x0 {region} {processor_name}

Shared regions:
{start_address} {size} {region} {source_func_block_name} {destination_function_block_name}
```

Figure 1-22. Syntax of memory map file
Figure 11-22 shows the syntax of the memory map file. Code and data region have the same syntax distinguishing the cases when they are used or not. To express the used code/data region, 4 attributes are needed.

- **start_address**: Indicates the address where the region begins at. This should be in hexadecimal.
- **size**: Size of the region in hexadecimal.
- **region**: Indicates what kind this region is. This should be one of ‘CODE’ and ‘DATA’.
- **processor_name**: Name of the processor that uses this region.

For the description of an unused region, following 2 attributes should be specified.

- **region**: Indicates what kind this region is. This should be one of ‘CODE’ and ‘DATA’.
- **processor_name**: Name of the processor that uses this region.

Code and data region of each processor should always appear in the description file regardless of whether they are used or not.

Shared memory regions are described with 5 attributes:

- **start_address**: Indicates the address where the region begins at. This should be in hexadecimal.
- **size**: Size of the region in hexadecimal.
- **region**: This should be ‘SHARED’
- **source_function_block_name**: This attribute is the name of the source function block that initiates communication with another processor using this region.
- **destination_function_block_name**: The name of the destination function block that becomes the target of communication initiated by another processor using this region.
11.7.3 An illustrative example

The memory map of Figure 11-18 is used again in this section. The associated memory map description file is captured in Figure 11-23. For example, the processor ‘ARM720T’ has code memory that covers from 0x0000000 to 0x019FFFF, whereas the processor ‘HW_DCT’ has neither code memory nor data memory. The communication between the processor ‘ARM720T’ (executing the function block ‘MC’) and ‘HW_ME’ (executing the function block ‘ME’) is made using the memory in the rage from 0xD000000 to 0x0xDFFFFFF.

11.7.4 BNF description of the syntax

The BNF description of memory map description file is as follows:

11.7.5 Regular expression of token

```
{string} ::= [<>]+  
{hex} ::= [0-9A-Fa-f]  
{hexa_digit} ::= (0x[0X]{hex}{hex}{hex}{hex}{hex}{hex}{hex}{hex}{hex})
```
11.7.6 BNF description of the syntax

| code_region ::= hexa_digit hexa_digit "CODE" string |
| data_region ::= hexa_digit hexa_digit "DATA" string |
| shared_desc ::= hexa_digit hexa_digit "SHARED" string string |
Chapter 12. Performance Estimation

Author: Seongnam Kwon, Youngmin Yi, and Soonhoi Ha

In PeaCE, a system is modeled as a composition of function blocks that can be reused in other systems. PeaCE offers HW/SW partitioning and these blocks can also be units of partitioning. In PeaCE, HW/SW partitioning can be done manually or automatically, and in both cases, performance information of blocks is needed as a basis of decision. For obtaining performance information of blocks, performance estimation method of function blocks will be explained in this chapter. We assume that performance information of hardware IPs is given from IP providers, and if any hardware block exists, its performance can be obtained with commercial design tools. So we will focus on explaining how performance information of SW blocks (CGC domain) can be obtained in PeaCE in this chapter.

12.1 Performance estimation method

In general, it is known that estimating performance of software is difficult. There are many causes that make performance estimation difficult and the followings are some of those.

1. Input data & interference between function blocks
2. Compilers, compiler versions, compiler options
3. Architecture variation that affects the memory access overhead

PeaCE offers an accurate performance estimation method considering these three factors. To consider interference between function blocks, the entire SPDF graph is simulated to obtain the performance of each block. We do simulation using the target instruction-set simulator (ISS) to obtain the performance information, compile the entire application instead of each block, and augment some code to split the blocks. Figure 12-1 shows how application code is augmented. With this approach, interference between function blocks is automatically considered. Moreover, real input data can be used as a test vector, so artificial test vector is not needed. All test vectors are automatically generated from the preceding blocks in the SPDF graph.

![Figure 12-1 Code augmentation for performance estimation](image-url)
If the target software application is large, simulating that application will be very time-consuming. Therefore we store estimated performance information of each block in a database for later reuse. To consider performance variation according to different compilers, compiler versions, and compiler options, performance information is stored in database with a set of different primary keys.

To consider performance variation according to different architectures, we separate performance information into two kinds of information. One is the architecture-independent information and the other is the architecture-dependent one. For example, CPU-execution cycle is the architecture-independent (but processor-dependent) information while memory access time is architecture-dependent. In the case of architecture-dependent information, we change it to an architecture-independent format. One of major architecture-dependent information is memory access time, but we use memory access count as performance information instead of memory access time. Memory access time can be calculated by multiplying the memory access count with the memory access overhead that is determined later after the final architecture is decided.

Performance of entire system can be calculated by the sum of all blocks’ execution times in the system, and this makes the estimation very fast in the partitioning step.

12.2 How to estimate the block performance

In this section, how to estimate the block performance is explained with our DivX player example.

12.2.1 Estimation environment setup

Since PeaCE is platform independent, there are many processors and also many simulators for a processor that can be supported in PeaCE. Currently, however, PeaCE supports only ARM instruction simulator (ARMulator). The interface codes for ARMulator (in ADS 1.2 Linux distribution) and PeaCE are provided in $PEACE/ads1.2/.

The files must be located in paths below:

- $ARMHOME/linux/Source/armulext/flatmem.c
- $ARMHOME/linux/Source/armulext/socksrc.{c, h}
- $ARMHOME/linux/Source/armulext/tracegen.{c, h}
- $ARMHOME/linux/Source/armulext/iomodel.{c, h}
- $ARMHOME/linux/Source/armulext/flatmem.b/linux86/Makefile
- $ARMHOME/linux/Source/armulext/compile

After locating these files in the correct paths, build Flatmem.so by typing the compile script ‘compile’. This shared library will be linked to ARMulator when it is invoked.

NOTE: flatmem.c is part of ADS 1.2 distribution. Thus, we only provide patch for this file. Type ‘cat flatmem.patch | patch --p0’ to generate modified version of flatmem.c

For multi-task execution in a SW component simulator, a certain set of OS APIs (e.g. task preemption) as well as startup code is required. PeaCE provides eCOS (http://sources.redhat.com/ecos) port to ARMulator for this purpose. The eCOS port to arm922T processor in ARMulator is provided in $PEACE/ecos/ecos-arm922T and $ECOS_HOME is set to $PEACE/ecos. Makefile uses this path variable in order to link the application code with the eCOS image.

- $ECOS_HOME/ecos-target/install/lib/: eCOS image and linking script file
NOTE: Changing the target processor requires a port of eCOS to that processor. Currently, PeaCE provides ARM720T and ARM922T ports (or ARM926EJ-S). Designers must copy all the files in $ECOS_HOME/ecos-target/simulator_port to the $ARMHOME/linux/Source/armulext/memInterf and run ‘compile’ script.

12.2.2 Performance estimation

Open the DivX player project file and click “arch” tab, then you can see the window as shown in Figure 12-2(1). To select the target processor, click CPU icon(2) and type the processor name (“arm922T”)(3). Click “Set Architecture” icon(4), then target processor is selected.

Click “estimation” tab(1), then you can see the window as shown in Figure 12-3. Click “run” icon(2), then PeaCE automatically generates software simulation codes for performance estimation of blocks. Enter the iteration count of simulation by typing a number in the edit field next to the “run” icon, whose default value is 10. If the “run” parameter is set to “YES”, PeaCE automatically run the generated code. Otherwise, you can simulate it later: you can run your code in the directory “$HOME/PEACE_SYSTEMS/(project_name)/esti/”:
“$/HOME/PEACE_SYSTEMS/DIVX/esti/” in this case. Move to this directory and type the following command
to run the generated code:

`./DIVX_0`

Figure 12-2 Architecture selection in Hae
If you click any key, simulator will start. During simulation, output image will be shown as Figure 12-4 and database is automatically updated with the simulation result.

Figure 12-3 Performance estimation in Hae
12.3 CGC block performance database

The estimated block performance is stored in a database and can be reused later. In this section, how the database is defined and can be updated will be explained.

In CGC block database, there are many fields that are summarized in Figure 12-5.

<table>
<thead>
<tr>
<th>Number</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Star name</td>
<td>Block name, <strong>primary key</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ex. <strong>Ramp</strong></td>
</tr>
<tr>
<td>2</td>
<td>Target processor</td>
<td>Target processor name, <strong>primary key</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ex. <strong>arm922T</strong></td>
</tr>
<tr>
<td>3</td>
<td>Implementation</td>
<td>Implementation method of block, <strong>primary key</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ex. Fixed point version DCT, Floating point version DCT,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In most cases, this value is “<strong>normal</strong>”</td>
</tr>
<tr>
<td>4</td>
<td>Factor value</td>
<td>This value is performance-dominant value, <strong>primary key</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, 2-input adder is much faster than 32-input adder, than</td>
</tr>
<tr>
<td></td>
<td></td>
<td>block writer can add “input-number” to factor value of adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>
| 5 | Compiler optimization option | Compiler optimization level, **primary key**  
|   |   | Ex. –O2 |
| 6 | Code size | Code memory size of this block |
| 7 | Data size | Data memory size needed to execute this block |
| 8 | Power | Power consumption of this block |
| 9 | Overhead | Reserved |
| 10 | Cost to function call | Reserved |
| 11 | Cycles average | CPU execution time  
|   |   | - average: average execution time, worst: worst case execution time, variance: variance of execution time |
| 12 | Cycles worst | Memory access counts |
| 13 | Cycles variance | This is for cache miss of target processor. So these are target processor-dependent information, not architecture-dependent. |
| 14 | Nonsequential read average | Rom access counts |
| 15 | Nonsequential read worst | If program is stored in ROM area, it will have different access cost of RAM access cost, so these are stored in separate fields. |
| 16 | Nonsequential read variance |   |
| 17 | Sequential read average |   |
| 18 | Sequential read worst |   |
| 19 | Sequential read variance |   |
| 20 | Nonsequential write average |   |
| 21 | Nonsequential write worst |   |
| 22 | Nonsequential write variance |   |
| 23 | Sequential write average |   |
| 24 | Sequential write worst |   |
| 25 | Sequential write variance |   |
| 26 | Rom nonseq. read average |   |
| 27 | Rom nonseq. read worst |   |
| 28 | Rom nonseq. read variance |   |
| 29 | Rom seq. read average |   |
| 30 | Rom seq. read worst |   |
| 31 | Rom seq. read variance |   |

**Figure 12-5 Database fields of CGC blocks**

In Figure 12-5, 1–5 fields are primary keys, so if these values are different, performance value is stored as another tuple. Primary keys are strings and other fields are unsigned integer values. In the current implementation of PeaCE, the performance estimation target of block only provides the CPU execution time and the memory access counts. The remaining fields should be inserted manually.

If you want to update your performance information to this CGC block database, “dbUpdate” utility can help you. At first, edit your performance information as like this text file. In this file, a line corresponds to a block, and space or tab is the delimiter of the fields. Figure 12-6 shows an example of this meta file.
Then run “dbUpdate” utility.

```
dbUpdate db_update_meta_file.txt 0
```

“dbUpdate” utility needs two arguments. The first one is the name of the text file and the second one is updating option – 0: always update, and 1: update only when the information of this block does not exist in the database. Run “dbUpdate” without any arguments if you want to see the options.
Chapter 13. HW/SW Partitioning

Author: Choonseung Lee, Hyunok Oh, and Soonhoi Ha

Hardware-Software partitioning for embedded systems gives us an optimal architecture for a given application by minimizing the cost of the system configuration with satisfying the constraints such as cost, power consumption, etc. PeaCE provides not only automatic partitioning but also manual partitioning capability for a given architecture model.

In the partitioning step, PeaCE solves the following three problems.

1. Component selection: determine the processing elements to be used. For hardware modules, an implementation of each module should be selected.
2. Partitioning and scheduling: partition the input tasks into processing elements and perform a static scheduling to estimate the execution time of tasks.
3. Performance evaluation: evaluate the quality of solution and check whether design constraints are met.

Note that component selection and partitioning are performed at the same time to make the step a “cosynthesis step” rather than a “partitioning” step. The cosynthesis framework implemented in PeaCE is extensible and adaptable. It is a fast heuristic that is made of an iteration loop of three steps that attack the sub-problems separately. It can partition the multi-mode multi-task applications, taking into account resource sharing and loop structure. There is no limitation on the number of processing elements.

A good thing on the cosynthesis framework of PeaCE is that it can be run independently since it uses interface files for input and output. It implies that any other algorithm can be integrated into PeaCE if it uses the same interface files. Our aim is to support various cosynthesis (or partitioning) algorithms in PeaCE. Then the PeaCE can be used as a research environment to compare the various algorithms.

13.1 Inputs to the Partitioning Step

The inputs to the partitioning step are three interface files generated from the previous graph analysis step in the design flow. They can be found in $HOME/PEACE_SYSTEMS/DesingName/ directory (Figure 13-1). The following is the short description of these input files: detailed explanation is in Chapter 11.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DesignName.xml</td>
<td>Represents the input task graph topology after looping transformation.</td>
</tr>
<tr>
<td>DesignName_TimeCost.xml</td>
<td>Represents three attributes, the execution time, power consumption and hardware area of all nodes of a task on each processor (SW) or FPGA (HW). If a node cannot be scheduled on a certain resource, the execution time is INFINITE value (-1). Note that because the partitioning algorithm in PeaCE is only considered to</td>
</tr>
</tbody>
</table>
DesignName_mode.xml

Represents the real-time constraints of given application. The constraints are composed of the task period and deadline. These attributes are used for real-time schedulability analysis. PeaCE can support multi-mode application and each mode has its own constraints.

13.2 Hardware-Software Partitioning in PeaCE Design Flow

The target for the partitioning step is the Cosynthesis target that has the following target parameters as shown in Figure 13-2.

13.2.1

The most important parameters are “multi mode multi task”, “Schedulability” and “Partition”.

289
multi mode multi task

- Default: YES
- Description: It is for multi-task applications. If you want to partition the single task system, you must say “No”.

Partition

- Default: Overlapped
- Description: It represents the method of task partitioning. In PeaCE kernel, two algorithms are implemented, “Independent” and “Overlapped”. Under the “Independent” scheme, all tasks are partitioned independently assuming that each task monopolizes all resources. Then each task tries to minimize the schedule length so that utilization of the processors becomes highly unbalanced: the fastest processor will be heavily utilized. It will not be good for multi-processor systems when more than one task can run concurrently. The “Overlapped” partitioning scheme solves this problem by assigning a weight to each processor based on the current utilization of the processors. We partition the highest priority task first. After a task is partitioned, it computes the utilization of each processor. If the utilization of a processor is high, we lower the performance of the processor before partitioning the next task. “Overlapped” is default value in PeaCE.

![Partition Parameter](image)

Figure 13-3 Partition parameter

Schedulability

- Default: Schedule-Based
- Description: “Schedulability” parameter selects the real-time schedulability analysis method of the given application. There are also two values: Utilization and Schedule-based. The “Utilization” represents the utilization-based schedulability test which is suitable only when a task monopolizes the whole system. It should be paired with “Independent” partitioning. For example, if utilization of two tasks is defined as $U(\text{Task}_1) = 0.6$, $U(\text{Task}_2) = 0.5$, then, the whole system utilization is $1.1$ ($0.6+0.5$), which fails the schedulability test. This method is not directly applicable for multi-processor environment. Therefore you should use the “Schedule-Based” method for multi-processor system. The “Schedule-Based” method is to construct a static schedule of tasks at compile time after the TM (Timed Multitasking) model. If there exists a static schedule that meets all timing requirements, we consider the schedulability test to be passed. This scheme is paired with the “Overlapped” partitioning option. Note that the schedule-based test does not guarantee the timing correctness if the static schedule is not maintained at run time: so it is not safely applicable for RTOS based system.

![Schedulability Parameter](image)

Figure 13-4 “Schedulability” parameter
While the current implementation has two options for the schedulability test, we will include another one in the next release for more general multi-tasking system.

13.2.2 Example: DivX player

In this section, we use the DivX player example and summarize the procedure of the partitioning step.

1. First open the schematic of schematic/Peace/Demo/Dataflow/DIVX then you can see three task nodes as shown in Figure 13-5:

- AviReader task
  - Parse the video and audio data frame from the AVI file.
- H263FRDivx task
  - H263 decoder algorithm specified by the fractional dataflow model.
- MADStream task
  - MAD MP3 algorithm

![Figure 13-5 DivX player design schematic](image)

2. Before run the partitioning step, please make sure that three input files are prepared (Figure 13-1).

3. Click the archi tab and confirm that the number of processing elements is the same as that in the TimeCost.xml file. For the DivX player example, Set the names of processor and ASIC as follows: CPU processor is
“arm926ej-s” and ASIC is “FPGA”. And then click “set architecture” button in “arch” tab (Figure 13-6). If you see the confirmation window message, press OK.

![Figure 13-6 Architecture description](image)

4. Now click the “partition” tab. You have two partitioning options: automatic and manual partitioning. First of all, you should set the partition target parameter. And then just press “Run” button for automatic partitioning. Otherwise, if you press the “manual partition” button, you can see the selectable check-list table representing the candidate processing elements and all nodes be partitioned as shown in Figure 13-7, 14. You can select on HW or SW for each function block. For an infeasible mapping, the check-box is disabled. After you check the list finish your manual partition, click “Set”.

![Figure 13-7 In case of mapping “FixCBPIDCTBlockI22” to hardware only](image)
5. The partitioning result will be showed up in a Gantt chart as shown in Figure 13-8 for the manual partition of Figure 13-7.

Figure 13-8 The Gantt chart in case of mapping “FixCBPIDCTBlockI22” to hardware only

For automatic partitioning option, the Gantt chart of Figure 13-9 will be shown as the partition result for DivX player with “Overlapped” and “Schedule-Based” parameters.

Figure 13-9 The Gantt chart of automatic partitioning

At the same time PeaCE generates the output interface file: DesignName_sched.xml”. It represents how function blocks are partitioned and scheduled for each processing element. The schedule file name is
DesignName_sched.xml is located at $HOME/PEACE_SYSTEMS/DesignName/ directory. If you want to know the description in detail, see the ‘Interface files’ chapter.

![Image](image.png)

Figure 13-10 The generated DesignName_sched.xml file

### 13.3 How to partition for randomly generated graph

Generally, PeaCE needs three input XML files as described in section 13.1. We also support a functionality to partition the randomly generated graph for research purpose. It means that you can use our partitioning algorithm without drawing an SPDF graph in PeaCE. Image how terrible it will be if you should draw the graph in Hae to evaluate the partitioning algorithm!

For this reason, we provide the sweet tool, XML converter, to convert a randomly generated graph or user-defined graph to generate the three input files. If you want to know what is the pre-defined format and how to convert the pre-defined file to XML files, see “Utility” section

After convert the text graph file to the XML files, you **should do** the followings:

- Change each XML file name to the name with DesignName_ as its prefix.
- Copy the XML files to $HOME/PEACE_SYSTEMS/DesignName/ directory.
- Create a project with the DesignName. You do not need anything except architecture selection in this project.
- Must make an initial architecture with the same number of processing element as the number of processing elements used in randomly generated DesignName_TimeCost.xml.
- Click “set architecture”
- Do partition

294
Chapter 14. Design Space Exploration

Author: Sungchan Kim and Soonhoi Ha

After the selection of processors to use is made in the partitioning step, you should pay your attention to how to connect them. We call the network connecting the processors selected in the partitioning step communication architecture. In the design space exploration (DSE) step, users can generate the communication architecture connecting the selected processors. The current version limits the communication architecture only to bus architectures.

The design space means many bus architectures that can be made considering architectural aspects of your design from a certain bus architecture (this is usually called the seed architecture). Such architectural aspects include the number of processors you selected, the partitioning result that affects the characteristics of communications between different processors, and so on. The bus architectures made by considering such aspects are estimated their performance in the DSE step. Two performance evaluation tools are implemented in the DSE module, which are static performance estimation and trace-driven simulation. The static performance is used for choose the qualified bus architectures fast. Then chosen bus architectures are evaluated again more precisely using the trace-driven simulation. So you can select the best bus architecture in terms of performance. You might have to iterate such evaluation process as long as the architecture with more improved performance is obtained. Thus, the DSE might require very long run time according to the various architectural features.

14.1 Inputs and outputs

Unlike other many design steps, the DSE step uses the separated program of which actual executable file is ${PEACE}/src/domains/bp/lib/dse/dse. Therefore, the files of pre-defined format are used for bridging between the DSE step and former (and later) steps integrated inside the kernel of PeaCE.

You should provide four inputs to the DSE, which include the followings:

- DesignName.xml – The schematic information of the algorithm specification in the CGC or VHDL domain.
- DesignName_sched.xml – The result of the partitioning that describes which function block (or star) is assigned to which processor.
- DesignName_mode.xml –
- DesignName.map – This file includes the address map of the selected processors. The address map of each processor includes code memory, data memory, and shared memory (if needed).
- Memory traces – These traces for memory accesses are generated from the cosimulation step.

All inputs should be placed with the same directory where the DSE is carried out in.

If the DSE is completed successfully, two files will be created:
- `DesignName_initArch.xml` – This file describes the most naïve but architecture, *i.e.* the single bus architecture.
- `DesignName_finalArch.xml` – This file may contain the same architecture as `DesignName_initArch.xml` or the optimized bus architecture obtained from the exploration according to the user-defined configurations. Such configuration can be made on either of the GUI (HAE) of PeaCE and the console.

More details on the structures and meanings of input and output files are explained in the section APPENDIX.

### 14.2 Usage scenario 1: as a part of the system level design flow of PeaCE

There are two ways of using the DSE step. First is to use it as a sub-design step incorporated into the entire system level design flow of PeaCE. As a second way of usage, users can run the DSE independent of PeaCE. Let’s take a close look on the first usage of the DSE. As explained in the previous section, users are required to have the correct four input files to begin the DSE step. Before executing the DSE, take a just little bit time to see if all of required input files exist. Let’s assume that the application users are designing now is DivX. Of course, if users have successfully completed all of the design steps prior to the DSE, they would be placed in their right position as listed in the left picture of Figure 14-1. We can see DIVX_mode.xml, DIVX.xml, DIVX_sched.xml, and DIVX.map. The memory traces would be stored in the directory ‘trc’ as shown in the right picture of Figure 14-1. A function block (star) has its own trace file in the form of plain text.

![Figure 14-1. The required input files including memory traces](imageURL)
We are now ready to run the DSE. To execute the DSE, the three followings are should be done as indicated in Figure 14-2.

- Select the tab named ‘dse’ of the schematic windows in the HAE, then the DSE step is prepared to be run.
- Make sure that the radio buttons for domain selection and target selection indicate ‘BP’ and ‘DSE’ respectively.

Finally, the last design property ‘findOptimal?’ should be set either of ‘NO’ or ‘YES’. If it is set to ‘NO’, the DSE generate the single bus architecture only without performing communication architecture exploration. On the other hand, in the case of ‘YES’, the DSE will try to find out the bus architecture with the best performance. In the current release, this design property is only what users are allowed to change.
Let’s run the DSE with the design property ‘findOptimal’ set to ‘NO’. After setting the three configurations, press the button ‘Run’ in the left of the bottom in the schematic window. New terminal titled ‘dse’ will be invoked like Figure 14-3. Since the design property ‘findOptimal’ was set to ‘NO’, the DSE will display the message as shown Figure 14-3 and will be quitted by pressing the enter key.

![Terminal showing DSE output](image)

**Figure 14-4. The resultant two files by setting the design property ‘findOptimal’ to ‘NO’**

Two resultant files (DIVX_initArch.xml, DIVX_finalArch.xml) would be created at the directory $HOME/PEACE_SYSTEMS/designName as listed in Figure 14-4, which are identically same and contain the single bus architecture. For more details on an architecture description file, please refer the APPENDIX.

![Schematic window showing design](image)

**Figure 14-5. Setting the design property ‘findOptimal?’ to ‘YES’ for full architecture exploration**
For next run, set the design property ‘findOptimal’ set to ‘YES’ and leave others the same like Figure 14-5. Then, press the button ‘Run’ in the left of the bottom in the schematic window again.

The same terminal titled ‘dse’ will be invoked again. However, contrary to the previous run that made the simple single bus architecture only, the DSE will progress trying to find out the bus architecture with the best performance scrolling up many messages in the terminal as captured in Figure 14-6. The running time for searching the best architecture is dependent on the complexity of architecture that users are dealing with currently. Such complexity is usually due to how many processors are used and how they communicate with each other. In the case of DivX, it takes about 3 minutes to get the final bus architecture in Xeon 3.0GHz workstation running Linux.

After quitting the DSE, the final architecture description file ‘DIVX_finalArch.xml’ is created. While the initial architecture is composed of only one bus, the final architecture is a dual bus architecture where two selected
processors ‘ARM926ej-s’ and ‘FPGA_1’ are connected to their own local buses and the bus bridge is attached between the buses.

14.3 **Usage scenario 2: as a standalone module independent of the PeaCE**

You can run the DSE step independent of the PeaCE. By utilizing the feature as a standalone module, you can use the DSE to your own development environment. There are no restrictions for utilizing the standalone feature of the DSE as long as the provided inputs are kept their required formats as explained in Section 14.1. If your account is correctly set for the use of the PeaCE in Linux, the executable file of the DSE module is symbolically-linked named as ‘dse’ in the directory $PEACE/bin.linux. The actual file is located in the directory $PEACE/src/domains/bp/lib/dse. First of all, move your work directory to $HOME/PEACE.SYSTEMS/designName if you are not there.

![Figure 14-8. The supported options in the current released version](image)

Then, just type ‘dse’ to see what options exist. You may use the DSE module in three ways including two options as well as no option.

- **‘skip’**: This option is the same functionality exactly to set the design property ‘findOptimal?’ to ‘NO’ in the HAE.
- **‘noit’**: When you specify this option, the DSE module estimates the performance of the architecture with the architecture description file name ‘designName_initArch.xml’. This option is not supported when the DSE is in use as a part of system design flow in the PeaCE.
- **No option**: If you don’t give the DSE any option, it will do full exploration of bus architectures to find out the performance optimum. This is same to set the design property ‘findOptimal?’ to ‘YES’ in the HAE.
Although the input files can be created manually, all necessary inputs were already prepared since you have run the DSE as the part of system level design flow in the PeaCE. The usage of the first option ‘-skip’ is ‘dse –skip designName’ in the command line. For example, by typing ‘dse –skip DIVX’, the DSE would work the same as setting the design property ‘findOptimal?’ to ‘NO’ as you have done in Section 14.2. Of course, the result will be also the same.
The result of trace-driven simulation

The result of Static estimation

Figure 14-11. The report on the performance of the best architecture by full architecture exploration in standalone mode

The second option ‘-noit’ is provided only when the DSE is used as a standalone module. The usage is ‘dse –noit designName’ as captured in Figure 14-10. If this option is given, the DSE module only evaluates the performance of the architecture named ‘designName_initArch.xml’ without further design space exploration. This option might be useful if you want to evaluate the performance of bus architecture in your design through manual architecture exploration. The result of performance estimation is shown in Figure 14-11. As explained above, the DSE module contains two performance estimation tools. For given the architecture ‘designName_initArch.xml’, the DSE provides all results by both performance evaluation tools.

The last usage with any option makes the DSE module execute full exploration on bus architectures by the command line ‘dse –noit designName’.
As introduced in the beginning of Section Chapter 14, full architecture exploration is mainly composed of two parts. The first is the generation of many bus architectures and the repetitive evaluation of their performance as shown in Figure 14-13.

Figure 14-12. Evaluating the performance of an architecture named ‘DIVX_initArch.xml’

Figure 14-13. Architecture exploration is being progressed in standalone mode
After evaluating all bus architectures of consideration, the best architecture is selected and its structure is displayed in the console and the wrap up is made summarizing the overall exploration like Figure 14-14.
Chapter 15. Cosimulation

Author: Youngmin Yi, Youngpyo Joo, and Soonhoi Ha

15.1 Introduction

Hardware/Software cosimulation enables designers to evaluate or verify the system under design before it is manufactured. In PeaCE, cosimulation can be used in different design stages with different purposes; it can be used to evaluate the performance of the system, to verify the functional and timing correctness of the system, or to generate memory trace files that will be used in design space exploration for communication architecture selection.

PeaCE provides its own cosimulation tool based on virtual synchronization scheme as well as the environment to run Seamless CVE, a well-known commercial verification tool, directly in PeaCE. Seamless CVE is very accurate but also very slow. It is for verification of the entire system including all implementations. For other purposes than verification, such as evaluation of the system performance or trace generation of each processing elements, cosimulation with virtual synchronization scheme is more adequate.

The virtual synchronization scheme enhances the speed of cosimulation drastically while employing the same kind of cycle-accurate simulators as in Seamless CVE. The enhancement mainly comes from the fact that time synchronization between simulators is virtually removed through trace-driven simulation with communication architecture model and SW architecture model like OS. Surely the detailed discussion of virtual synchronization is beyond the scope of this user manual. Just enjoy the benefits of fast cosimulation.

NOTE: For HW/SW cosimulation, we need component simulators that would be integrated to the PeaCE design flow through foreign interface facility that the simulator provides. In this release, we support ARMulator for processor cores and ModelSim for hardware IPs. To use other types of processing element, contact us directly to know how to do it.

The detailed explanation for environment setting and how to perform each kind of cosimulation will be discussed in the following sections.

15.2 Cosimulation Environment Structure

Figure 15-1 describes the overall cosimulation environment structure. It comprises the Backplane executable, component simulators along with proper simulation interface for virtual synchronization.
The following is the explanation of the parts of the structure.

1) **Generated codes**: C codes and VHDL codes are automatically generated from the PeaCE.

2) **Interface codes**: OS API definition for multi-task execution and HW interface codes (IF) such as memory interface and synchronization logic are compiled and built with the generated application codes.

3) **BP executable**: It is the simulator engine that schedules the invocation of each component simulator and delivers data to and from the simulators. In case of virtual synchronization scheme, it includes additional simulation models such as communication channel model (i.e., memory, memory controller, arbitration logic) and OS model (i.e., scheduler and task synchronization APIs provided by target OS). In case of Seamless CVE verification, it is simply Seamless CVE engine.

4) **Component simulators**: Cycle-accurate simulators

5) **Simulation interface**: Simulation interface is required to apply virtual synchronization technique to off-the-shelf simulators. It generates traces with timing information and data, and delivers them to the BP executable or delivers data from the BP executable to the simulator. It saves the traces into files or handles I/O requests, if necessary. In seamless CVE environment, it is not required.

### 15.3 Interface Code Generation

C code generation of a block mapped to SW component and VHDL code generation of a block mapped to HW component has been explained in chapter 5 and 6 respectively. To perform cosimulation, complete codes including interface codes must be built in advance. In this section, interface code generation is explained.

#### 15.3.1 Interface Code Generation for Seamless CVE

Interface is very much dependent on the target architecture. Synchronization logic and memory interface logic are obtained from the design library and application dependent interface codes are automatically generated.

**NOTE: Assumptions and Limitations**

1) **SW Parts**:

- It doesn’t support blocking read/write and preemptive scheduling between tasks, because there is no OS environment (especially thread environment) yet. It supports only priority-based non-preemptive scheduling.
- System calls like open(), write() and read() need to be replaced by fopen(), fwrite(), fread(), etc. Take a look at the generated codes of DIVX_cve schematic as an example.
- System calls like printf(), and putc() are supported by Text Output Console.
- I/O handler that handles general I/O requests in Seamless CVE environment will be included in the next release.

2) HW Parts:
- It supports only single bus, AMBA AHB. Its clock frequency is 31.25 MHz. (modifiable)
- It supports only SRAM. Its clock frequency is the same as the bus clock frequency.
- It supports only one ARM926ej-s in system as for now. Its clock frequency is 250 MHz.
- It supports multiple HW IPs, but it has been tested when there is only one HW IP. Its clock frequency is the same as the bus frequency.
- Because of memory controller’s restriction, ROM can be mapped between 0x0 ~ 0x20000000, and RAM can be mapped between 0x40000000 ~ 0x70000000.
- It does not support FRDF due to synchronization controller’s limitation. It supports only single/multi rate synchronization for now. This restriction will be removed in the next release.

1) SW Interface Code

$HOME/PEACE_SYSTEMS/Project_Name/seamless/functionAPI_arm.c
: It contains sequential execution scheduler codes, read/write codes between tasks. C entry function, main() is defined in this file.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/libads/init.s
: Initialization codes. It calls the C entry function.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/libads/irq_handler.c
: Interrupt handler and system calls that handle interrupts for SW applications.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/libads/page_table.s
: Page table for MMU (automatically generated).

$HOME/PEACE_SYSTEMS/Project_Name/seamless/libads/retarget.c, strncasecmp.c
: System Calls like fopen(), fwrite(), strncasecmp() for SW applications.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/libads/scat.scf
: Memory map file for linker (automatically generated).

$HOME/PEACE_SYSTEMS/Project_Name/seamless/libads/system.*
: Channel (between SW and HW) information and related APIs (automatically generated).
2) HW Interface Code

$HOME/PEACE_SYSTEMS/Project_Name/seamless/VHDL/libahb/a926ahb.vhd
: Module that wraps ARM926ej-s CPU core module.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/VHDL/libahb/ahbarb.vhd
: AHB Bus arbiter.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/VHDL/libahb/ahbsyne*.vhd
: Synchronization controller for interrupt based synchronization between CPU and HW IP.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/VHDL/libahb/ahbwrapper*.vhd
: AHB bus wrapper for HW IP. It provides interface for HW IP to access shared and local memories.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/VHDL/libmem/sram32.vhd
: 32bit SRAM module.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/VHDL/libetc/clkgen.vhd
: This module generates several clocks. If you want to change clock frequency, you should modify the value in this file. Clken is a clock enable signal for synchronization between Bus and CPU.

$HOME/PEACE_SYSTEMS/Project_Name/seamless/VHDL/libetc/top.vhd
: Top entity of the entire system.

15.3.2 Interface Code Generation for Virtual Synchronization Scheme

For the cosimulation based on the virtual synchronization scheme, eCOS which is an open-source RTOS made in RedHat is used to provide a multi-task execution environment and a startup code for underlying target processor architecture. Data synchronization between SW and HW tasks are not simulated in the device driver level but modeled in higher abstraction. Therefore, SW-side interface codes such as device drivers and interrupt handler for synchronization logic are not generated.

HW-side interface codes are generated with additional signals for the simulation interface (i.e., FLI: see section 15.4.2). Some signals such as interrupt in synchronization logic are excluded for the same reason; in virtual synchronization scheme, they are not simulated but modeled.
15.4 Cosimulation Environment Setting For Virtual Synchronization

15.4.1 eCOS

For multi-task execution in a SW component simulator, a certain set of OS APIs (e.g., task preemption) as well as startup code is required. PeaCE provides eCOS (http://sources.redhat.com/ecos) port to ARMulator for this purpose. Note that only the minimal necessary OS APIs excluding OS scheduler is used in virtual synchronization scheme since scheduling and task synchronization are performed in the OS model in BP executable. The eCOS port to arm922T processor in ARMulator (see section 15.4.2) is provided in $PEACE/ecos/ and this path must be set as $ECOS_HOME. Designers might build their own eCOS port for other target processor core. In that case, make sure to locate the newly ported eCOS image to the following directory.

- $ECOS_HOME/ecos-target/install/lib/: eCOS image and linking script file
- $ECOS_HOME/ecos-target/install/include/: header files

NOTE: To change the target processor, there must be a port of eCOS to that processor. Currently, PeaCE provides ARM720T and ARM922T ports (or ARM926EJ-S). Designers must copy all the files in $ECOS_HOME/ecos-target/simulator_port to the $ARMHOME/linux/Source/armulext/ and run ‘compile’ script.

15.4.2 Simulation interface

It is required to have proper simulation interface in each component simulator to support virtual synchronization technique. Currently, only ARM architecture is supported as a target processor and ARMulator is used as an cycle-accurate ISS. As an HW component simulator PeaCE supports ModelSim. In this section, inserting simulator interface codes for ARMulator and ModelSim is described.

1) ARMulator interface

The interface codes are provided in $PEACE/ads1.2/ assuming ADS 1.2 (Linux version) has been installed. Designers must copy these simulation interface codes to the proper path. Brief description of each source code along with its path is given as below:

<Modified source code>

$ARMHOME/linux/Source/armulext/flatmem.c
: Flat memory model of ARMulator: modified to call APIs defined in below files

NOTE: This file is part of ADS 1.2 distribution. Thus, we only provide patch for this file. Type ‘cat flatmem.patch | patch –p0’ to generate modified version of flatmem.c

<New source codes>

$ARMHOME/linux/Source/armulext/socksrc.{c, h}
: It contains APIs that establishes socket connection between simulation engine (BP executable) and delivers data to and from it. Also memory trace information is delivered to the simulation engine.

$ARMHOME/linux/Source/armulext/tracegen.{c, h}
: It contains APIs that generate memory access traces
: It contains the APIs that handle I/O requests such as file I/O or audio and display device access. It is very useful when designers cannot use semihosting feature of ARMulator to use other compiler than armcc (e.g., arm-elf-gcc).

<Makefile>
$ARMHOME/linux/Source/armulext/flatmem.b/linux86/Makefile
$ARMHOME/linux/Source/armulext/compile
: Run ‘compile’ script to build new Flatmem.so that contains simulation interface between ARMulator and PeaCE

2) ModelSim interface

HW simulator interface codes are implemented as FLI (Foreign Language Interface) and PeaCE provides ModelSim simulation interface implemented using ModelSim FLI. Since they are implemented as FLI, the interface codes are located where HW IP codes are generated and are built together with them. Brief description of each FLI code along with its path is given as below:

$HOME/PEACE_SYSTEMS/Project_Name/\{cosim|dse\}/VHDL/libahb/ahbwrapper.c
: It establishes socket connection between the simulation engine (BP executable) and ModelSim for data delivery. It also generates memory access traces and delivers them to the BP executable.

$HOME/PEACE_SYSTEMS/Project_Name/\{cosim|dse\}/VHDL/libahb/ahbsync.c
: Subsidiary interface codes modeling synchronization logic.

15.5 Cosimulation for Trace Generation

Even after partitioning has been determined, there remain a lot of design choices for communication architecture selection. In chapter 14, design space exploration scheme for communication architecture selection is explained in detail. The memory traces of processing elements needed in the design space exploration step are obtained through cosimulation.

15.5.1 How to perform cosimulation for trace generation

Do the following to perform cosimulation for trace generation:

i. Select the dse tab and check whether the target is correctly configured as TraceGen. (Figure 15-2)

ii. Click ‘Run’. This cosimulation does not have ‘Run Count’ because, in trace generation, cosimulation is finished automatically when all of the tasks have been executed at least one iteration.
   - SW code is generated as C codes in $HOME/PEACE_SYSTEMS/Project_Name/dse/
   - HW codes are generated as VHDL codes in $HOME/PEACE_SYSTEMS/Project_Name/dse/VHDL.

iii. To start cosimulation, type the name of BP executable, Project_Name_0 in $HOME/PEACE_SYSTEMS/Project_Name/dse/. In this example, type DIVX_0 and cosimulation will start. (Figure 15-3)
Figure 15-2 Running cosimulation for trace generation

Figure 15-3 Cosimulation for trace generation
At first, BP executable or simulation engine will invoke the I/O server that handles I/O requests from the simulators. Then, it will invoke each simulator in turn and waits for the user to press any key to start execution. After completion of cosimulation, it also waits for the user to press any key to close all simulators.

15.5.2 Generated trace files

As a result of cosimulation, traces for each block in the system are generated and saved in separated files. The path and the name of the file is made as follows:

$HOME/PEACE_SYSTEMS/Project_Name/trc/full_name_of_a_block.txt

The format of the trace file is explained in chapter 11.

15.6 Cosimulation for Evaluation of System Performance

To enhance the speed of cosimulation while employing cycle-accurate simulators, a technique called virtual synchronization has been adopted in PeaCE. In this scheme, communication architecture and SW architecture like OS is not actually simulated but modeled. This cosimulation is useful for verifying component IPs other than communication architecture and for evaluating the whole system in a short time.

15.6.1 How to perform cosimulation for system evaluation

iv. Select the cosim tab and check whether the target is correctly configured as Cosimulation. (Figure 15-4)
v. Enter iteration count of a task with the minimal period in the system in the text box next to ‘RUN’ button.
 vi. Click Run.
   - SW code is generated as C codes in $HOME/PEACE_SYSTEMS/Project_Name/cosim/
   - HW codes are generated as VHDL codes in
     $HOME/PEACE_SYSTEMS/Project_Name/cosim/VHDL.
 i. To start cosimulation, type the name of BP executable, Project_Name_0 in
    $HOME/PEACE_SYSTEMS/Project_Name/cosim/. In this example, type DIVX_0 and cosimulation
    will start just like in Figure 15-3

At first, BP executable or the simulation engine will invoke the I/O server that handles I/O requests from the
simulators. Then, it will invoke each simulator in turn and waits for the user to press any key to start execution.
After completion of cosimulation, it also waits for the user to press any key to close all simulators.

**15.7 Coverification**

Designers can verify the whole entire system, or virtual prototype, through highly accurate but very slow
cosimulation such as Seamless CVE directly in PeaCE.

### 15.7.1 How to perform coverification with Seamless CVE

DIVX_cve is an example for coverification with Seamless CVE. This example is a modified version of DivX
example in which mp3 decoder task is removed.

**NOTE: Preconditions**

1) default.map:

   This file is needed in $HOME/PEACE_SYSTEMS/Project_Name/ directory. This file defines the
   system memory map. Its recommended form is in the Table 15-1 below. The third or the fourth
   column means the device name, and the first column means its address, the second column means
   its size.
2) Opening a file:
Designers must load a input file to a specific memory region before it is read. In DIVX_cve example, input file for Avi file parser is assumed to be located in 0x06000.0000. This can be done using armsd command, getfile. For example, type “getfile friend.avi 0x6000.0000” in armsd.

3) Compiler option of H.263 decoder task is set with the value, ‘-D_READ_OPTIMIZE’.

![Figure 15-5 Running coverification](image-url)
Select the cosim tab and check whether the target is correctly configured as Seamless CVE. (Figure 15-5)

Click Run.

- SW codes are generated as C codes in $HOME/PEACE_SYSTEMS/Project_Name/seamless/.
- HW codes are generated as VHDL codes in $HOME/PEACE_SYSTEMS/Project_Name/seamless/VHDL/.

If ‘Compile?’ and ‘Run?’ parameters are set to ‘YES’, Coverification will start automatically. (Figure 15-6)

In the armsd command prompt, type “getfile friend.avi 0x6000.0000” where friend.avi is the input file name.

Type “go” in armsd command prompt to continue the execution.

Figure 15-6 Coverification with Seamless CVE

15.7.2 Manual compilation and running for Seamless CVE

Set ‘Compile?’ and ‘Run?’ parameters to ‘NO’.

Click Run. Then, SW and HW codes and makefiles are generated.

SW codes can be compiled by typing ‘make’ in $HOME/PEACE_SYSTEMS/Project_Name/seamless/.

HW codes have more than two compile depths in PeaCE, so it’s not recommended to compile manually.

Run the coverification by typing the generated script ‘./platform.cve’ in $HOME/PEACE_SYSTEMS/Project_Name/seamless/
Chapter 16. C++ and SystemC code generation (Experimental)

Author: Seongnam Kwon and Soonhoi Ha

In the CGC domain, C is the target language of the generated software code. C is one of the most famous and powerful high level language, but many engineers use another high level language to design, simulate, and implement systems. So someone may want PeaCE to generate software codes in another language. For this reason, we are developing two experimental targets (C++ code generation target and SystemC code generation target) in this chapter. In particular SystemC is adopted as the initial system specification language in many system level design tools. Therefore PeaCE may be used as a front-end tool to those design tools if PeaCE generates SystemC code from the Task model specification.

16.1 C++ code generation target

The C++ code generation target generates a class for each task in PeaCE. Since variables defined inside a class are only visible in the class, name conflict between blocks will be resolved much easier compared with C code generation. Moreover if there are multiple task instances of the same task in a system, they can share the class code with different instances, and this can make the binary code size much smaller.

If block should be rewritten for the C++ code generation, redundant effort and time must be needed. In order not to do that, in PeaCE, generating C++ code is automatically performed with a little modification of blocks using “CGC star parser.” This is also used in SystemC code generation target.

16.1.1 CGC Star parser

To translate a C code block into a C++ code block automatically, block structure should be explored, which is what the “CGC star parser” does. The “CGC star parser” reads and analyzes C code streams, and stores the structure of code streams in a data structure. C++ code generation target reconstructs the code streams into C++ code streams as shown in Figure 16-1. Note that this target is incomplete and has some limitation. Most critical limitation is that “#ifdef...#endif” is not supported. The current “CGC star parser” cannot analyze nested “#ifdef...#endif” structure, so ignore it. If you want to use this target, you should eliminate “#ifdef...#endif” statements inside any code stream in your block definition. And other rare coding patterns cannot be analyzed by the “CGC star parser.” This will be corrected in the following version.
16.1.2 Running C++ code generation target in Hae

At first, open the schematic “schematic/Peace/Demo/CGC/CPP/DivX/DivXPlayer_cpp”. This example consists of three tasks – AviParser, H.263 decoder, and mp3 decoder. And each task is assigned “CGC-CPPGeneration” target in the “CGC” domain as shown in Figure 16-2. The top schematic of DivX player example should be assigned the “default-BP target in the “BP” domain as shown in Figure 16-3. Other parameters are the same as in the original DivX player example. In short, to generate a C++ code, each task should be assigned the “CGC-CPPGeneration” target and the top schematic should be assigned the “default-BP” target.
Figure 16-2 Schematic of the DivX player example - CGC-CPPGenerationTarget

Figure 16-3 Top schematic with the default-BP target

318
Then just click the run icon, then PeaCE generates C++ codes in the directory “${HOME}/PEACE_SYSTEMS/(project_name)/func/”. In this example, codes will be generated in the directory “${HOME}/PEACE_SYSTEMS/DivXPlayer_cpp/func/”. Its structure is similar to “CGC-TM” target.

16.2 SystemC code generation target

For fast HW/SW cosimulation, the transaction level modeling(TLM) is becoming important in system level design. SystemC is a C++-extension language developed to model the hardware and the software blocks with a unified language. In TLM, an hardware block is modeled in SystemC for fast simulation speed with high abstraction level.

In this version, PeaCE generates not the TLM simulation code but the untimed functional simulation code. The TLM simulation code will be generated in the next version. A model is generated for each task, and the top model is generated in the top schematic to bind the models of tasks. To communicate between models, sc_fifo is used as channel.

The usage of this target is same as the “CGC-CPPGeneration” target except that the systemC library should be installed.

16.2.1 Running SystemC code generation target in Hae

Open the schematic “schematic/Peace/Demo/CGC/SystemC/DivX/DivXPlayer_systemC” as shown in Figure 16-4. This example is same as in the C++ code generation target, but the target of each task should be set to the “CGC-SystemC” target in the “CGC” domain.

Just click the “run” icon, then PeaCE generates SystemC codes. These codes will be located in the directory “${HOME}/PEACE_SYSTEMS/(project_name)/func/”: in this case, this directory will be “${HOME}/PEACE_SYSTEMS/DivXPlayer_systemC/func/”.

Note that the name of the makefile for generated codes is fixed as “Makefile.linux”, not the name of the project file, and the name of the executable binary is also fixed as “run.x”. If you want to run this code later, you should execute “run.x” or rename it for later use.
16.2.2 Structure of generated SystemC codes

Figure 16-5 shows the structure of the generated SystemC code for the DivX player example. Each task is generated as a SystemC model. In this example, three tasks - AviParser, H.263 decoder and MP3 decoder - are generated. Communications between tasks are performed through the “sc_fifo” channel. Each task has “sc_fifo_in/out” ports to be connected to sc_fifo channels. The “Top” model initiates SystemC models of tasks.
and channels, and when simulation starts, each model runs in parallel. If one model cannot write data to another model, it is blocked until channel is available and data can be written. In the ‘read’ case, it is same as ‘write’. If one model cannot read data from another model when data is needed, this model is blocked until data can be read. If channel is too small, deadlock can occur. But it is not detected automatically. It is up to the system designer.

<table>
<thead>
<tr>
<th>AviParser.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;systemc.h&gt;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>class AviParser : public sc_module</td>
</tr>
<tr>
<td>private:</td>
</tr>
<tr>
<td>int member_functions();</td>
</tr>
<tr>
<td>int member_var;</td>
</tr>
<tr>
<td>public:</td>
</tr>
<tr>
<td>int preinit();</td>
</tr>
<tr>
<td>int init();</td>
</tr>
<tr>
<td>int go();</td>
</tr>
<tr>
<td>int wrapper();</td>
</tr>
<tr>
<td>sc_fifo_out&lt;char&gt; outPort0;</td>
</tr>
<tr>
<td>sc_fifo_out&lt;char&gt; outPort1;</td>
</tr>
<tr>
<td>sc_fifo_out&lt;char&gt; *outPort[2];</td>
</tr>
<tr>
<td>void InitParameters()</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>outPort[0] = &amp;outPort0;</td>
</tr>
<tr>
<td>outPort[1] = &amp;outPort1;</td>
</tr>
<tr>
<td>preinit();</td>
</tr>
<tr>
<td>init();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>SC_CTOR(AviReader_systemCl0)</td>
</tr>
<tr>
<td>: outPort0(&quot;outPort0&quot;), outPort1(&quot;outPort1&quot;)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>AviReader_systemCl0::InitParameters();</td>
</tr>
<tr>
<td>SC_THREAD(go);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>int write_port(int chid, char * data, int size);</td>
</tr>
<tr>
<td>int read_port(int chid, char * data, int size);</td>
</tr>
<tr>
<td>int available(int chid);</td>
</tr>
<tr>
<td>int init_port(int taskId, int portId);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AviParser.cc</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &quot;AviParser.h&quot;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>// member function definitions....</td>
</tr>
<tr>
<td>void AviParser::go()</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>//main codes....</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>int AviParser::write_port(int chid, char * data, int size)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>if channel doesn’t have enough buffer, wait....</td>
</tr>
<tr>
<td>write data to channel (chid is port number)</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>int AviParser::read_port(int chid, char * data, int size)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>if channel doesn’t have enough data, wait....</td>
</tr>
<tr>
<td>read data to channel (chid is port number)</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>int AviParser::available(int chid)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>return available data number;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>int init_port(int taskId, int portId)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>initialize port</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

**Figure 16-6 Generated SystemC pseudo codes of AviParser**

Figure 16-6 shows generated SystemC pseudo codes of AviParser. In the header file, output ports (outPort0, outPort1) are declared, and in function “InitParameters()”, they are mapped to the pointer array “outPort[]”. This pointer array enables the port to be accessed with index number in the generated code. All functions about port I/O (read_port, write_port…) access I/O ports with this index. Main codes of AviParser are in the function “go()” and this function runs as like SC_THREAD type in SystemC codes. For this purpose, in “AviParser.cc”, main codes are wrapped within “while(1) {...}” statements.

AviParser writes data to other tasks with “write_port()”. In function “write_port()”, if there is not enough buffer space to write into, the model is blocked until buffer is available. In contrast, in the function “read_port()”, if there are not enough data to read in the buffer, this model is blocked until data is available. Function “available()”
returns the count of available data. So if “available()” is used before “read_port()” is called, remained data can be read without blocking.

```cpp
#include “AviParser.h”
...
class top : public sc_module
{
public:
    AviReader_systemCI0 *AviReader_systemCI0_object;
    H263Decoder_systemCI3 *H263Decoder_systemCI3_object;
    mp3_task_systemCI5 *mp3_task_systemCI5_object;
    sc_fifo<char> channel_0;
    sc_fifo<char> channel_1;

    virtual void InitParameters() {} 
    virtual void InitInstances();
    virtual void DeleteInstances();

    SC_CTOR(top)
    : channel_0("channel_0", 10000),channel_1("channel_1", 10000) 
    {
        top::InitParameters();
        top::InitInstances();
    }

    virtual ~top() {
        top::DeleteInstances();
    }
};
void top::InitInstances() {
    AviReader_systemCI0_object = new AviReader_systemCI0("AviReader_systemCI0_object");
    H263Decoder_systemCI3_object = new H263Decoder_systemCI3("H263Decoder_systemCI3_object");
    mp3_task_systemCI5_object = new mp3_task_systemCI5("mp3_task_systemCI5_object");
    AviReader_systemCI0_object->outPort0(channel_0);
    AviReader_systemCI0_object->outPort1(channel_1);
    H263Decoder_systemCI3_object->inPort0(channel_0);
    mp3_task_systemCI5_object->inPort0(channel_1);
}
void top::DeleteInstances() {
    delete all instances....
}

int sc_main(int ac, char *av[]) {
    top *top_object = new top("top_object");
    sc_start(-1);
    delete top_object;
}
```

Figure 16-7 Generated SystemC pseudo code of the top model
Figure 16-7 shows the generated SystemC pseudo code of the top model. In this model, models of tasks are declared. In the “InitInstances()” function, these models are instantiated and channels are connected to the ports. The top model is instantiated in “sc_main” and starts execution.

The detailed communication protocol is up to the SystemC library in this version. But in the next version, TLM simulation codes will be generated.
Chapter 17. Cosynthesis
Chapter 18. StarEdit

Author: Kyungjoo Oh and Soonhoi Ha

StarEdit is a tool to aid the creation of a new atomic block, also called a ‘star’. Making a new block requires a very specific knowledge about PeaCE star file conventions. But with StarEdit, some knowledge about programming in C is enough to make your own star. This user manual covers how to create and modify star files using StarEdit. StarEdit is a plugin of jEdit (http://www.jedit.org), so if you are not familiar with jEdit, you may need the jEdit User Manual with this.

18.1 Installation

18.1.1 Tarball Installation

Download the tarball file from our website; http://peace.snu.ac.kr.

It contains all about using StarEdit in jEdit. jEdit 4.2 final, StarEdit.jar, star syntax highlighting file, and some other plugins are included in this tarball. Uncompress this file on your local system, and launch jEdit. But if you do not have Java Runtime Environment on your system, it is required to install JRE or JDK first.

Check the URL: http://jedit.org/index.php?page=quickstart

18.1.2 Individual Installation

Please follow the steps below.

Install jEdit

You are advised to check the URL; http://jedit.org/index.php?page=quickstart

jEdit.org site has a huge information about jEdit. The final version of jEdit is 4.2. StarEdit plugin is fully compatible with this version of jEdit.

Download the StarEdit.jar file

‘StarEdit.jar’ file can be downloaded from our website; http://peace.snu.ac.kr.

You need to copy StarEdit.jar file into ‘jars’ directory of jEdit root to use StarEdit,. By default, jEdit root directory is “C:\Program Files\jEdit” on Windows machine.

Copy Syntax highlighting file

‘Star’ syntax highlighting file in xml format can be downloaded from our website; http://peace.snu.ac.kr.
You need to download the following two files.

- star.xml: http://peace.snu.ac.kr/…………
- catalog: http://peace.snu.ac.kr/…………

After downloading these files, copy them into the ‘modes’ directory of jEdit root.

Where is source code available

StarEdit source code is available on our website http://peace.snu.ac.kr/.

18.2 Starting StarEdit

18.2.1 Starting jEdit

Now you have just completed the installation of jEdit+StarEdit to your system. Now you are ready to start StarEdit. On the most systems, you just need to run jEdit and activate StarEdit plugin inside jEdit, by selecting jEdit from a menu or double-clicking jEdit icon. The next section is about how we can activate the StarEdit plugin.

18.2.2 Activating StarEdit Plugin

If ‘StarEdit.jar’ file is in ‘jars’ directory under jEdit root directory, you can see the main window like Figure 18-1 by selecting Plugins>StarEdit.

![Figure 18-1 Activating StarEdit Plugin](image_url)
Selecting **StarEdit>Dock** instantiates a new StarEdit plugin window as shown in Figure 18-2 (a). By default, StarEdit is shown in a floating window. This window can be docked using the commands in the top-left corner popup menu. See the Figure 18-2 (b). **Dock at Left** makes StarEdit plugin window be docked at the left corner of jEdit window.

![Figure 18-2 StarEdit plugin window](image)

And we need “File System Browser” to open an existing file on the local or remote machine. **Utilities>File System Browser** displays the file system browser. It can be docked the same way like StarEdit window, as shown in Figure 18-3. If you want more information about “File System Browser”, see the jEdit User Manual.
Now, StarEdit and “File System Browser” are shown in the docked windows of jEdit like figure below. Selecting the left tab displays the corresponding window.
Figure 18-4 StarEdit and “File System Browser” are docked at left corner

Using “File System Browser”, you can navigate the file system and find the file you want to edit; specially ‘star’ file. The extension of ‘star’ file is .pl. After selecting the .pl file, click the tab “StarEdit” that is located at the docked windows region. Then StarEdit plugin parses the selected star file. See Figure 18-5.
18.3  StarEdit Basics

18.3.1 Interface Overview

Tree view, code view, and wizard view are three main parts of StarEdit.

When StarEdit reads a star file, it parses the file contents, checks the elementary syntax, and shows the tree view of the star section hierarchy in the StarEdit window. See Chapter 5.2 for more information about the block(star) definition in .pl format.

Code view is an editing panel in jEdit, inherently a part of jEdit. In the code view, you can edit the file as you do in other text editors. See Figure 18-6.

But user can change the code view to wizard view; see Chapter 18.3.5. Some specific star sections such as ‘input’, ‘state’, or ‘codeblock’ have its own wizard view. By filling the values in a wizard form, user can do star definition more easily than editing the code view. See Figure 18-7.

Status pane shows the parsing result in a very simple form. Input port number, output port number, state number, and codeblock number are reported from left to right.
Figure 18-6 StarEdit interface

Figure 18-7 Wizard view
18.3.2 Tree View

The tree view gives user the block definition file (.pl file) structure at a glance. It shows what sections this block definition is composed of. The nodes on the tree named ‘input’, ‘output’, ‘state’, and ‘codeblock’ are the collection of ‘input’, ‘output’, ‘state’, and ‘codeblock’ sections in the block definition file respectively. So using mouse click or arrow keys, you can expand it to display its child nodes. Other important feature of the tree view is that it is a map of the star file. In other words, you can navigate the sections in the star file by selecting the node you want to see with the tree view.

Clicking on node or background in the tree view invokes the context wizard popup menu. See Figure 18-8.

![Figure 18-8 Popup menu on tree view: (a) click on state, (b) click on background](image-url)
18.3.3 Code View

Because the code view is a main part of jEdit, you had better refer to jEdit User Manual to use it thoroughly. jEdit has many good features to edit files: for example, select all the text body by ‘CTRL+a’. ‘CTRL+i’ is the short key for ‘indent selected lines’. There are many short keys worthy of memorizing.

As explained earlier, a user can easily construct the star code by the wizard view. But the wizard view creates just a template of the block definition. So the user is usually requested to edit the template code in the code view manually. **Code view and Wizard view help each other.** After you modify the text in the code view, save it by short key CTRL+s. If save event occurs, StarEdit reparses the file automatically.

18.3.4 Wizard View

All star sections do not have an associated wizard view. For example the ‘Initcode’ section is generated automatically when a user adds a new codeblock using the wizard view. But if you want to insert more code in the ‘Initcode’ section, you should work on the code view since there is no corresponding wizard view to the ‘Initcode’ section. The sections that have wizard view include ‘input’, ‘output’, ‘state’, and ‘codeblock’.

Refer to Table 18-1 for more information.

18.3.5 Switching Code / Wizard View

This section has very useful information. Code/Wizard switching at the right time is an alpha and omega in using StarEdit. There are four ways to do this.

**Double click on code / wizard view**

This is the most intuitive and recommended way to switch the view. For example, if you want to edit ‘input port’ in the wizard view, double click on the ‘input port’ section in the code view. Then, the code view will be changed to the wizard view as illustrated in Figure 18-9. And double clicking on the background of the wizard view will make you return back to the code view. But double click on the section range in code view that is not associated with any wizard has no effect: ‘initcode’ section is an example.
Double click on the tree view
Double click on background in the tree view toggles between the two views. When the code view switches to the wizard view, a wizard view appears depending on which node (or section) get focused in the tree view.

Using Short key
It is the same as ‘Double click on the tree view’ except that ‘t’ or ‘w’ short key rather than mouse double click activates switching view. To switch mode using this way, the tree view has to get the key event in. The tree view should get the focus before the key pressed.

Click code tab, wizard tab
Clicking the corresponding tab at the bottom of the StarEdit window switchs to the other view as shown in Figure 18-6.

18.3.6 Configuration
Click ‘config’ button on the top of StarEdit window.

What to Include
It is a filter option. If checked, that element is shown on the tree view. ‘Version’, ‘Author’, ‘Copyright’, ‘Location’, and ‘Description’ are not mandatory parts of the star sections. So you can hide them by unchecking.
Default attribute values
For relief of laborious typing all of them whenever making a new star file, the default values can be inserted in the “new star” wizard input form.

Replacement special strings
- “filename”: replace it with the current file name; ex) CGCRamp.pl
- “date”: replace it with the current date. ex) 15/11/2004
- “star”: the block(star) name ex) Ramp

They are replaced automatically when making a new star file with “make new star” wizard.

![Figure 18-10 Configure options](image)

**18.4 Star Sections Supported in StarEdit**

Refer to Chapter 5.2 of PeaCE User Manual for block definition file convention.

Table 18-1 shows the StarEdit directives supported in the star sections according to the following three sides: Show on tree, Wizard support, and Highlighting.
<table>
<thead>
<tr>
<th>Directives</th>
<th>Show on tree</th>
<th>Wizard support</th>
<th>Highlighting</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>domain</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>derivedFrom</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>desc or descriptor</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>version</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>author</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>copyright</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>location</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>htmldoc</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>explanation</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>hinclude, ccinclude</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>input, output</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>inmulti, outmulti</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>state or defstate</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>private, protected, public</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>code</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>constructor</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
</tr>
<tr>
<td>setup</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
</tr>
<tr>
<td>begin</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>initCode</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
</tr>
<tr>
<td>go</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
</tr>
<tr>
<td>wrapup</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
</tr>
<tr>
<td>codeblock</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>method</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 18-1 Star sections in a Star file

‘Δ’ means that this section code is generated by other section’s wizard. For example, setup {} code is generated by changing the input port wizard with rate or the message structure information.
18.5 Working With Star Files

18.5.1 Creating a new Star File

*Popup Menu > make new star* creates a new star wizard. See Figure 18-11 and Figure 18-12.

![Figure 18-11 Make a new star](image1)

![Figure 18-12 Star wizard](image2)
After filling the form in the star wizard, press ‘OK’. Then the wizard generates the star code and you can see the code in the code view. The wizard view is changed to the code view by double clicking on the background. See Figure 18-13.

![Figure 18-13 Star wizard generates a star code](image)

Since we did not save the new file with a proper name, an error (“filename is not correct!”) appears on the StarEdit tree window. Press CTRL+s, save it as the following star file name convention. See Figure 18-14.

**Star filename convention**

“Domain name “ + “Star name” + “.” + “pl”

ex) Domain = CGC, Star = “Ramp” → CGCRamp.pl

If you fail to name the star file according to the convention, you might see the error marks in the StarEdit tree view.
Now, you can add input, output, state, and codeblock from the popup menu, or copies from other star files and edit it.

### 18.5.2 Editing a Star File

Once you opened a file, jEdit stores it to an internal buffer. Figure 18-15 shows the buffer switcher. Even though the entry number of buffer switcher is limited, if you want to edit the already opened file, you don’t have to enable “File System Browser”. Just click the buffer switcher, and select the star file you want to edit. StarEdit tree parses it automatically and shows the result on tree view.
You can add a star section like port, state by the associated wizard or code manually in the code view. Or it is possible to copy some blocks from other files and modify it in the code view. See Chapter 18.6 for more information on the latter topic.

Figure 18-16 shows the popup menu invoked in the background on the tree view.

- make new star
- add input
- add output
- add state
- add codeblock

It is the simplest way to construct a star file structure. More details are on Chapter 18.6.

Figure 18-17 shows a way of deleting a particular state. If you want to delete a star element, just click mouse right button on that element in the tree view. Then ‘delete …’ popup menu will appear. If not, you should delete it manually in the code view.
Figure 18-16 Popup menus

Figure 18-17 Delete state
18.5.3 Editing Remote Files

PeaCE package contains many library block files (.pl files). And in most cases, pl files likely exist in the server machine where PeaCE server is running. But users may have their own desktop computer where PeaCE client program called ‘Hae’ and our jEdit powered by ‘StarEdit’ are running. So we need to create or edit a star file on a remote machine. This can be done easily by installing ‘FTP’ jEdit plugin. jEdit has a very flexible plugin architecture. By simply installing various plugins, features can be easily added. That is the main reason why we choose jEdit for platform of StarEdit.

Now let’s install the ‘FTP’ plugin. Figure 18-18 shows a Plugin Manager in jEdit. It is invoked by Plugins > Plugin Manager. Install tab shows the current available plugins via the internet. Check the ‘FTP’ in network category. Then press the ‘Install’ button.

![Figure 18-18 Plugin manager](image)

At the same time downloading finishes, the installation is complete. Select “System File Browser” and press the drop down list like Figure 18-19. Connect to secure ftp server. Even though two options are available to connect the ftp server, Secure ftp is recommended for security reason.
Figure 18-19 Download and Config FTP

Figure 18-20 Connect to the secure ftp server
If you give secure FTP server the right authentication information, something like Figure 18-21 appears. It shows the remote file system as your local file system. Now you can edit a remote star file like a local file.

Figure 18-21 After connecting a remote machine

18.6 Making Star Code With Wizard

As documented in Chapter 18.5, a user can create a new star, or edit an existing star file. Since editing in the code view is the same as other popular text editors, now focus on the usage of wizard. The reason why we should edit a star file by wizard is that some sections are closely related with the other sections. For example, ‘input’ is associated with the ‘setup’ section. Without the wizard help, a user has to remember the rules and code whenever he (or she) creates and modifies that section related to other sections. Thanks to wizard, we can avoid this annoying and tricky job. In addition to the main body of the wizard target section, the wizard can automatically generate some extra codes in other sections.

The main important sections are as follows.
18.6.1 Star Attributes

See Figure 18-12. There are some attributes visible in the star wizard.

**Name**: Name of the block (required)

**Domain**: CGC (c code generation) support (required)

**Author**: Author of the block

**Version**: Version of the block definition

**Copyright**: Copyright of the block

**Location**: Location of this block in the PeaCE server

**Description**: Description of the block behavior

**noInternalState**: Check box whether this star has any internal state. If you check this option, the following code is generated in ‘constructor{}’.

```c
constructor {
    noInternalState();
}
```

After you fill the proper values in the wizard form, press “OK”. Then the following star definition code is generated in new star mode, or is changed in edit star mode.

```c
defstar {
    name { Ramp }
    domain { CGC }
    desc {
        Generates a ramp signal, starting at "value" (default 0) with step size "step" (default 1).
    }
    version {@(#)CGCRamp.pl 1.9 01/01/96}
    author { E. A. Lee }
    copyright {
        Copyright (c) 1990-1996 The Regents of the University of California. All rights reserved.
        See the file $PTOLEMY/copyright for copyright notice,
    }
}
```
18.6.2 Input, Output Port

Refer to Chapter 5.2 for more information about port. Input, output, inmulti, and outmulti sections are generated by this wizard. See Figure 18-22. This figure is a new input port wizard. If you go into the wizard view in edit mode of existing input ports, the values of each field appears on the form in the wizard view.

The attribute fields of port include the followings. These attributes are the same to output port.

Name: Input or output port name (required).

Type: int, float, or message, any other customized type (required)
**Description**: Describe this port.

**Message** If you type ‘message’ in the type field, you should provide the specific c-codes related to this message type like this.

```c
struct IntBlock {
    int data[64];
}
```

Wizard makes a new codeblock named ‘def_’ followed by {structure name}; for the example, ‘def_IntBlock’.

```c
codeblock(def_IntBlock) {
    struct IntBlock {
        int data[64];
    }
}
```

You need to change the type of this codeblock into ‘GLOBAL’. Refer to Chapter 18.6.4.

Then, add the two lines in the setup section: ‘input’ is the port name.

```c
input.setMessageName("struct IntBlock");
input.setMessageSize(sizeof(struct IntBlock));
```

**Rate**: It represents the number of samples produced or consumed at this port per node firing. This attribute is related to the ‘setup {}’ section. There may be two cases.

- **Case 1)** If you type the number like ‘10’
  The following code is generated in the setup section; ‘output’ is the port name.
  ```c
  output.setSDFParams(10,9);
  ```

- **Case 2)** If you use the variable name like ‘count’
  The following code is generated in the setup section; ‘output’ is the port name.
  ```c
  output.setSDFParams(int(count),int(count)-1);
  ```

Next, make a new state named ‘count’. The following code will be generated.

```c
defstate {
```
name {count} 
  type {int} 
  default {1} 
  desc {} 
}

Check the generated code in the code view, and modify it manually. Or invoke the state wizard by double click in that state region in code view.

**Multi-Input / Output** : If you check this, the wizard changes the section name ‘input’ to ‘inmulti’, ‘output’ to ‘outmulti’.

The fully generated code is as follows if you make a message typed input port having the sampling rate ‘count’ and named ‘input’, and supply the struct c-code like above example.

```
input { 
  name {input } 
  type {message} 
  desc {} 
}

setup { 
  input.setMessageName("struct IntBlock");
  input.setMessageSize(sizeof(struct IntBlock));
  input.setSDFParams(int(count), int(count)-1);
}

defstate { 
  name {count} 
  type {int} 
  default {1} 
  desc {}
}

codeblock(def_IntBlock) { 
  struct IntBlock { 
    int data[64];
  }
}
```

In the editing mode of the port wizard view, you can type the new message and rate value even though the current values do not appear. The newer values replace old ones.
If you delete this port later by popup ‘delete’ menu, Wizard will try to delete the main body of this port and other extra code with associated with this port. The generated code in the setup section is deleted automatically. But the codeblock and the state code associated with this port is not deleted, because other port might use this port.

18.6.3 State

Refer to Chapter 5.2 for more information about state.

This section is used to define a state or a parameter of a block. The following are the attributes of a state. See Figure 18-23.

**Name**: State name, (required)

**Type**: int, float, or other

**Default**: Default value

**Attribute**: Attributes of this state, refer to Chapter 5.2.8.

**Description**: Description of the state

The generated code example is as follows.

```python
defstate {
    name { value }
    type { float }
    default { 0.0 }
    desc { Initial (or latest) value output by Ramp. }
    attributes { A_SETTABLE | A_NONCONSTANT }
}
```
18.6.4 Codeblock

You need to know about codeblock to use codeblock wizard fully in StarEdit. Codeblock defines the C-code segment to be emitted by code generation method. The most important attribute of codeblock in StarEdit is its type. Codeblock type means the place where this codeblock is emitted during code generation process. Refer to Chapter 5.2.8 for more information.

The codeblock wizard has the following four attributes.

**Name**: Codeblock name (required).

**Type**: There are six types of codeblock. Each type is related to a specific section and command such as ‘addCode()’ or ‘addGlobal()’. So in addition to the main body of this codeblock, the wizard generates the extra codes according to its type.

- **GLOBAL**
  
  Global codeblock is a global variables declaration section or type definition section in C. The following extra code is generated in the ‘initCode’ section; ‘defStruct’ is the codeblock name, and ‘struct IntBlock” is this codeblock’s tag name.

  ```c
  addGlobal(defStruct,"struct IntBlock");
  ```
• PROCEDURE
   Procedure codeblock is a collection of functions. The following code is generated in the ‘initCode’ section; ‘foos’ is the codeblock name, and ‘myFoos’ is the identifier of the codeblock.

   \[\text{addProcedure(foos, "myFoos");}\]

• CODE
   The following code is usually generated in the ‘initCode’ section; ‘code’ is the codeblock name, and ‘myCode’ is the identifier of the codeblock.

   \[\text{addCode(code, "myCode");}\]

• GO
   The following code is generated in the ‘go’ section; ‘main’ is the codeblock name.

   \[\text{addCode(main);}\]

• WRAPUP
   The following code is generated in the ‘wrapup’ section; ‘close’ is the codeblock name.

   \[\text{addCode(close);}\]

• TEMP
   No extra code is generated.

Tag : The tag name associated with the codeblock. Once you specify the tag name, it is used in addCode(), addGlobal() line insertion like ‘addCode(block0, “myblock0”)’.

Code : You can copy and paste the c-code segment from other files here. When “OK” is pressed, the c source lines are intended automatically.
Figure 18-25 shows the generated codeblock named ‘main’ and ‘addCode()’ line in go section. The type of ‘main’

codeblock’ is GO.

In the editing mode of a codeblock, you can change the type of the codeblock freely but may not change the name.
If you change the type of ‘main’ codeblock from ‘GO’ to ‘CODE’, the wizard deletes the ‘addCode(main)’ in go
section, and insert ‘addCode(main)’ line inside initcode section. When you delete the codeblock in the popup
menu, that codeblock and related extra lines are also deleted.
18.7 Import Star With Hae

For more detail information, refer to Chapter 3.5.

After you import the star you are making using StarEdit, you can have your own block. Also you can make a more complex schematic or design project with PeaCE and Hae based on this star as a building block.
Chapter 19. Utilities

Author: Kiseun Kwon, Seongnam Kwon, and Soonhoi Ha

This chapter explains some utilities in PeaCE. As of now, three utilities are explained here: XMLConverter, makeStarSrcPath, and dbUpdate

19.1 XMLConverter

19.1.1 Usage

XMLConverter translates a simple text file that represents a random graph into the interface files that are generated from the graph analysis step. They are Design.xml for graph topology, Design_TimeCost.xml for block-performance information, and Design_mode.xml for timing constraints of tasks. These files are used at the partitioning stage in the PeaCE codesign flow. This utility helps the designer run the partitioning algorithm with the randomly generated graph without drawing the SPDF graphs in Hae.

“XMLConverter” exists in the directory “$PEACE/bin.linux”, and if peace is installed, it will work correctly.

To use the utility, do the following:
1. make an input text file whose format is explained in the next sub-section.
2. Command as follows: ‘xmlconverter input_file_name Design_name’

Figure 19-1 shows the screen display after executing ‘XMLConverter’. It prints the parsing process. After executing ‘XMLConverter’, you can see aforementioned three interface files as can be found in Figure 19-1.
19.1.2 Syntax description of input file

The syntax of an input file is simple. This file can be easily created by a random graph generator or by hand. All you have to do is filling the corresponding fields leaving no empty field. For an invalid field, just write it down by 0. You must not skip any field in an input file format. That’s all. Figure 19-2 shows the input file format.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>processor_type processor_name processor_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.....</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>task_name</td>
</tr>
<tr>
<td></td>
<td>number of deadlines</td>
</tr>
<tr>
<td></td>
<td>task_deadline</td>
</tr>
<tr>
<td></td>
<td>number of nodes</td>
</tr>
<tr>
<td></td>
<td>node_name node_input_port_number output_port_number</td>
</tr>
<tr>
<td></td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td>number of nodes</td>
</tr>
<tr>
<td></td>
<td>node_name exec_time cost exec_time cost ....</td>
</tr>
<tr>
<td></td>
<td>(for number of processors)</td>
</tr>
<tr>
<td></td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td>number of arcs</td>
</tr>
<tr>
<td></td>
<td>arc_id from_node_name from_port_id to_node_name to_port_id ipc</td>
</tr>
<tr>
<td></td>
<td>.....</td>
</tr>
</tbody>
</table>
Figure 19-2 Input file format

Description of processor

- number of processors: the total number of processors
- processor_type: if the type of the processor is SW then '0', else if it's type is HW then '1'
- processor_name: the name of the processor, for example arm922T, FPGA and so on..
- processor_cost: the cost of the processor

Description of task

- number of tasks: the total number of tasks
  There should be as many definition of tasks as this number.
- task_name: the name of a task
- number of deadlines: the number of deadlines that the task can have.
  A task can have one or more deadlines according to which graph it belongs to
- deadline: deadline of the task

Description of node

- number of nodes: the total number of nodes in a task
- name: the name of a node
- number of input/output port: the number of input/output ports of the node
- exec_time: execution time of the node
- cost: cost of the node

Description of arc

- number of arcs: the total number of arcs in a task
- arc_id: identifier of an arc (0,1,2,...)
- from_node_name: name of the source node of the arc
- from_node_port: output port id of the source node of the arc
- to_node_name: name of the destination node of the arc
- to_node_port: input port id of the destination node of the arc
- ipc: communication cost between two nodes.
### 19.1.3 Example of an input file

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcX</td>
<td>100</td>
</tr>
<tr>
<td>ProcY</td>
<td>50</td>
</tr>
<tr>
<td>ProcZ</td>
<td>20</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Task</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>HouTask1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of deadlines</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline</td>
<td>350</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Node</th>
<th>Name</th>
<th>Number of input ports</th>
<th>Output Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Node</th>
<th>Name</th>
<th>Execution</th>
<th>Cost in ProcX</th>
<th>Cost in ProcY</th>
<th>Cost in ProcZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5 12</td>
<td>1 18 1 18</td>
<td>1 18 1</td>
<td>1 18</td>
<td>1 18</td>
</tr>
<tr>
<td>b</td>
<td>10 1</td>
<td>18 1 40</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>5 12</td>
<td>1 18 1 18</td>
<td>1 18</td>
<td>1 18</td>
<td>1 18</td>
</tr>
<tr>
<td>d</td>
<td>35 1</td>
<td>85 1 195</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>15 1</td>
<td>22 1 80</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>30 1</td>
<td>75 1 180</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>15 1</td>
<td>25 1 85</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>15 1</td>
<td>35 1 47</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>7 1 10</td>
<td>30 1</td>
<td>30 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>10 1</td>
<td>28 1 35</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Arc</th>
<th>ID</th>
<th>Source Node</th>
<th>Port ID of Source Node</th>
<th>Destination Node</th>
<th>Port ID of Destination Node</th>
<th>IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a 1 b 1</td>
<td>a</td>
<td>1</td>
<td>b</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>a 1 b 1</td>
<td>a</td>
<td>2</td>
<td>d</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>b 1 c 1</td>
<td>b</td>
<td>1</td>
<td>c</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>c 1 d 2</td>
<td>c</td>
<td>1</td>
<td>d</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>d 1 e 1</td>
<td>d</td>
<td>1</td>
<td>e</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>d 2 f 1</td>
<td>d</td>
<td>2</td>
<td>f</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>d 3 g 1</td>
<td>d</td>
<td>3</td>
<td>g</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>e 1 i 1</td>
<td>e</td>
<td>1</td>
<td>i</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>f 1 i 2</td>
<td>f</td>
<td>1</td>
<td>i</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>f 2 j 1</td>
<td>f</td>
<td>2</td>
<td>j</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>f 3 h 1</td>
<td>f</td>
<td>3</td>
<td>h</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
19.2 makeStarSrcPath: find the source path of all blocks

Hae is the graphical user interface (GUI) of PeaCE and users can easily manage their schematics and blocks using Hae GUI. In Hae, you can see the internal definition of a block by just double clicking the block. But if you add your new block to PeaCE, Hae is unable to locate the source file of the block. It means that you should the add path information of an added block to the database for seeing the definition. You can add new information manually, but we recommend using the “makeStarSrcPath” utility.
“makeStarSrcPath” exists in the directory “$PEACE/bin”, and if peace is installed, it will work correctly.

Usage is very simple, just execute “makeStarSrcPath” without any other arguments, and it will search block sources and update database automatically.

Note that this utility will check the directory $PEACE and its child directories.

19.3  **dbUpdate: update the CGC block performance database**

In PeaCE, performance information of blocks is stored in a database. While PeaCE offers a performance estimation method of software blocks, you may want to update the database manually. Then you need this utility, “dbUpdate.”

“dbUpdate” exists in the directory “$PEACE/bin.linux”, and if peace is installed, it will work correctly.

“dbUpdate” reads a text file and update it to the CGC performance database. You should make the text file of block performance. File format is described in chapter 12 of PeaCE manual.

Usage is very simple. If you execute “dbUpdate” without any arguments, this message will be shown.

**Usage: dbUpdate input_file_name Update_op(0:always, 1:conditional)**

First argument is the text file name, and second is the execution option. If option is 0, “dbUpdate” overwrites the block performance information that already exists in the database. In case of 1, if there is already the performance information of the same block, overwriting is skipped.

19.4  **peacepasswd: update a user's authentication tokens**

User authentication is one of the most important problems in security. For this reason, when a user logs-in PeaCE through Hae, peaceDaemon authenticates the user by checking the password. In this step, PeaCE reads the password file “$(HOME)/.peacepasswd” that keeps encrypted user password tokens. To make this file, PeaCE offers user password updater utility. You can update password by running “peacepasswd” file. Its usage is very simple – very similar to the Linux utility “passwd”. Just run “peacepasswd” then you can see following messages.

Changing password
(current) PeaCE password:
New PeaCE password:
Retype New PeaCE password:
If you already set your password, you should input the previous password to update it. If not, you should just type your new password twice.
## Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ref</td>
<td>65</td>
</tr>
<tr>
<td>$val</td>
<td>65</td>
</tr>
<tr>
<td>,pl</td>
<td>61</td>
</tr>
<tr>
<td>“bus” connection</td>
<td>110</td>
</tr>
<tr>
<td>“get()” method</td>
<td>168</td>
</tr>
<tr>
<td>“put(time_stamp)” method</td>
<td>168</td>
</tr>
<tr>
<td>A delay</td>
<td>42</td>
</tr>
<tr>
<td>addCompileOption</td>
<td>70</td>
</tr>
<tr>
<td>addDeclaration</td>
<td>67</td>
</tr>
<tr>
<td>addGlobal</td>
<td>69</td>
</tr>
<tr>
<td>addInclude</td>
<td>67</td>
</tr>
<tr>
<td>addLinkOption</td>
<td>70</td>
</tr>
<tr>
<td>addProcedure</td>
<td>66, 70</td>
</tr>
<tr>
<td>Arch.xml</td>
<td>269</td>
</tr>
<tr>
<td>architecture description file</td>
<td>269</td>
</tr>
<tr>
<td>Array states</td>
<td>38</td>
</tr>
<tr>
<td>ArrayState</td>
<td>73</td>
</tr>
<tr>
<td>arrivalTime</td>
<td>168</td>
</tr>
<tr>
<td>attributes</td>
<td>73</td>
</tr>
<tr>
<td>Backplane model</td>
<td>58</td>
</tr>
<tr>
<td>block</td>
<td>59</td>
</tr>
<tr>
<td>BP Domain</td>
<td>207</td>
</tr>
<tr>
<td>bufferSharing</td>
<td>87</td>
</tr>
<tr>
<td>bus manipulation</td>
<td>111</td>
</tr>
<tr>
<td>bus-line</td>
<td>110</td>
</tr>
<tr>
<td>calendar queue</td>
<td>173</td>
</tr>
<tr>
<td>causality error</td>
<td>57</td>
</tr>
<tr>
<td>ccinclude</td>
<td>71</td>
</tr>
<tr>
<td>Clustered.xml</td>
<td>245</td>
</tr>
<tr>
<td>codeblock</td>
<td>65, 74</td>
</tr>
<tr>
<td>combo-box for a state</td>
<td>65, 74</td>
</tr>
<tr>
<td>comment</td>
<td>40</td>
</tr>
<tr>
<td>comments in a state</td>
<td>40</td>
</tr>
<tr>
<td>completionTime</td>
<td>168</td>
</tr>
<tr>
<td>complex</td>
<td>82</td>
</tr>
<tr>
<td>Complex-valued states</td>
<td>38</td>
</tr>
<tr>
<td>conName</td>
<td>105</td>
</tr>
<tr>
<td>constructor</td>
<td>74</td>
</tr>
<tr>
<td>conValue</td>
<td>105</td>
</tr>
<tr>
<td>cosimulation</td>
<td>241</td>
</tr>
<tr>
<td>coverification</td>
<td>241</td>
</tr>
<tr>
<td>DE model</td>
<td>166</td>
</tr>
<tr>
<td>defstate</td>
<td>72</td>
</tr>
<tr>
<td>delay-free</td>
<td>170</td>
</tr>
<tr>
<td>delay-free loop</td>
<td>175</td>
</tr>
<tr>
<td>delays</td>
<td>41</td>
</tr>
<tr>
<td>delay-type</td>
<td>167, 170</td>
</tr>
<tr>
<td>derivedFrom</td>
<td>71</td>
</tr>
<tr>
<td>design browser</td>
<td>30</td>
</tr>
<tr>
<td>design tabs</td>
<td>226</td>
</tr>
<tr>
<td>Discrete Event (DE) model</td>
<td>57</td>
</tr>
<tr>
<td>domain</td>
<td>56</td>
</tr>
<tr>
<td>Domain</td>
<td>60</td>
</tr>
<tr>
<td>Dynamic constructs</td>
<td>104</td>
</tr>
<tr>
<td>dynamic linking</td>
<td>68</td>
</tr>
<tr>
<td>dynamic sample rate</td>
<td>107</td>
</tr>
<tr>
<td>event generator</td>
<td>167</td>
</tr>
<tr>
<td>event generators</td>
<td>172</td>
</tr>
<tr>
<td>event-driven</td>
<td>58, 166</td>
</tr>
<tr>
<td>Export</td>
<td>29</td>
</tr>
<tr>
<td>fFSM model</td>
<td>185</td>
</tr>
<tr>
<td>fFSM Model</td>
<td>180</td>
</tr>
<tr>
<td>fixed point</td>
<td>81</td>
</tr>
<tr>
<td>fixed-point representation</td>
<td>129</td>
</tr>
<tr>
<td>fixed-point simulation</td>
<td>112</td>
</tr>
<tr>
<td>Fractional Rate</td>
<td>92</td>
</tr>
<tr>
<td>fractional rate dataflow</td>
<td>57</td>
</tr>
<tr>
<td>FRDF</td>
<td>57</td>
</tr>
<tr>
<td>FSM</td>
<td>180</td>
</tr>
<tr>
<td>fullName</td>
<td>71</td>
</tr>
<tr>
<td>functional block</td>
<td>167</td>
</tr>
<tr>
<td>functional blocks</td>
<td>170</td>
</tr>
<tr>
<td>Functional task</td>
<td>210</td>
</tr>
<tr>
<td>galaxy</td>
<td>59</td>
</tr>
</tbody>
</table>
global event queue ......................................................... 173
go ........................................................................... 74
graph analysis ......................................................... 233
graph topology description file ........................................ 245
High-Order-Function .................................................. 110
hinclude ...................................................................... 71
HOF block ................................................................. 110
Image ........................................................................... 99
import ........................................................................... 65
import a block ............................................................. 43
Import Star ................................................................. 43
initCode ........................................................................... 66, 74
in-line codeblock ........................................................ 76
input ........................................................................... 72
Internal event .............................................................. 188
iteration ........................................................................... 77
library browser ......................................................... 24, 35
Link Star ................................................................. 45
linking ........................................................................... 68
main declarations ........................................................ 70
mainClose ................................................................. 70
mainInit ................................................................. 70
mainLoop ................................................................. 70
Matrix ........................................................................... 99
message ...................................................................... 84
mode description file .................................................... 258
mode.xml ...................................................................... 258
ModelSim ................................................................. 124
multi-port ...................................................................... 67
nodeName ...................................................................... 195
noInternalState .......................................................... 64
Octave .......................................................................... 177
output ............................................................................ 72
Particle .......................................................................... 170
Performance estimation ................................................. 230
periodic task .............................................................. 210
phase ............................................................................ 95
Piggyback ................................................................. 102
processorId ............................................................... 229
ptlang ............................................................................ 61
ref .............................................................................. 67, 70
Register Star .............................................................. 44
repetition ratios .......................................................... 78
resmap ......................................................................... 131
resource file ............................................................... 130
schedule ........................................................................... 77
Scheduler ........................................................................... 77
script language ............................................................ 195
Seamless CVE ........................................................... 242
setAccessSequence ..................................................... 95
setAtomicType ............................................................ 95
setsourcedenum .......................................................... 131
setSDFParams ............................................................ 64
simultaneous events ..................................................... 171
SPDF .......................................................................... 57
Sporadic task .............................................................. 213
star .............................................................................. 59
starName ...................................................................... 71
starSymbol .................................................................... 71
start-up dialog ............................................................ 23
state .............................................................................. 72
state-update ............................................................... 102
staticBuffering ........................................................... 81
stop time ...................................................................... 173
String states ............................................................... 38
StringList ...................................................................... 67
super block ............................................................... 59
super-block .............................................................. 45
super-block states ....................................................... 47
Synchronous Piggybacked Data Flow .......................... 104
synchronous piggybacked dataflow ............................... 57
target ................................................................. .......................... 56
Target ................................................................. 219
task group ................................................................. 60
Task Model .............................................................. 58, 207
task types ................................................................. 207
task wrapper ............................................................. 208
Tel interpreter ............................................................. 39
tcl/tk ........................................................................... 109