System-On-Chip Component Models

Andreas Gerstlauer
Lukai Cai
Dongwan Shin
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-03-26
August 11, 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

http://www.cecs.uci.edu

System-On-Chip Component Models

Andreas Gerstlauer
Lukai Cai
Dongwan Shin
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-03-26
August 11, 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

{gerstl,lcai,dongwans,doemer,gajski}@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

This report defines and describes a format for models of system components required for system-on-chip (SoC) design. In an SoC design process, starting from an initial system specification, an implementation of the system is created through a series of interactive and automated steps by gradually synthesizing and assembling a system design using components taken out of a set of databases. Generally, databases are needed for processing elements (PEs), bus and other communication protocols, and RTL units. In this report we aim to provide an exhaustive list of requirements for components in an automated SoC design flow using the example of a concrete database format. Following a description of the basic database format in general, this report defines the format of each of the three databases in detail. Using information in this report, specific database formats for diverse SoC design flows can be developed. Specifically, the database format in this report is used successfully in our SoC Design Environment, SCE.

Contents

List of Figures

List of Tables

List of Listings

System-On-Chip Component Models

A. Gerstlauer, L. Cai, D. Shin, R. Dömer, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
{gerstl,lcai,dongwans,doemer,gajski}@cecs.uci.edu
http://www.cecs.uci.edu

Abstract

This report defines and describes a format for models of system components required for system-on-chip (SoC) design. In an SoC design process, starting from an initial system specification, an implementation of the system is created through a series of interactive and automated steps by gradually synthesizing and assembling a system design using components taken out of a set of databases. Generally, databases are needed for processing elements (PEs), bus and other communication protocols, and RTL units. In this report we aim to provide an exhaustive list of requirements for components in an automated SoC design flow using the example of a concrete database format. Following a description of the basic database format in general, this report defines the format of each of the three databases in detail. Using information in this report, specific database formats for diverse SoC design flows can be developed. Specifically, the database format in this report is used successfully in our SoC Design Environment, SCE.

1 Introduction

In state-of-the-art design flows for system-on-chip (SoC) design, an initial specification of a system is taken down to an actual implementation through a series of interactive and automated steps [1]. In such design flows, the system is gradually synthesized and assembled using system components taken out of a set of databases. Specifically, an SoC design flow needs to include databases for processing elements (PEs), bus or other communication protocols, and RTL units.

The *processing element (PE) database* contains programmable processors, synthesizable custom hardware components, system memories, IPs, etc. at different levels of abstraction. Such PEs are used during the design flow to implement the computation of the system by mapping computational parts onto PEs taken out of the PE database.

The *bus protocol database* contains timing-accurate descriptions of protocols of system busses and other communication structures used to implement communication between the different PEs in the system.

Finally, the *RTL unit database* contains register-transfer units like register files, ALUs and other functional units, memories, local busses, etc. RTL units taken out of the RTL database are used to synthesize computation mapped to custom hardware PEs down to a cycle-accurate RTL description.

This report describes and defines a format of databases for use in automated SoC design flows, thereby outlining requirements for modeling components for system design in general. First, Section 2 defines the general format of the SoC databases. Then, in Section 3, Section 4, and Section 5 the specifics of the PE, bus, and RTL databases are described in detail, respectively.

2 General Database Format

Our databases for SoC design are described in the form of SpecC code, i.e. the format of the databases is based on the SpecC syntax [2] and the SpecC source code for each database must be compilable into SIR (SpecC Internal Representation) files using the SpecC compiler (‘`scc`’, Section A.1).

2.1 Database Organization

For each database, there is exactly one top-level SIR file. The top-level database SIR acts as a container for all components stored in the database. The database SIR includes components through import of individual component SIR files where component SIR files have to be stored in the same directory as or a sub-directory of the directory the database SIR is located in.

Each component in the database must be stored in a separate SIR file. Component SIRs will be imported by SCE as needed throughout the design flow. Therefore, component SIRs must be self-contained. Furthermore, in case of parameterizable components, the component source code must be made available as part of the database, too.

In case of databases with multiple models of each component at different levels of abstraction, each model can be stored in a separate SIR file as long as the top-level database SIR contains all component models through direct or indirect import. In those cases, the basic component model used during allocation will contain pointers to other models (in the form of annotations) and SCE will import those models when needed in the design flow. The advantage of separate component files is that at any stage of the design flow, the system design will not contain any yet unused component models.

Since different models of components out of the different databases get imported into the same system design throughout the design flow, SIR design (import) names and identifiers defined in the SIR files must be unique across all databases and components. Furthermore, identifiers should be chosen appropriately to reduce conflicts with names defined in the system design itself. It is highly recommended to use unique prefixes for component names such that naming conflicts are avoided.

Listing 1 shows a simple example for database organization. The database `database` (Listing 1(c)) contains two components, *Component0* and *Component1*, that are stored in separate files `component0` (Listing 1(a)) and `component1` (Listing 1(b)), respectively.

2.2 Component Format

Components are described by SpecC objects (behaviors or channels) in the SoC databases. Depending on the database, part or all of the functionality of a component is described through the SpecC code of the component object. As needed, component objects can be hierarchically composed out of other SpecC objects stored together with the top-level component object in the database. Note that unless noted, SpecC code describing component functionality in the databases is for simulation purposes only and does not have to be synthesizable, i.e. apart from the restrictions of the database format defined in this document, any valid SpecC code can be used.

On top of SpecC code to describe functionality, additional meta-information about each component is stored in the database in the form of SpecC annotations attached to the component object. Apart from general annotations for database management, components generally have attributes, parameters, and profiling weight tables.

```
attribute =
        value
    | attribute_range

attribute_range =
        '{' default_value ',' range_list '}'

range_list =
        range_value
    | range_list ',' range_value

range_value =
    value
    | '{' min ',' max '}'
```

Figure 1: Format of component attribute annotations.

2.2.1 Attributes

Component attributes describe characteristics or metrics for a component. Attributes of a component are stored as annotations attached to the component object under different keys or names as defined by the database.

The general format of the value of an attribute annotation is defined in Figure 1. An attribute is either a simple annotation giving a fixed attribute value or a complex annotation describing a range of possible values for an adjustable attribute. For an adjustable attribute, the system designer will be allowed to tune the attribute value during allocation in the SoC design process within the range defined by the annotation.

The range of values for an adjustable attribute is given as a list of possible values. Each entry in the list is either a single, fixed value or a pair of values defining the lower and upper boundaries of an interval of values (for numeric types only). The first entry in the list must be a single value describing a sensible default. In all cases, all values must be of the same type as defined by the attribute.

Listing 2 shows an example of component attribute annotations. The annotations define a fixed value of 2.3 for `_PE_COST` and a range of values from 0.0 up to 60000000.0 with a default of 60000000.0 for `_PE_CLOCK_FREQUENCY`.

2.2.2 Parameters

All components in the SoC databases can be parameterizable. For a parameterized component, the system designer selects values for each of the component's parameters during allocation. The SoC design tools will then supply the parameter values to the SpecC Design Generator (‘`sir_gen`’, Section A.2) to generate application-specific implementations of the component for use in the design as needed. In case of components with multiple models, only parameters defined for the component model used during

```
behavior Component0();
// ...
//
```

(a) `component0.sc`

```
behavior Component1();
// ...
//
```

(b) `component1.sc`

```
// Component database

import "component0";
import "component1";
```

(c) `database.sc`

Listing 1: Database organization example.

```
behavior DSP();

note DSP._PE_COST = 2.3e0;

note DSP._PE_CLOCK_FREQUENCY = { 0.6e+08, { 0.0, 6e7 } };
```

Listing 2: Component attributes example.

```
_SIR_PARAMETERS =
        '{' parameter_list '}'

parameter_list =
        parameter
        | parameter_list ',' parameter

parameter =
        '{' name ',' deflt ',' prm_range ',' description ',' unit '}'

prm_range =
        '{' range_list '}'

range_list =
        range_value
        | range_list ',' range_value

range_value =
        value
        | '{' min ',' max '}'
```

Figure 2: Format of `_SIR_PARAMETERS` annotation.

	int	float
+	1	2
*	3	4

Table 1: Example of a profiling table.

allocation are relevant and parameter values supplied to this model are reused to generate all subsequent models.

Components are made parameterizable through the `_SIR_PARAMETERS` annotation used by '`sir_gen`'. Figure 2 describes the extended format of the `_SIR_PARAMETERS` annotation for use in component parameterization. The annotation defines a list of parameters where each parameter is a tuple consisting of the parameter name followed by a default value, a range of possible values, a string describing the parameter, and a string for the parameter unit. The range of parameter values is described as a list of range value in the same format as for attributes, see Section 2.2.1. The type of the default value defines the type of the parameter and all values in the range have to be of the same type.

Instances of parameterizable components are generated from the component source code by substituting the value of a parameter for all occurrences of the parameter name in the component macro described through the SpecC code of the component. For more information, please refer to the documentation of '`sir_gen`'.

For the purpose of inclusion in the database, components should be written in such a way that they compile with default parameter values using the standard SpecC compiler ('`scc`', Section A.1) instead of '`sir_gen`'

An example of a parameterized component is shown in Listing 3. The component has one parameter *BITWIDTH* of unsigned integer type that can vary between 2 and 64. The parameter description is "Width" and parameter values are given in "bits". The component source code defines a default value *BITWIDTH_DFLT* of 32 for the parameter and uses this value if no value is supplied for *BITWIDTH* through '`sir_gen`', e.g. for inclusion in the database.

2.2.3 Weight Tables

Component weight tables describe the special characteristics or metrics of a component which are used to evaluate the design metrics.

The design metrics are determined by the metrics of the system architecture represented by component weight tables and the metrics of the system behavior represented by profiling tables. For example, Table 1 represents the profiling table of a system behavior. The top/bottom row represents an addition("+")/multiplication("*") operation, and the left/right column represents integer("int")/float("float")

	int	float
+	1	2
*	4	8

Table 2: Example of the operation weight table of a DSP.

	int	float
+	1	4
*	12	32

Table 3: Example of a weighted result.

data type. Then, the "3" indicates that the system behavior executes the multiplication operation of integer type three times during simulation.

Assume we map the above system behavior to a DSP component. The weight table of the DSP representing the required clock cycles for each operation is listed in Table 2. The "4" describes that the DSP takes 4 clock cycles to complete the execution of one multiplication operation of integer type.

Accordingly, the weighted result computed by multiplying the weights in the weight table with the entries in the profiling table is described in Table 3. The summation of the weighted result, which equals to 49, indicates the total computation time of the design is 49 clock cycles.

Design metrics can be classified into three categories: operation metric set, traffic metric set, and memory metric set. In the component weight table, each row represents the weights for one item type and each column represents the weights for one data type. The possible item types and data types of different metric sets are listed in Section 2.2.4 and Section 2.2.5, respectively. Table 2, Table 4, and Table 5 are the examples of operation weight table, traffic weight table, and memory weight table. In general, operation weight tables and traffic weight tables are manually generated by designers. Memory weight tables are automatically generated by SCE based on the memory attributes.

```
item_header =
    '"'item_list'"'

item_list =
    item_type
    | item_list ',' item_type
```

Figure 3: Format of item header annotations.

Weight tables are stored in the data base by use of three annotations: item header annotation, data header annotation, and weight annotation. Each metric set contains an item header annotation and a data header annotation, which

```
#define BITWIDTH_DFLT 32u

#ifndef BITWIDTH
#define BITWIDTH BITWIDTH_DFLT
#endif

behavior Add(out unsigned bit[BITWIDTH-1:0] sum,
             in  unsigned bit[BITWIDTH-1:0] a,
             in  unsigned bit[BITWIDTH-1:0] b)
{
  note _RT_COST = BITWIDTH * 3.0;

  note _SIR_PARAMETERS = {
    { "BITWIDTH", BITWIDTH_DFLT, { {2u, 64u} }, "Width", "bits" }
  };

  void main(void) {
    sum = a + b;
  }
};
```

Listing 3: Parameterized component example.

```
behavior DSP();

note DSP._PE_WEIGHT_OPERATION_HEADER_ITEMTYPE = "+,*";

note DSP._PE_WEIGHT_OPERATION_HEADER_DATATYPE = "int,float";

note DSP._PE_WEIGHT_OPERATION_DYNAMIC = "1,2,4,8";
```

Listing 4: Component weight table example.

	int	float
in	1	2
out	3	4

Table 4: Example of a traffic weight table.

	int	float
local	1	4
global	12	32

Table 5: Example of a memory weight table.

lists the supported item and data types, respectively. The format of the values of the item header annotation and the data header annotation are defined in Figure 3 and Figure 4. The available item types and data types are listed in Section 2.2.4 and Section 2.2.5.

Each metric contains one weight annotation. The items in the weight table are stored as one dimensional array represented by a string, as defined in Figure 5. Each *value* must be of integer type.

Listing 4 shows an example of a component weight table annotation, which matches the example described earlier.

2.2.4 Item Types for Metric Set

The possible values for item types of operation metric sets are listed in Table 6 and Table 7. An operation metric set defines values for basic operations (Table 6) and basic statements (Table 7) corresponding to weights for the data flow and control flow complexity of the code.

For the operation metric set, besides the items listed in Table 6, each global function can be treated as an independent operation type. For example, if there is a global function $float\ F1(int\ arg1, long\ arg2)$, we can put ”F1” into the item header annotation of operation. Consequently, the weighted result of function ”F1” is computed by multiplying the weight of the entry in the row representing the item type ”F1” and in the column representing the data type ”float”(returning type of function), with the execution number of function ”F1”. In this case, the operations inside ”F1” will not be calculated again.

The possible values of item types of the traffic metric sets are listed in Table 8. The possible values of item types of the memory metric sets are listed in Table 9.

2.2.5 Data Types for Metric Set

Operation and traffic metric sets share the same set of possible values of data types, which are listed in Table 10.

Besides the data types listed in Table 10, data type annotation can also contain three groups of hierarchial data

Name	Description
void	void operation
#1	const data access operation
#i	identifer access operation
()	parenthesis operation
this	”this” pointer access operation
[]	array access operation
f()	function call operation
.	member access operation
->	member pointer access operation
p++	post-increment operation
p--	post-decrement operation
[:]	bit slice operation
++p	pre-increment operation
--p	pre-decrement operation
&p	address deriving operation
*p	content deriving operation
+x	unary plus operation
-x	negation operation
~	not operation
!	logic not operation
sizeof(E)	size of operation (on expression)
sizeof(T)	size of operation (on type)
()x	type conversion operation
@	concatenation operation
*	multiplication operation
/	division operation
%	modulo operation
+	addition operation
-	substraction operation
<<	left shift operation
>>	right shift operation
<	less operation
>	greater operation
<=	less equal operation
>=	greater equal operation
==	equal operation operation
!=	not equal operation
&	and operation
^	eor operation
|	or operation
&&	logic and operation
||	logic or operation
:?	condition operation
=	assignment operation

Table 6: Item types for operation metric set (operations).

```
data_header =
    '"'data_list'"'

data_list =
    data_type
    | data_list ',' data_type
```

Figure 4: Format of data header annotations.

```
weight =
    '"'  weight_list  '"'

weight_list =
    value
    | weight_list ',' value
```

Figure 5: Format of weight annotations.

Name	Description
`*=`	multiplication & assignment operation
`/=`	division & assignment operation
`%=`	modulo & assignment operation
`+=`	addition & assignment operation
`-=`	substraction & assignment operation
`<<=`	left shift & assignment operation
`>>=`	right shift & assignment operation
`&=`	and & assignment operation
`^=`	eor & assignment operation
`|=`	or & assignment operation
`,`	comma operation

Table 6: Item types for operation metric set (operations) (continued).

Name	Description
`:`	labeled statement
`{}`	compound statement
`*+`	expression statement
`if`	if statement
`else`	else statement
`switch`	switch statement
`case`	case statement
`default`	default statement
`while`	while statement
`do`	do statement
`for`	for statement
`goto`	goto statement
`continue`	continue statement
`break`	break statement
`return`	return statement
`par`	par statement
`pipe`	pipe statement
`exception`	exception statement
`timing`	timing statement
`fsm`	fsm statement
`fsmd`	fsmd statement
`wait`	wait statement
`waitand`	waitand statement
`waitfor`	waitfor statement
`notify`	notify statement
`notifyone`	notifyone statement

Table 7: Item types for operation metric set (statements).

Name	Description
`in`	input traffic
`out`	output traffic

Table 8: Item types for traffic metric set

Name	Description
`local`	local variable storage (stack memory)
`global`	global/static variable storage (static and heap memory)

Table 9: Item types for memory metric set

Name	Description
`bool`	bool
`char`	char
`unsigned char`	unsigned char
`short int`	short int
`unsigned short int`	unsigned short int
`int`	int
`unsigned int`	unsigned int
`long int`	long int
`unsigned long int`	unsigned long int
`long long int`	long long int
`unsigned long long int`	unsigned long long int
`float`	float
`double`	double
`long double`	long double
`void`	void(for return value of function)
`event`	event
`void*`	pointer

Table 10: Data types for metric sets.

types.

`Bit vector` We can define bit vectors with different lengths as different data types. The corresponding data type format is ”bit[size]” or ”unsigned bit[size]”. For example, data belonging to either *bit[8:0]* or *bit[9:1]* can be defined as an independent data type `"bit[9]"` in the data header annotation.

`Array` We can define arrays with different lengths as different data types. The corresponding data type format is *basic_type*[size]. For example, data belonging to int[23] and int[24] type can be defined as independent data types `"int[23]"` and `"int[24]"` respectively.

`User Defined Type` Every user defined structure/union/enum type can be treated as one independent data type. It uses the structure/uniton/enum names as the data type names. For example, if a structure is defined as ”struct s1{int a; float b}” and we want to define it as an independent data type, the defined data type ”struct s1” can be put into the data header annotation.

If above three groups of hierarchical data types are not defined, then we use their basic types to compute the weighted result. For example, if no data type ”int[24]” is defined, the data with type int[24] will be treated as 24 integer types.

	int	float	long
+	1	2	0
*	4	8	0
-	0	0	0

Table 11: An alternative operation weight table of the DSP.

2.2.6 Default Values

When specifying weight tables, designers don’t need to put all the item types in Table 6, Table 7, Table 8, or Table 9 to the item header annotation, and don’t need to put all the data types in Table 10 to the data header annotation. The default value of weights for the item types that are not in the item header annotation is 0. Similarly, the default value of weights for data types that are not in the data header annotation is 0. As a result, there is no different between Table 2 and Table 11 in terms of values of weights.

3 Processing Elements

Processing elements are represented by SpecC behaviors in the PE database. For each PE model, a corresponding behavior declaration needs to exist in the database.

The PE database can contain different PE models at varying levels of abstraction corresponding to the steps in the design flow. At minimum, behavioral PE models used for PE allocation and for initial implementation of specification behaviors mapped onto PEs are required to exist for each PE (Section 3.1). In addition, PEs can optionally have bus-functional models for use after communication synthesis (Section 3.2) and implementation models for cycle-accurate simulation in case of PEs with part or all of their functionality fixed (Section 3.3). In case of synthesizable or programmable PEs, PE models provide templates that will be filled automatically during the design process.

3.1 PE Behaviors

Behavioral PE models are the basic models of PEs inside SCE and the PE behaviors represent the PEs available in the database. Therefore, for each PE there must be at least a behavioral model and the set of PE behaviors defines the list of PEs in the PE database. PE behaviors are used for PE allocation and they define the basic characteristics like attributes and parameters for a PE and all of its lower-level models. Furthermore, the SpecC code of behavioral PE models can describe high-level functional aspects of the PE in case all or part of the PE’s functionality is pre-defined and fixed.

3.1.1 General Format

```
pe_behavior =
    behavior pe_name ports_opt interfaces_opt
        body_opt ';'

ports_opt =
    <nothing>
    | '(' ')'
    | '(' channel_port_list ')'

channel_port_list =
    channel_port
    | channel_port_list ',' channel_port

channel_port =
    interface_name
    | interface_name port_name

interfaces_opt =
    <nothing>
    | implements interface_list

body_opt =
    <nothing>
    | '{' internal_declarations_opt '}'
```

Figure 6: Format of PE behaviors.

A PE behavior is created through a special SpecC behavior declaration or definition. The format of PE behaviors is shown in Figure 6. The name of the SpecC behavior defines the name of the PE component. For PEs with fixed computation functionality, the PE behavior has ports of abstract, message-passing interface type and an internal or external body (Section 3.1.5).

PE behaviors carry a number of annotations for general database management, attributes (Section 3.1.2), and optional parameters (see Section 2.2.2).

Table 12 lists the general annotations that are attached to PE behaviors for database management and in order to describe basic information about the corresponding processing element. In detail, the following annotations are supported:

_PE_LIBRARY Name describing the PE database the behavior is a member of. Used to distinguish between PE and other behaviors, i.e. required for PE behaviors.

_PE_CATEGORY Name of the category the PE belongs to. PE categories are mandatory and are used to classify PEs into different groups for PE allocation and selection.

_PE_COMMENT Brief description of the PE that will be used and displayed as an aid during selection and allocation.

_PE_BF_MODEL Name of the behavior providing a bus-functional model of the PE. The named behavior has to exist in the PE database. Required for PEs that have pre-defined, fixed interfaces on the pin level. See Section 3.2.

_PE_CA_MODEL Name of the behavior providing a cycle-accurate implementation model of the PE. The named behavior has to exist in the PE database. Required for PEs that have pre-defined, fixed hardware implementation. See Section 3.3.

Listing 5 shows an example of a simple PE behavior. The behavior defines a PE named *MyDSP* that is part of the "processor" library and belongs to the "DSP" category in that database. In addition, bus-functional and cycle-accurate models *MyDSP_BF* and *MyDSP_ISS* of the PE are made available.

3.1.2 Attributes

Basic characteristics and metrics of PEs are described through attribute annotations attached to the PE behaviors. Attribute annotations can be given using the format defined previously in Section 2.2.1. Specifically, the following attributes can be attached to PEs (Table 13):

_PE_CLOCK_FREQUENCY Clock frequency the PE is operating at. For PEs where the clock can be adjusted up to a maximum frequency, this should be an adjustable attribute.

_PE_MIPS MIPS performance rating of the PE, i.e. maximal throughput of the PE in million instructions per second.

_PE_COST Cost of the PE. The cost unit depends on the database (e.g. price or area).

_PE_POWER Power consumption rating of the PE, i.e. usually average power consumption when not idling.

_PE_INSTRUCTION_WIDTH Number of bits in a PE instruction word, i.e. width of the internal instruction bus.

_PE_DATA_WIDTH Number of bits in a PE data word, i.e. width of the internal data bus.

_PE_CHAR_WIDTH Number of bits in a basic machine character, i.e. width of smallest addressable unit.

_PE_DATA_MEMORY Size of internal data memory of PE. For PEs without separate program memory space, data memory size is the total size of combined program and data memory.

Name	Description	Type	Default
_PE_LIBRARY	Name of parent PE database	String	–
_PE_CATEGORY	Name of PE category	String	””
_PE_COMMENT	PE description	String	””
_PE_BF_MODEL	Name of bus-functional PE model	String	–
_PE_CA_MODEL	Name of cycle-accurate PE model	String	–

Table 12: General annotations for behavioral PE models.

```
behavior MyDSP();
note MyDSP._PE_LIBRARY   = "processors";
note MyDSP._PE_CATEGORY  = "DSP";
note MyDSP._PE_COMMENT   = "PE example";
note MyDSP._PE_BF_MODEL  = "MyDSP_BF";
note MyDSP._PE_CA_MODEL  = "MyDSP_ISS";
```

Listing 5: PE behavior example.

Name	Description	Unit	Type	Default
_PE_CLOCK_FREQUENCY	Clock frequency	Hz	double	0.0
_PE_MIPS	Throughput	MIPS	double	0.0
_PE_COST	Cost		double	0.0
_PE_POWER	Power consumption	W	double	0.0
_PE_INSTRUCTION_WIDTH	Instruction word width	bits	uint	0u
_PE_DATA_WIDTH	Data word width	bits	uint	0u
_PE_CHAR_WIDTH	Machine character width	bits	uint	8u
_PE_DATA_MEMORY	Size of internal data memory	byte	uint	0u
_PE_PROGRAM_MEMORY	Size of internal program memory	byte	uint	–

Table 13: PE behavior attribute annotations.

_PE_PROGRAM_MEMORY Size of internal program memory of PE for PEs with dedicated program memory space separate from data memory.

Note that using adjustable attributes for these annotations will define the amount of flexibility available in the PE template.

3.1.3 Weight Tables

Weight tables of PEs are described through weight table annotations attached to the PE behaviors. Weight table annotations can be given using the format defined previously in Section 2.2.3. Specifically, the weight annotations that can be attached to PEs are listed in Table 14. Each one is used by at least one design metric after mapping a system behavior to a system architecture. The weight annotations and examples of their represented design metrics are:

_PE_WEIGHT_OPERATION_STATIC Each item in the weight table represents the occupied instruction word or control word for an operation with a certain item type and data type in the PE.

For general purpose processors and DSPs, one design metric represents the number of of instructions that the system behavior contains after mapping the system behavior to the PE. For custom hardware, it represents the number of control words that the system behavior contains after synthesizing the system behavior to the PE.

_PE_WEIGHT_OPERATION_DYNAMIC Each item in the weight table represents the clock cycles required to execute an operation with a certain item type and data type in the PE.

One design metric represents the computation time of the design.

_PE_WEIGHT_TRAFFIC_STATIC Each item in the weight table represents the occupied machine characters by a storage unit (such as a variable) of the system behavior with a certain item type and data type in the PE. The machine character corresponds to the ”char” data type in the SpecC language.

This is used to compute the static traffic. One design metric represents the port width of the system behavior. It equals to the traffic when each port of the behavior is accessed once.

_PE_WEIGHT_TRAFFIC_DYNAMIC Each item in the weight table represents the occupied machine characters by a storage unit (such as a variable) of the system behavior with a certain item type and data type in the PE. The machine character corresponds to the ”char” data type in the SpecC language.

This is used to compute the dynamic traffic. One design metric represents the dynamic traffic of the design. It equals to the amount of traffic going through the ports of the system behavior during simulation.

Besides the weight annotation, the item header annotation and data header annotation are also required. The header annotations for operation metric set and traffic metric set are shown in Table 15, and Table 16 respectively. `_PE_WEIGHT_OPERATION_STATIC` and `_PE_WEIGHT_OPERATION_DYNAMIC` use header annotations in Table 15. `_PE_WEIGHT_TRAFFIC_STATIC` and `_PE_WEIGHT_TRAFFIC_DYNAMIC` use header annotations in Table 16.

3.1.4 Memory Tables

Memory tables attached to PE behaviors describe the layout of variables in the PE memory as determined by the compiler on top of the machine architecture. Memory table annotations follow the same basic format as for weight table annotations defined in Section 2.2.3. Specifically, the following annotations are used to define PE memory tables (Table 17):

_PE_MEMORY_SIZE Size of variables of basic type on the stack and the heap in machine characters as determined by the `sizeof()` operator.

_PE_MEMORY_ALIGNMENT Address alignment boundaries for variable of basic type on the stack and the heap.

Memory tables defined for the basic types will be used to compute memory weights (sizes) for all basic and composite types found in the design.

Memory tables for sizing and alignment share the same item and data header annotations shown in Table 18 where item and data types iterate over all types of memory (local/global) and all basic SpecC data types, respectively.

3.1.5 Computation Functionality

For intellectual-property (IP) PEs with fixed functionality in terms of the computation the PE can perform, the PE behavior in the PE database has to provide a high-level, abstract simulation model of the PE. The behavioral IP model will be plugged into the system for simulation and has to accurately model the computation performed by the IP including estimated timing. On the other hand, the behavioral IP model should exclude unnecessary implementation details in order to achieve the fastest possible simulation

Name	Metric	Unit	Type	Default
`_PE_WEIGHT_OPERATION_STATIC`	Code size	words	double	0.0
`_PE_WEIGHT_OPERATION_DYNAMIC`	Computation time	cycles	double	0.0
`_PE_WEIGHT_TRAFFIC_STATIC`	Static traffic	chars	double	0.0
`_PE_WEIGHT_TRAFFIC_DYNAMIC`	Dynamic traffic	chars	double	0.0

Table 14: PE behavior weight annotations.

Name	Type	Default
`_PE_WEIGHT_OPERATION_HEADER_DATATYPE`	in Table 10	–
`_PE_WEIGHT_OPERATION_HEADER_ITEMTYPE`	in Table 6 &Table 7	–

Table 15: Header annotations for operation metrics.

Name	Type	Default
`_PE_WEIGHT_TRAFFIC_HEADER_DATATYPE`	in Table 10	–
`_PE_WEIGHT_TRAFFIC_HEADER_ITEMTYPE`	in Table 8	–

Table 16: Header annotations for traffic metrics.

Name	Description	Unit	Type	Default
`_PE_MEMORY_SIZE`	Variable size	chars	uint	0u
`_PE_MEMORY_ALIGNMENT`	Variable alignment	chars	uint	0u

Table 17: PE behavior memory table annotations.

Name	Type	Default
`_PE_MEMORY_HEADER_DATATYPE`	in Table 10	””
`_PE_MEMORY_HEADER_ITEMTYPE`	in Table 9	””

Table 18: Header annotations for memory tables.

speeds. The IP model is a black-box model used for simulation only and therefore only needs to be functionally correct in terms of the input and output values that can be observed at its ports.

For such IPs, the PE database has to provide a SpecC body for the PE behavior. The body can be either internal or external. In the former case, the SIR file of the PE contains a complete SpecC definition of the PE behavior including its body. In the latter case, the PE database provides a shared library with the PE behavior's body to be linked against the design for simulation. The shared library and associated annotations at the PE behavior can be created via the IP mechanism of the SpecC compiler (see '`scc`' manual page, Section A.1).

Behavioral PE models of IPs communicate with the rest of the system at an abstract, message-passing level. PE behaviors of IPs must have ports that are of one of the following interface types out of the standard SpecC channel library:

- `i_sender`, `i_receiver`, or `i_tranceiver`
- `i_typed_sender`, `i_typed_receiver`, or `i_typed_tranceiver`

PE behaviors are not allowed to have ports of any other interface or standard type.

A simple example of an IP with pre-defined computation is shown in Listing 6. The IP has two ports for receiving and sending integers that get processed internally. The body of the IP's PE behavior provides SpecC code for the *main* method that simulates the IP's functionality (in this case, simple incrementation) and timing (through a `waitfor` statement modeling the IP's input-to-output delay). Since the IP behavior represents real hardware that never exits, the IP model is executing in an endless loop. Finally, the IP behavior provides the required general annotations for database management. Although not shown in the example, IPs supply standard attributes and can be made parameterizable like any other PE behavior.

3.1.6 Memory

```
mem_interface =
    interface interface_name
            '{' '}' ';'
```

Figure 7: Format of PE behavior memory interfaces.

For PEs that can act as system-level memories shared among other PEs, the PE behavior has to be turned into a server providing global storage. This is modeled by letting the PE behavior implement a channel interface through which other PEs can access the memory. The PE's memory interface in the database has to be empty and acts as a template that will be filled as part of the design flow in SCE. See Figure 7 for the general format of memory interface definitions where a memory PE behavior implements such an interface. In addition, memory interfaces have to have a number of general annotations for database management (Table 19):

`_PE_MEMORY` Boolean flag to distinguish PE behavior memory interfaces from other normal interfaces. True for memory interface templates.

An example of a system memory PE behavior is shown in Listing 7. The memory behavior *MyMem* implements the memory interface *IMyMem*. As required, the memory interface template is empty and has the proper annotation attached. Attached to the memory PE behavior are the normal general annotations which mark the behavior as a PE behavior in the "Memory" category and a bus-functional model in the form of the "MyMem_BF" behavior of the PE database (since the memory has a fixed, pre-defined pin-level interface). Since the PE is a pure system memory that can not provide any computational functionality (IP with fixed but empty computation, see Section 3.1.5), the memory's PE behavior does not have any ports and the *main* method is empty.

3.2 Bus-Functional PE Models

For PEs with fixed, pre-defined interfaces and communication functionality, the PE database has to contain a bus-functional model of the PE. A bus-functional PE model accurately describes the PE interface at the pin-level and it provides a simulation-model of the PE's communication aspects on top of any computation functionality as defined by the PE's behavioral model, if any (see Section 3.1).

Bus-functional models can be thought of as additional communication layers that wrap around the PE behavioral model. A bus-functional model can consist of several layers of behaviors that create a hierarchy or tree of behavior instantiations. At minimum, a top-level bus-functional layer has to exist that provides a pin-accurate model of the PE. Through this layer and its optional sublayer instance hierarchy, the bus-functional PE model describes the communication behavior of the PE at its pins and it has to provide the same computational functionality as the PE's behavioral model.

For IPs with fixed computation and communication functionality the bus-functional IP model provides a timing- and data-accurate descriptions in terms of signals that can be observed at the PE's pins. Bus-functional IP models only have to provide a single bus-functional layer

```
#include <c_typed_double_handshake.sh>

DEFINE_I_TYPED_SENDER(ipint, int)
DEFINE_I_TYPED_RECEIVER(ipint, int)

behavior IP(i_ipint_receiver input, i_ipint_sender output)
{
  note _PE_LIBRARY  = "processors";
  note _PE_CATEGORY = "IP";
  note _PE_COMMENT  = "IP example";
  note _PE_BF_MODEL = "IP_BF";
  note _PE_CA_MODEL = "IP_RTL";

  int data;

  void main(void)
  {
    while(true)
    {
      input.receive(&data);
      data++;
      waitfor(1000);
      output.send(data);
    }
  }
};
```

Listing 6: Example of PE behavior for IP with given functionality.

Name	Description	Type	Default
`_PE_MEMORY`	Flag for memory interface templates	bool	false

Table 19: Memory interface annotations.

```
interface IMyMem {
  note _PE_MEMORY = true;
};

behavior MyMem()
  implements IMyMem
{
  note _PE_LIBRARY = 1;
  note _PE_CATEGORY = "Memory";
  note _PE_BF_MODEL = "MyMem_BF";

  void main(void) {
  }
};
```

Listing 7: Example of PE behavior for system memory.

but they can consist of several hierarchical layers internally (Section 3.2.1). However, bus-functional IP models need to supply an adapter that wraps and abstracts communication with the IP up to the message-passing level equivalent to the IP's behavioral model (see Section 3.1.5).

For programmable PEs with flexible computation behavior (i.e. no functionality provided in the PE behavior, Section 3.1.5) but fixed, pre-defined interfaces and communication functionality, the bus-functional PE model has to provide a hierarchy with at least two layers: a top-level bus-functional layer describing the PE pin interface on the outside and an internal empty hardware abstraction layer (HAL) at the leaf of the bus-functional model hierarchy describing the interface for accessing the PE's communication implementation from the programmable computation on the inside (Section 3.2.2).

A PE is considered programmable in terms of its computation if the bus-functional PE model provides a hardware abstraction layer (HAL). As part of the design process in SCE, communication synthesis will use the HAL of the bus-functional PE model as a template and modify it to implement computation on the PE on top of the services provided by the HAL model. In terms of services, the HAL defines the insertion point for implementing the PE's computation and it provides communication services for stream and memory I/O and interrupt handling. The HAL together with the outer bus-functional layers model the corresponding capabilities of the pre-defined PE implementation (e.g. number and type of external interfaces, amount and level of interrupts, etc.).

The HAL marks the boundary between hardware and software in programmable PEs. SpecC code for the HAL and the subbehavior hierarchy inserted into the HAL by SCE will later be implemented in software. On top of an implementation of the HAL on the target processor taken out of the OS databases, target-specific code will be generated for the HAL model. Layers of the PE model above up to and including the bus-functional layer represent the hardware implementation of the PE and will later be replaced with a description of the real PE hardware taken out of the databases for manufacturing.

If no hardware abstraction layer is provided, on the other hand, the bus-functional PE model is considered to be a self-contained simulation model of the complete PE including computation and communication that will be plugged into the system simulation as is.

3.2.1 Bus-Functional Layer

The general format of bus-functional PE models is shown in Figure 8. The outer bus-functional layer is given as a SpecC behavior definition with an internal or external body, i.e. for IPs the body can be supplied as an external

```
bf_behavior =
    behavior pe_bf_name pins body_opt ';'

pins =
    '(' pin_list ')'

pin_list =
    pin
    | pin_list ',' pin

pin =
    pin_direction signal bit_sign bit_vector pin_name

pin_direction =
    <nothing>
    | in
    | out
    | inout

bit_sign =
    <nothing>
    | signed
    | unsigned

bit_vector =
    bit '[' const_expression ':' const_expression ']'
    | bit '[' const_expression ']'

body_opt =
    <nothing>
    | '{' internal_declarations_opt '}'
```

Figure 8: Format of bus-functional PE models.

library in the same manner as for the behavioral PE model (Section 3.1.5). The name of the bus-functional layer behavior defines the name of the bus-functional PE model and has to match the corresponding annotation at the PE behavior (see Section 3.1.1, `_PE_BF_MODEL` annotation). Finally, bus-functional PE behaviors have to have a list of ports of bitvector signal type representing the pins of the PE.

Bus-functional PE behaviors have a set of general purpose annotations for database and design flow management (Table 20):

`_PE_BF_BUS` Name of the bus channel in the bus protocol database that describes the protocol for communication with the PE. The pin interface of the bus-functional PE model has to match the list of wires in the bus protocol by name and type.

`_PE_BF_ADAPTER` Name of the channel providing the adapter wrapping the protocol stack for message-passing communication with the IP. The named channel has to exist in the PE database. See Section 3.2.4.

`_PE_HAL_MODEL` Name of the behavior providing the hardware abstraction layer (HAL) for the bus-functional PE model (see Section 3.2.2). The named behavior has to exist in the PE database. The HAL behavior has to be instantiated as a leaf in the tree of subbehaviors of the bus-functional PE model.

`_PE_INT_HANDLER` Name of the method in the HAL behavior that acts as the interrupt handler for the interrupt pin the annotation is attached to. See Section 3.2.2.

An example of a simple bus-functional PE model for the IP behavioral PE from Listing 6 is shown in Listing 8. The model has only one top-level bus-functional layer with an internal body that implements the IP's computation and communication functionality. Computation is the same as in the behavioral IP model introduced previously. Communication is modeled accurately by driving and sampling the PE pins. All in all, the bus-functional PE behavior provides an accurate simulation model of the IP for co-simulation with the rest of the system. When composing the system, the bus-functional IP model defines that in order to exchange data, other PEs communicating with the IP have to implement the protocol *IP_Bus* stored in the bus protocol database (Section 4). Apart from that, the IP comes with an adapter channel *IP_Adapter* that implements the complete protocol stack (including the bus protocol) for communication with the IP at the message-passing level (see Section 3.2.4).

3.2.2 Hardware Abstraction Layer

```
hal_behavior =
    behavior pe_hal_name port_list_opt
        implements_interface_opt
            '{' hal_body_list '}' ';'

hal_body_list =
    hal_body
    | hal_body_list hal_body

hal_body =
    note_definition
    | protocol_channel_instance
    | interrupt_handler_method
    | hal_main_method

interrupt_handler_method =
    void handler_name '(' void ')' '{' '}'

hal_main_method =
    void main '(' void ')' '{' '}'
```

Figure 9: Format of hardware abstraction layer (HAL) for bus-functional PE models.

The general format of the hardware abstraction layer (HAL) for bus-functional PE models is shown in Figure 9. The HAL is a SpecC behavior definition whose name defines the name of the HAL model and has to match the corresponding annotation at the high-level PE behavior (see Section 3.1.1, `_PE_HAL_MODEL` annotation). A HAL model generally has ports and implements interfaces to communication with the outer layers of the bus-functional PE model. In particular, the HAL layer implements an interface that makes its interrupt handler methods public in order for them to be invoked by the outer layers as required.

The HAL model has to provide an internal SpecC body that acts as a template to be filled by SCE. It has to contain exactly one *main* method (required by SpecC) that is empty. Furthermore, the HAL model has to define empty interrupt handler methods that are called by the outer layers of the bus-functional model whenever an interrupt is detected on one of the PE's interrupt pins. The name of a handler has to match the corresponding annotation attached to the interrupt pin in the bus-functional layer (see Section 3.2.1, `_PE_INT_HANDLER` annotation).

Finally, the HAL model has to provide access to the PE's communication services for use by the code that will be inserted in the design process by SCE. The bus-functional layer the HAL model is part of defines the protocol in the bus database that is used by the PE for external communication (see Section 3.2.1, `_PE_BF_BUS` annotation). The HAL then has to make all the high-level interfaces at the top of the protocol stack available (i.e. data link and memory access interfaces, see Section 4.3). Depending on the

Name	Description	Type	Default
`_PE_BF_BUS`	Name of the PE's bus protocol	String	–
`_PE_BF_ADAPTER`	Name of the wrapper for IPs	String	–
`_PE_HAL_MODEL`	Name of PE hardware abstraction layer (HAL) model	String	–
`_PE_INT_HANDLER`	Name of HAL interrupt handler method for an interrupt pin	String	–

Table 20: Bus-functional PE model annotations.

```
behavior IP_BF(in  signal bit[1]    ready,
               out signal bit[1]    ack,
                   signal bit[31:0] dat)
{
  note _PE_BF_BUS = "IP_Bus";
  note _PE_BF_ADAPTER = "IP_Adapter";

  int data;

  void main(void) {
    while(true)
    {
      wait(rising ready);       // receive data
      data = dat;
      waitfor(5);
      ack = 1b;

      data++;

      wait(falling ready);      // send data
      dat = data;
      ack = 0b;
    }
  }
};
```

Listing 8: Example of bus-functional PE model for simple IP.

boundaries between hardware and software in the PE, interfaces can be made available either as ports of the HAL model of corresponding interface type or by instantiating part or all of the protocol stack channels inside the HAL model. In either case, all bus communication available inside the HAL is eventually mapped to transactions on the PE pins through the hierarchy of layers of the bus-functional PE model.

An example of a bus-functional model for a programmable PE is shown in Listing 9. The bus-functional PE model consists of a hardware abstraction layer (Listing 9(a)) and an outer bus-functional layer (Listing 9(b)).

The HAL model (Listing 9(a)) imports the PE's external bus protocol out of the bus database in oder to instantiate its data link layer channel. The data link layer channel communicates with the lower layers of the bus protocol stack through a corresponding port of the HAL model. As required, the HAL model contains an empty *main* method and it provides a single interrupt handler *intHandler* that is made public through the HAL's implemented interface.

The top-level bus-functional layer (Listing 9(b)) then imports the HAL model and the bus-functional behavior *MyDSP_BF* instantiates the HAL behavior (line 22) and provides a pointer to the HAL model through the corresponding annotation (line 18). Inside the bus-functional behavior's `main` method, the HAL model is executed in a `try-interrupt` construct that models the interrupt control logic of the PE's hardware. Normal execution of the HAL is interrupted and the HAL's interrupt handler method is invoked whenever an interrupt (rising edge) on the PE's interrupt pin *Intr* is detected. As required, the annotation at the interrupt pin points to the HAL interrupt handler to reflect that fact (line 19). Finally, the bus-functional model provides the required pointer to the PE protocol in the bus database (line 17), it instantiates the lower layer channels of the protocol stack (line 21), and it maps the bus protocol onto the HAL's ports to complete the protocol implementation in the model.

3.2.3 Interrupt Handling

Bus-functional models for programmable PEs have to include a definition of the PE's interrupt capabilities and its interrupt handling. As described previously, the top-level bus-functional layer defines the interrupt pins available at the physical PE interface and the HAL model provides corresponding empty interrupt handler templates. The different layers of the bus-functional PE model then describe the PE's interrupt behavior of detecting interrupts, suspending regular computation, and executing the HAL interrupt handlers during system simulation. The example of Listing 9 shows how to model such behavior for a typical simple processor core with a single interrupt condition input.

In order to support more complex interrupt capabilities with more than one source of interrupts, different priorities, masking, etc. inside SCE, the PE database needs to include interrupt controllers as part of the bus-functional PE models. Interrupt controllers sit in front of the basic PE core model and are modeled by adding another layer to the bus-functional PE model between the processor core and the outer bus-functional layer. Typically, the interrupt controller provides a set of interrupt lines at the pins of the top-level bus-functional layer while internally communicating with the core via the PE bus and the core's interrupt condition input. The core then interrupts normal computation and executes the appropriate handler depending on the inputs received from the interrupt controller. Overall, the combination of layers has to simulate the proper interrupt behavior while maintaining the relationship between interrupt pins at the bus-functional layer and interrupt handlers in the HAL required by the database format for SCE.

An example of a bus-functional PE model for a programmable PE that includes an interrupt controller is shown in Listing 10. In contrast to the example shown previously, the hardware abstraction layer (Listing 10(a)) now contains templates for two separate interrupt handlers.

In the next layer (Listing 10(b)), the interrupt control logic in the model of the processor core hardware is modified to communicate with the interrupt controller. The interrupt control logic is triggered to interrupt normal computation on events at the core's interrupt condition line and bus control lines (line 33). Internally, the control logic processes pending interrupts as long as the interrupt condition is active and the bus is not busy (in order to maintain bus protocol timing, normal computation must not be interrupted in the middle of a bus transaction). If there is a pending interrupt, the interrupt logic communicates with the interrupt controller over the PE bus to acknowledge the interrupt and receive the interrupt vector. It then subsequently calls the corresponding interrupt handler in the HAL.

Finally, in the top-level bus-functional layer of the PE model (Listing 10(c)), the processor core model and the interrupt controller model are instantiated and connected via wires and pins of the bus-functional layer. Furthermore, the appropriate annotations are attached to the interrupt pins of the bus-functional layer pointing back to the interrupt handlers in the HAL.

An example of a simple interrupt controller model is shown in Listing 11. The interrupt controller (Listing 11(c)) consists of interrupt detection modules (one per interrupt line) and a control module that communicates via an interrupt status register (ISR). The interrupt detection behaviors (Listing 11(a)) model the edge detection on the interrupt lines, setting a bit in the status register when-

```
import "MyDSP_Bus";                          // bus protocol

interface IMyDSP_IntHandlers {
  void intHandler(void);
};

behavior MyDSP_HAL(IMyDSP_Bus_Protocol protocol)
  implements IMyDSP_IntHandlers
{
  MyDSP_Bus_LinkAccess link(protocol);   // bus data stream
  MyDSP_Bus_MemAccess  mem(protocol);    // bus memory access

  void intHandler(void) {
  }

  void main(void) {
  }
};
```

(a) Hardware abstraction layer (HAL)

Listing 9: Example of bus-functional model for programmable PE.

```
import "MyDSP_HAL";                 // hardware abstraction layer

// Interrupt control logic
behavior MyDSP_ICL(IMyDSP_Bus_Protocol bus, IMyDSP_IntHandlers handlers)
{
  void main(void) {
    handlers.intHandler();
  }
};

// Bus-functional layer
behavior MyDSP_BF(out signal bit[31:0] Addr,
                      signal bit[31:0] Data,
                  out signal bit[2]    Ctrl,
                  in  signal bit[1]    Int)
{
  note _PE_BF_BUS = "MyDSP_Bus";
  note _PE_HAL_MODEL = "MyDSP_HAL";
  note Int._PE_INT_HANDLER = "intHandler";

  MyDSP_Bus_Master protocol(Addr, Data, Ctrl);   // bus protocol
  MyDSP_HAL hal(protocol);
  MyDSP_ICL icl(protocol, hal);

  void main(void) {
    try {
      hal.main();
    }
    interrupt(rising Int) {
      icl.main();
    }
  }
};
```

(b) Bus-functional layer

Listing 9: Example of bus-functional model for programmable PE (continued).

```
import "MyDSP_Bus";                        // bus protocol

interface IMyDSP_IntHandlers {
  void intAhandler(void);
  void intBhandler(void);
};

behavior MyDSP_HAL(IMyDSP_Bus_Protocol protocol)
  implements IMyDSP_IntHandlers
{
  MyDSP_Bus_LinkAccess link(protocol);   // bus data link access
  MyDSP_Bus_MemAccess  mem(protocol);    // bus memory access

  void intAhandler(void) {
  }
  void intBhandler(void) {
  }

  void main(void) {
  }
};
```

(a) Hardware abstraction layer (HAL)

Listing 10: Example of bus-functional model for PE with interrupt controller.

ever an interrupt is detected. The control behavior (Listing 11(b)) watches the status register, signals interrupts to the core, and performs interrupt acknowledge cycles on the PE bus according to the priority of interrupts as long as there are interrupts pending.

3.2.4 IP Adapters

For IP PEs with fixed computation and communication functionality (see also Section 3.1.5), communication with the IP can generally not be synthesized arbitrarily but has to adhere to the proprietary IP protocol on all levels. Apart from the IP bus model that describes the lower layers of IP communication in the bus protocol database, bus-functional IP models therefore need to supply a complete protocol stack in the form of an IP adapter that covers communication with the IP from the abstract, message-passing level down to the pins of the IP bus.

IP adapters are SpecC channel definitions with an internal body as shown in Figure 10. The name of the channel defines the name of the adapter and has to match the corresponding annotation at the bus-functional PE model (see Section 3.2.1, _PE_BF_ADAPTER annotation).

On the one side, adapter channels connect to the IP through their list of ports of bitvector signal type that has to match the list of wires in the IP bus and hence the list of IP pins with reversed directions. On the other side, adapter channels export methods through implemented interfaces that provide message passing communication with the IP equivalent to the set of channels connecting to the ports of

```
ip_adapter_channel =
    channel adapter_name pins
        implements interfaces_opt
        '{' internal_declarations_opt '}' ';'

pins =
    '(' pin_list ')'

pin_list =
    pin
    | pin_list ',' pin

pin =
    pin_direction signal bit_sign bit_vector pin_name

pin_direction =
    <nothing>
    | in
    | out
    | inout

bit_sign =
    <nothing>
    | signed
    | unsigned

bit_vector =
    bit '[' const_expression ':' const_expression ']'
    | bit '[' const_expression ']'
```

Figure 10: Format of IP adapters.

```
import "MyDSP_HAL";               // hardware abstraction layer

// Interrupt control logic
behavior MyDSP_ICL(IMyDSP_Bus_Master bus, IMyDSP_IntHandlers handlers,
                   in signal bit[1] Int, in signal bit[1] Busy)
{
  void main(void) {
    unsigned bit[0:0] vec;
    while(Int && !Busy) {         // process pending interrupts
      vec = bus.ackIntr();        // acknowledge, receive vector
      switch(vec) {               // call corresponding interrupt handler
        case 0: return handlers.intAhandler();
        case 1: return handlers.intBhandler();
      }
    }
  }
};

// Processor core
behavior MyDSP_HW(out signal bit[31:0] Addr,
                      signal bit[31:0] Data,
                      signal bit[2]    Ctrl,
                  in  signal bit[1]    Int)
{
  MyDSP_Bus_Master protocol(Addr, Data, Ctrl);  // bus protocol
  MyDSP_HAL hal(protocol);
  MyDSP_ICL icl(protocol, hal, Int, Ctrl[0]);

  void main(void) {
    try {
      hal.main();
    }
    interrupt(Int, Ctrl) {
      icl.main();
    }
  }
};
```

(b) Processor core layer

Listing 10: Example of bus-functional model for PE with interrupt controller (continued).

```
import "MyDSP_HW";          // processor core
import "MyDSP_IC";          // interrupt controller

behavior MyDSP_BF(      signal bit[31:0] Addr,      // bus
                        signal bit[31:0] Data,
                        signal bit[2]    Ctrl,
                    in  signal bit[1]    intA,      // interrupts
                    in  signal bit[1]    intB)
{
  note _PE_BF_BUS = "MyDSP_Bus";
  note _PE_HAL_MODEL = "MyDSP_HAL";
  note intA._PE_INT_HANDLER = "intAhandler";
  note intB._PE_INT_HANDLER = "intBhandler";

  signal bit[1] Int = 0;                            // interrupt line

  MyDSP_HW hw(Addr, Data, Ctrl, Int);               // processor core
  MyDSP_IC ic(Addr, Data, Ctrl, Int, intA, intB);   // interrupt ctrl

  void main(void) {
    par {
      hw.main();
      ic.main();
    }
  }
};
```

(c) Bus-functional layer

Listing 10: Example of bus-functional model for PE with interrupt controller (continued).

interface type in the behavioral IP model (Section 3.1.5). For every port in the behavioral IP model, the IP adapter channel has to provide corresponding reverse methods. To avoid name clashes in the adapter, method names are suffixed with '_' plus the name of the port in the behavioral model.

Internally, adapter channels describe the implementation of IP communication by mapping each message down to its proper sequence of events on the IP bus wires for transferring data, synchronization, etc. IP communication can be described directly in the body of the adapter or hierarchically through a stack of other protocol channel instances (e.g. partly using IP bus protocol models out of the bus database).

An example of an IP adapter for communication with the IP introduced in Listing 8 is shown in Listing 12. The adapter connects to the IP bus through *Addr*, *Data*, and *Ctrl* ports. It implements *send_input* and *receive_output* methods corresponding to the *input* and *output* ports of the behavioral IP model. Internally, the methods drive and sample the bus wires according to the IP protocol to exchange data and synchronize with the IP.

3.3 Cycle-Accurate PE Models

For PEs with fixed functionality (i.e. programmable PEs with fixed communication functionality or IPs with fixed computation and communication functionality) the PE database has to contain a cycle-accurate implementation model of the PE in addition to the (mandatory in those cases) bus-functional PE model. At its interface, a cycle-accurate model has to match exactly the interface of its corresponding bus-functional model. Hence, a cycle-accurate model is a SpecC behavior with ports of signal bitvector type representing pins and the general format of cycle-accurate models is the same as for bus-functional models shown in Figure 8. Directions, types, names, and order of ports have to be the same as in the corresponding bus-functional model.

A cycle-accurate model has to have an external or internal body providing a clock cycle accurate simulation model of the PE implementation. Therefore, compared to the bus-functional model, the pin interface is the same but the timing is refined and more accurate.

For programmable PEs, the cycle-accurate model has to implement an instruction set simulation (ISS) that takes the name of the object code for the processor generated within SCE as a parameter and executes the instructions therein cycle by cycle.

```
behavior MyDSP_ICDetect(in signal bit[1] intr, out signal bit[1] flag)
{
  void main(void)
  {
    while(true) {
      wait(rising intr);
      flag = 1;
    }
  }
};
```

(a) Interrupt detection logic

```
import "MyDSP_Bus";       // bus protocol

behavior MyDSP_ICCtrl(IMyDSP_Bus_Slave bus, signal bit[2] ISR,
                      out signal bit[1] Int)
{
  void main(void) {
    while(true) {
      while(!ISR) wait(ISR);    // wait for interrupt

      Int = 1;                  // signal interrupt condition
      if(ISR[0])
        bus.ackIntr(0);         // wait for acknowledge...
      else if(ISR[1])           // ...and send vector
        bus.ackIntr(1);
      Int = 0;                  // reset interrupt condition
    }
  }
};
```

(b) Control logic

Listing 11: Example of interrupt controller model.

```
import "MyDSP_Bus";              // bus protocol

import "MyDSP_ICDetect";         // detection logic
import "MyDSP_ICCtrl";           // controller

behavior MyDSP_IC(in  signal bit[31:0] Addr,     // PE bus
                      signal bit[31:0] Data,
                  in  signal bit[2]    Ctrl,
                  out signal bit[1]    Int,
                  in  signal bit[1]    intA,     // interrupts
                  in  signal bit[1]    intB)
{
  MyDSP_Bus_Slave protocol(Addr, Data, Ctrl);    // bus protocol

  signal bit[2] ISR = 0;                         // status register

  MyDSP_ICDetect intAdetect(intA, ISR[0]);       // detection logic
  MyDSP_ICDetect intBdetect(intB, ISR[1]);

  MyDSP_ICCtrl ctrl(protocol, ISR, Int);         // control

  void main(void) {
    par {
      ctrl.main();
      intAdetect.main();
      intBdetect.main();
    }
  }
};
```

(c) Interrupt controller model

Listing 11: Example of interrupt controller model (continued).

```
channel IP_Adapter(out signal bit[1]    ready,
                   in  signal bit[1]    ack,
                       signal bit[31:0] dat)
  implements IIP_Adapter
{
  void send_input(int input)
  {
    dat = input;
    ready = 1b;
    wait(rising ack);
  }

  void receive_output(int &output)
  {
    ready = 0b;
    wait(falling ack);
    *output = dat;
  }
};
```

Listing 12: Example of IP adapter model.

For IPs, the cycle-accurate model in SpecC has to correspond to the RTL description of the IP that has to be available outside of SCE in the form of synthesizable (soft core) or gate-level (hard core) HDL (e.g. VHDL or Verilog) code.

3.3.1 Instruction Set Models

For an instruction set model, the body of the cycle-accurate PE behavior contains SpecC code that executes an instruction set simulation of the processor's object code generated through SCE. The instruction set simulation reads instructions from the supplied object file and executes them cycle by cycle while simulating their effects on the pins of the PE model. Instruction set models can trade of accuracy in terms of timing observed at the PE's pins versus simulation speed depending on the granularity of simulated cycles.

Existing instruction set simulators (ISS) for PEs can be plugged into the SpecC simulation as long as they provide a C level API that allows hooks into the simulation flow to observe and manipulate the PE pins. The cycle-accurate SpecC behavior in the PE database acts as a wrapper around calls to the ISS API and gets later linked against the ISS libraries to create the co-simulation executable. The wrapper code advances simulation time and drives and samples pins on the SpecC side by controlling the ISS such that both sides can be synchronized on events at the PE pin interface.

Listing 13 shows an example of such a simple SpecC instruction set model wrapper around an existing ISS for the PE example from Listing 9. After initialization of the ISS (line 15 and line 16), the instruction set model simulates one clock cycle of the ISS (line 21) in an endless loop (line 18). In each cycle, the model drives the ISS's asynchronous inputs (line 19) and it checks for any I/O instructions executed in the ISS. In case of external bus reads (line 23) or bus writes (line 27), the model simulates a corresponding bus cycle via calls to the PE's bus protocol model (line 10) imported from the bus protocol database (line 3). In case of a normal processor cycle, the instruction set model simply advances time on the SpecC side corresponding to the simulated time spent in the ISS (line 31).

3.3.2 IP Models

For IPs with fixed functionality, the cycle-accurate PE model has to provide a RTL description of the IP that can be plugged into the system simulation as is. The body of the cycle-accurate PE behavior can be internal or external where parts or all of the model can be supplied as an external shared library created through the IP mechanism of the SpecC compiler ('`scc`', Section A.1).

RTL descriptions for IPs in SpecC can be at any level, e.g. behavioral, FSMD-style, structural or even gate-level RTL descriptions are possible. Furthermore, if an IP model is available in the form of an external simulator that satisfies the requirements, it can be plugged in by providing a SpecC wrapper in the same manner as for ISS models described in Section 3.3.1. In any case, the PE database does not pose any restrictions on the body of cycle-accurate IP models and existing guidelines for RTL modeling can be followed freely.

4 Bus Protocols

The bus database contains models of busses including associated protocols where the term "bus" refers to communication structures in general, e.g. networks and their protocols. Models taken out of the bus database are used within SCE to implement and synthesize communication inside the PEs connected to the busses of the system.

Bus models in the bus database consist of a stack of protocol layers At the bottom of the stack, the physical layer is connected to the actual physical bus wires and it implements the bus primitives defined by the bus protocol for data transfers, synchronization and arbitration. On top of the physical layer, the media access layer provides an abstraction of external communication into data links and memory accesses by using and combining bus primitives to regulate media accesses and slice abstract data into bus words. Note that even though these layers are loosely based on the OSI reference model, they do not necessarily match the OSI definitions exactly.

Each protocol layer can have two separate sides with different implementations for bus masters and bus slaves. The different models of the two protocol layers for bus models are stored as SpecC channels or behaviors (for passive or active protocol models, respectively). Each channel or behavior provides a protocol implementation for one single PE connected to the bus, i.e. each connected PE implements a bus protocol by creating internal instances of the required protocol models. Physical layer models connect to the bus wires through ports of the model and pins of the PE. Higher-level models are stacked on top of each other via interfaces implemented by each model where a model calls the methods of the model beneath it via ports of corresponding interface type.

At the top-level of the bus database, all channels and behaviors that are part of the same bus model are then grouped together under a single, top-level bus channel that acts as a container representing the overall bus protocol in the bus database.

```
#include "iss.h"          // ISS C interface

import "MyDSP_Bus";       // bus protocol

behavior MyDSP_ISS(out    signal bit[31:0] Addr,
                   inout  signal bit[31:0] Data,
                   inout  signal bit[2]    Ctrl,
                   in     signal bit[1]    Int)
{
  MyDSP_Bus_Master protocol(Addr, Data, Ctrl);   // bus protocol

  void main(void) {
    bit[31:0] data;

    iss_startup();                      // initialize ISS
    iss_load(OBJECT_NAME);              // load program

    while(true) {                       // run simulation
      iss_intr = Int;                   // drive ISS input

      iss_exec();                       // run DSP cycle

      if(iss_IR == MOVEM_RD) {          // bus read?
        protocol.readLong(iss_AR, &data);
        iss_DR = data;
      }
      else if(iss_IR == MOVEM_WR) {     // bus write?
        protocol.writeLong(iss_AR, iss_DR);
      }
      else {                            // normal cycle
        waitfor(1e9 / CLOCK_FREQUENCY);
      }
    }
  }
};
```

Listing 13: Example of instruction set simulation (ISS) PE model.

4.1 Bus Channels

Bus channels represent bus protocols in the bus database. For each bus in the bus database there must be one bus channel and the set of bus channels defines the list of bus protocols available in the database. Bus channels are used for bus allocation and they act as a container for all protocol models and other information like attributes and parameters associated with a bus.

4.1.1 General Format

```
bus_channel =
    channel bus_name no_port_opt
        implements_interface_opt
        '{' bus_body_list '}' ';'

no_port_opt =
    <nothing>
    | '(' ')'

bus_body_list =
    bus_body
    | bus_body_list bus_body

bus_body =
    note_definition
    | wire_definition
    | instance_declaring_list
    | method_definition

wire_definition =
    signal bit_sign bit_vector wire_name
        initializer_opt ';'

bit_sign =
    <nothing>
    | signed
    | unsigned

bit_vector =
    bit '[' const_expression ':' const_expression ']'
    | bit '[' const_expression ']'

initializer_opt =
    <nothing>
    | '=' const_expression
```

Figure 11: Format of bus channels.

A bus channel is created through a special SpecC channel definition. The general format of bus channels in shown in Figure 11. The name of the bus channel defines the name of the bus protocol in the bus database. A bus channels does not have any ports but it can implement interfaces to optionally expose bus communication interfaces instantiated internally as described below for documentation and readability purposes.

A bus channel has to have an internal body that defines the set of wires of the bus and the protocol stacks for implementation of the bus protocol inside partners connected to and communicating via the bus. Bus wires are defined by instances of variables of signal bitvector type with optional initializers defining default values for driving a bus wire. Protocol stacks are given as instances of channels or behaviors describing protocol models at different layers. For each protocol model associated with the bus, an instance of that channel or behavior has to exist in the bus channel. Furthermore, every wire has to be connected to at least one channel or behavior instance. Mappings of wires to ports of low-level instances and of instances at lower layers to ports of instances of higher layers thereby define the proper connectivity to assemble protocol stacks inside PEs.

In addition to the definition of the communication hierarchy implementation, bus channels also carry a number of annotations for general database management, attributes (Section 4.1.2), and optional parameters (see Section 2.2.2).

General annotations for database management store basic information about the protocol organization (Table 21). Specifically, the following general annotations are attached to bus channels:

`_BUS_LIBRARY` Name describing the bus database the channel is a member of. Used to distinguish between bus and other channels, i.e. required for bus protocol channels.

`_BUS_CATEGORY` Name of the category the bus belongs to. Bus categories are mandatory and are used to classify busses into different groups for bus allocation and selection.

`_BUS_COMMENT` Brief description of the bus that will be used and displayed as an aid during selection and allocation.

`_BUS_ADDR_PORT` Name(s) of the variable(s) representing the wire(s) over which addresses are communicated. Either a single string or a complex annotation with a list of strings. Required for busses that support addressing.

`_BUS_DATA_PORT` Name(s) of the variable(s) representing the wire(s) over which data is communicated. Either a single string or a complex annotation with a list of strings. Required for all busses.

`_BUS_MASTER_PROTOCOL` Name of the channel or behavior providing the implementation of the data transfer protocol physical layer for bus masters. The

Name	Description	Type	Default
_BUS_LIBRARY	Name of parent bus database	String	–
_BUS_CATEGORY	Name of bus category	String	””
_BUS_COMMENT	Bus description	String	””
_BUS_ADDR_PORT	Name of the bus wire carrying addresses	String	–
_BUS_DATA_PORT	Name of the bus wire carrying data	String	–
_BUS_MASTER_PROTOCOL	Name of data transfer protocol master model	String	–
_BUS_SLAVE_PROTOCOL	Name of data transfer protocol slave model	String	–
_BUS_MASTER_INT	Name of synchronization protocol receiver model	String	–
_BUS_SLAVE_INT	Name of synchronization protocol sender model	String	–
_BUS_MASTER_ACCESS	Name of bus request model of arbitration protocol	String	–
_BUS_SLAVE_ACCESS	Name of bus grant model of arbitration protocol	String	–
_BUS_MASTER_LINK	Name of data link master model	String	–
_BUS_SLAVE_LINK	Name of data link slave model	String	–
_BUS_MASTER_MEM	Name of memory access model	String	–
_BUS_READ_METHOD	Name of media access layer read method	String	–
_BUS_WRITE_METHOD	Name of media access layer write method	String	–

Table 21: General annotations for bus protocol channels.

named channel or behavior has to exist in the bus database. Required for all busses. See Section 4.2.2.

_BUS_SLAVE_PROTOCOL Name of the channel or behavior providing the implementation of the data transfer protocol physical layer for bus slaves. The named channel or behavior has to exist in the bus database. Required for all busses. See Section 4.2.2.

_BUS_MASTER_INT Name of the channel or behavior providing the implementation of the synchronization protocol physical layer for bus masters, i.e. event detection on the receiver side. The named channel or behavior has to exist in the bus database. Required for busses that support interrupt lines. See Section 4.2.3.

_BUS_SLAVE_INT Name of the channel or behavior providing the implementation of the synchronization protocol physical layer for bus slaves, i.e. event generation on the sender side. The named channel or behavior has to exist in the bus database. Required for busses that support interrupt lines. See Section 4.2.3.

_BUS_MASTER_ACCESS Name of the channel or behavior providing the arbitration protocol implementation for bus accesses by bus masters. The named channel or behavior has to exist in the bus database. Required for busses where masters have to participate in a separate media access protocol among hosts attached to the bus. See Section 4.2.4.

_BUS_SLAVE_ACCESS Name of the channel or behavior providing the arbitration protocol implementation for bus accesses by bus slaves. The named channel or behavior has to exist in the bus database. Required for busses where masters have to participate in a separate media access protocol among hosts attached to the bus. See Section 4.2.4.

_BUS_MASTER_LINK Name of the channel providing the media access layer implementation for data links inside bus masters. The named channel has to exist in the bus database. Required for all busses. See Section 4.3.2.

_BUS_SLAVE_LINK Name of the channel providing the media access layer implementation for data links inside bus slaves. The named channel has to exist in the bus database. Required for all busses. See Section 4.3.2.

_BUS_MASTER_MEM Name of the channel providing the media access layer implementation for external memory accesses inside bus masters. The named channel has to exist in the bus database. Required for busses that support shared memories. See Section 4.3.3.

_BUS_READ_METHOD Name of the media access layer method for bus read accesses. The named method has to exist in all link and memory access media access layer channels (see Section 4.3.2 and Section 4.3.3). Required for all busses.

_BUS_WRITE_METHOD Name of the media access layer method for bus write accesses. The named method has to exist in all link and memory access media access layer channels (see Section 4.3.2 and Section 4.3.3). Required for all busses.

Note that different protocol model annotations can point to the same behavior or channel, i.e. a protocol implementation can provide more than one of the mandatory or optional functionalities within the restrictions noted in the rest of this Section 4.

An example of a bus channel is shown in Listing 14. The bus channel defines a bus protocol named *MyDSP_Bus* that is part of the "busses" library and belongs to the "Processor" category (the example being the protocol for the *MyDSP* processor from the PE database bus-functional examples, see Section 3.2). The bus channel annotations provides pointers to the different channels implementing the layers of the bus protocol model described in the following sections. The bus provides the mandatory models for the data transfer protocol physical layer (*MyDSP_Bus_Master* on the master side and *MyDSP_Bus_Slave* on the slave side) and the media access layer link (*MyDSP_Bus_Link* for both sides) implementations. The bus supports shared memory communication and it defines the corresponding media access layer implementations (*MyDSP_Bus_MemAccess* for performing memory accesses from a master). In both cases, link and memory type communication is provided through *read* and *write* methods of the corresponding channels. Finally, the bus defines a built-in synchronization protocol (*MyDSP_Bus_IntGenerate* for interrupt generation on the slave side and *MyDSP_Bus_IntDetect* for interrupt detection on the master side) but does not support arbitration to regulate bus accesses (no `_BUS_MASTER_ACCESS` annotation), i.e. it does not support multiple masters to be connected.

The bus channel example contains variable instances defining the basic bus wires (address bus *Addr*, data bus *Data*, and control lines *Ctrl*) and two built-in interrupt lines (*intA* and *intB*). On top of the wires, protocol stacks for the master and slave side of the bus are defined by instantiating all the channels named in the annotations in the proper hierarchy. Physical layer instances connect directly to the wires for basic bus (*master*, *slave*) and interrupt protocols (*masterIntX*, *slaveIntX*), respectively. On top of that, media access layer instances for links (*masterLink*, *slaveLink*) and memory accesses (*masterMem*) connect to the physical bus protocol implementations to use the bus primitives provided by them.

4.1.2 Attributes

Basic characteristics and metrics of busses and protocols are described through attribute annotations attached to the bus channels. Attribute annotations can be given using the format defined previously in Section 2.2.1. Specifically, the following attributes can be attached to bus channels (Table 22):

`_BUS_BANDWIDTH` Maximal bandwidth of the bus, i.e. throughput of the bus at 100% utilization.

`_BUS_COST` Cost of the bus. The cost unit depends on the database (e.g. price or area).

`_BUS_ADDR_WIDTH` Number of bits in bus addresses (e.g. number of wires in the address bus or bitwidth of address field in bus frames). Determines the size of the bus address space.

`_BUS_DATA_WIDTH` Number of bits in bus data words (e.g. number of wires in the data bus or bitwdith of data part in bus frames). Determines the maximal amount of data transfered in a single regular bus cycle.

`_BUS_CHAR_WIDTH` Number of bits in a basic bus character for memory accesses where bus characters are memory words aligned at unit address boundaries (one bus character per address), i.e. width of smallest addressable memory unit.

`_BUS_ALIGNMENT` Boundary at which whole bus data words are aligned in the bus address space, i.e. the address of a any part of a bus word modulo the alignment determines the address of the corresponding whole bus data word.

Note that bus character width multiplied by the bus alignment must not necessarily be equal to the bus data word width in case memory accesses use only partial data words, for example.

4.1.3 Weight Tables

Weight tables of buses are described through weight table annotations attached to the bus channels. Weight table annotations can be given using the format defined previously in Section 2.2.3. Specifically, the weight annotations can be attached to buses are listed in Table 23. Each one is used by at least one design metrics after mapping system channels to system buses. The weight annotations and examples of their represented design metrics are:

`_BUS_WEIGHT_TRAFFIC_STATIC` Each item in the weight table represents the occupied bus data words by a storage unit (such as a variable) of the system channel with a certain item type and data type on the bus.

This is used to compute the static traffic. One design metric represents the port width of the system channel. It equals to the traffic when each port of the channel is accessed once.

```
import "MyDSP_Bus_Int";          // interrupt protocol
import "MyDSP_Bus_Link";         // data link media access
import "MyDSP_Bus_Mem";          // memory media access

channel MyDSP_Bus() {
  note _BUS_LIBRARY = "busses";
  note _BUS_CATEGORY = "Processor";
  note _BUS_COMMENT = "Bus_example";
  note _BUS_ADDR_PORT = "Addr";
  note _BUS_DATA_PORT = "Data";
  note _BUS_MASTER_PROTOCOL = "MyDSP_Bus_Master";
  note _BUS_SLAVE_PROTOCOL = "MyDSP_Bus_Slave";
  note _BUS_MASTER_INT = "MyDSP_Bus_IntDetect";
  note _BUS_SLAVE_INT = "MyDSP_Bus_IntGenerate";
  note _BUS_MASTER_LINK = "MyDSP_Bus_LinkAccess";
  note _BUS_SLAVE_LINK = "MyDSP_Bus_LinkAccess";
  note _BUS_MASTER_MEM = "MyDSP_Bus_MemAccess";
  note _BUS_WRITE_METHOD = "write";
  note _BUS_READ_METHOD = "read";

  signal bit[31:0] Addr;                         // bus wires
  signal bit[31:0] Data;
  signal bit[2]    Ctrl = 00b;
  signal bit[1]    intA = 0b;                    // interrupt lines
  signal bit[1]    intB = 0b;

  MyDSP_Bus_Master       master(Addr, Data, Ctrl); // master stack
  MyDSP_Bus_LinkAccess   masterLink(master);
  MyDSP_Bus_MemAccess    masterMem(master);
  MyDSP_Bus_IntDetect    masterIntA(intA);
  MyDSP_Bus_IntDetect    masterIntB(intB);

  MyDSP_Bus_Slave        slave(Addr, Data, Ctrl); // slave stack
  MyDSP_Bus_LinkAccess   slaveLink(slave);
  MyDSP_Bus_IntGenerate  slaveIntA(intA);
  MyDSP_Bus_IntGenerate  slaveIntB(intB);
};
```

Listing 14: Bus channel example.

Name	Description	Unit	Type	Default
_BUS_BANDWIDTH	Max. throughput	bits/s	double	0.0
_BUS_COST	Cost		double	0.0
_BUS_ADDR_WIDTH	Address bus width	bits	uint	8u
_BUS_DATA_WIDTH	Data bus width	bits	uint	0u
_BUS_CHAR_WIDTH	Memory character width	bits	uint	8u
_BUS_ALIGNMENT	Bus word address alignment	addr	uint	1u

Table 22: Bus channel attribute annotations.

Name	Metric	Unit	Type	Default
_BUS_WEIGHT_TRAFFIC_STATIC	Static traffic	words	double	0.0
_BUS_WEIGHT_TRAFFIC_DYNAMIC	Dynamic traffic	words	double	0.0

Table 23: Bus channel weight annotations.

_BUS_WEIGHT_TRAFFIC_DYNAMIC Each item in the weight table represents the occupied bus data words by a storage unit (such as a variable) of the system channel with a certain item type and data type on the bus.

This is used to compute the dynamic traffic. One design metric represents the dynamic traffic of the design. It equals to the amount of traffic going through the ports of the system channel during simulation.

Besides the weight annotation, the item header annotation and data header annotation are also required. The header annotations are displayed in Table 24.

4.2 Physical Layer

Physical layer protocol models provide primitives for atomic bus transactions at their interfaces and they describe the basic timing of events (value changes) on the wires of the bus for each primitive. Timing diagrams of bus transactions are represented through state machines and associated timing constraints defining the possible sequences of driving and sampling bus wires. Physical layer protocol models are used in the SCE design flow to provide models of the protocol behavior on the wires of the bus both for system simulation and for synthesis of protocol implementations in actual hardware.

In general, a bus can have separate physical protocols for basic data transfers, synchronization, and arbitration. A protocol model provides a description for implementation of the protocol in a PE connected to the bus' wires. Each physical protocol can have two separate models with different implementations for bus master and bus slave type PEs.

4.2.1 General Format

The general format of physical layer protocol models is shown in Figure 12. Physical layer models are defined as either SpecC channels for passive models or SpecC behaviors for active models. In case of active models, the behavior defines a *main* method that has to be executed concurrently with the main computation inside a PE for busses where a PE needs to constantly participate in the protocol. For example, an active slave side might be necessary to answer and decline polling requests in case of a protocol that requires an acknowledge from slave to master to complete a transfer.

In both cases, the physical layer model defines a set of methods and exports them through an implemented interface such that higher layers can use physical layer services for communication by calling appropriate methods as needed. Internally, a physical layer channel or behavior

```
protocol_model =
    protocol_channel
        | protocol_behavior

protocol_channel =
    channel protocol_name pins
        implements interface_list
            '{' internal_declarations_opt '}' ';'

protocol_behavior =
    behavior protocol_name pins
        implements interface_list
            '{' internal_declarations_opt '}' ';'

pins =
    '(' pin_list ')'

pin_list =
    pin
    | pin_list ',' pin

pin =
    pin_direction signal bit_sign bit_vector pin_name

pin_direction =
    <nothing>
    | in
    | out
    | inout

bit_sign =
    <nothing>
    | signed
    | unsigned

bit_vector =
    bit '[' const_expression ':' const_expression ']'
    | bit '[' const_expression ']'
```

Figure 12: Format of physical layer protocol models.

Name	Type	Default
`_BUS_WEIGHT_TRAFFIC_HEADER_DATATYPE`	in Table 10	–
`_BUS_WEIGHT_TRAFFIC_HEADER_ITEMTYPE`	in Table 8	–

Table 24: Header annotations for traffic metrics.

models the corresponding protocol for driving and sampling the bus wires through their ports of bitvector signal type. The list of ports has to match a subset of the list of bus wires such that physical layer models can be connected to the wires through pins of the bus-functional PE.

4.2.2 Data Transfer Protocol

The mandatory data transfer protocol is the core of the bus and it describes primitives for transferring native bus words distinguished by bus addresses. It has to provides methods for all atomic bus cycles available over the core bus, including special types like burst mode, etc.

In case of a master/slave arrangement (separate models for master and slave side), the master side is actively initiating bus cycles and the slave slide can only passively listen on the bus for the start of a cycle to participate in. Consequently, methods on the slave side are considered blocking and only return when the transfer has been completed successfully with the corresponding master. In order to enable polling, methods on the master side, on the other hand, must not block even if no corresponding slave is available to successfully complete the transfer (note that in order to satisfy this requirement, active slaves might be necessary to answer and decline a transfer if no data is available through higher layers).

In case there is no distinction between masters and slaves (master model and slave model are the same), the single data transfer protocol model acts both as master and slave where for each transfer the sending side is assumed to be the master and the receiving side the slave. Apart from that, the same restrictions for the sending (master) and receiving (slave) methods apply.

Listing 15 and Listing 16 show an example of a data transfer protocol model. The data transfer protocol interface shown in Listing 15 provides primitives for reading and writing data operands of different sizes from/to the bus in one shot. Since bus addresses have to be aligned correctly, the slice of addresses accepted by each primitive depends on the size of the operand.

The example protocol is defined as two separate passive models in the form of channels for the master side (Listing 16(a)) and the slave side (Listing 16(b)). Both sides implement the same data transfer protocol interface *IMyDSP_Bus_Protocol* shown previously. Internally, the sequence of statements in the methods drive and sample the *Addr*, *Data*, and *Ctrl* wires of the bus through ports of the channels. Furthermore, the protocol state machines are enclosed in `do-timing` constructs to specify the constraints on timing that have to be obeyed when implementing the protocol.

4.2.3 Synchronization Protocol

As explained in the previous section, the data transfer protocol generally only supplies inherent one-way synchronization from master to slave. However, in order to implement reliable communication with guaranteed data delivery, two-way synchronization between communication partners is required. Therefore, a bus can supply an optional, distinct synchronization protocol to efficiently send events from slave to master. Usually, this means an interrupt protocol and interrupt wires through which a slave can send interrupts to a master.

In the same manner as data transfer protocols (see Section 4.2.2), interrupt protocols can be arranged as separate models for master side and slave side, or as one common model acting as both master (when sending) and slave (when receiving). In both cases, the master and slave sides have to implement the *i_receive* and *i_send* interfaces out of the standard SpecC channel library to receive and send events, respectively.

If the synchronization protocol annotations in the bus channel point to the data transfer protocol model(s), two-way synchronization with blocking transfers on both sides has to be implemented as part of the data transfer protocol, and no separate synchronization protocol is available or necessary. Similarly, if no synchronization protocol is supplied, no event transfer mechanism is available as part of the bus.

An example of a basic synchronization protocol to send interrupts from a slave to a master is shown in Listing 17. On the master side (Listing 17(a)), the interrupt detection logic channel receives events by recognizing rising edges on the interrupt line connected to its port. On the slave side (Listing 17(b)), the interrupt generation logic channel sends events by creating a pulse on the interrupt line through its port.

```
interface IMyDSP_Bus_Protocol
{
  void readByte(bit[31:0] addr, bit[7:0]  *data);
  void readWord(bit[31:1] addr, bit[15:0] *data);
  void readLong(bit[31:2] addr, bit[31:0] *data);

  void writeByte(bit[31:0] addr, bit[7:0]  data);
  void writeWord(bit[31:1] addr, bit[15:0] data);
  void writeLong(bit[31:2] addr, bit[31:0] data);
};
```

Listing 15: Example of a data transfer protocol interface.

```
channel MyDSP_Bus_Master(
    out signal bit[31:0] Addr,
        signal bit[31:0] Data,
    out signal bit[2]    Ctrl)
  implements IMyDSP_Bus_Protocol
{
  void readLong(bit[31:2] A,
                bit[31:0] *D) {
    do {
      l1: Addr = A @ 00b;
          Ctrl = 01b;
          waitfor(10);
      l2: *D = Data;
          Ctrl = 00b;
    } timing {
      range(l1; l2; 10; );
    }
  }

  void writeLong(bit[31:2] A,
                 bit[31:0] D) {
    do {
      l1: Addr = A @ 00b;
          Data = D;
          Ctrl = 11b;
          waitfor(10);
      l2: Ctrl = 00b;
    } timing {
      range(l1; l2; 10; );
    }
  }

  // ...
};
```

(a) Master side

```
channel MyDSP_Bus_Slave(
    in  signal bit[31:0] Addr,
        signal bit[31:0] Data,
    in  signal bit[2]    Ctrl)
  implements IMyDSP_Bus_Protocol
{
  void writeLong(bit[31:2] A,
                 bit[31:0] D) {
    do {
      l1: wait(rising Ctrl);
          if((A != Addr[31:2]) ||
             (Ctrl[1])) goto l1;
          waitfor(5);
      l2: Data = D;
    } timing {
      range(l1; l2; ; 10);
    }
  }

  void readLong(bit[31:2] A,
                bit[31:0] *D) {
    do {
          wait(rising Ctrl);
          if((A != Addr[31:2]) ||
             (!Ctrl[1])) goto l1;
      l1: waitfor(5);
      l2: *D = Data;
    } timing {
      range(l1; l2; ; 10);
    }
  }

  // ...
};
```

(b) Slave side

Listing 16: Example of data transfer protocol model.

```
import "i_receive";

channel MyDSP_Bus_IntDetect(
    in signal bit[1] intr
    )
  implements i_receive
{
  void receive(void)
  {
    wait(rising intr);
  }

};
```

(a) Interrupt detection logic

```
import "i_send";

channel MyDSP_Bus_IntGenerate(
    out signal bit[1] intr
    )
  implements i_send
{
  void send(void) {
    intr = 1;
    waitfor(5);
    intr = 0;
  }
};
```

(b) Interrupt generation logic

Listing 17: Example of synchronization protocol model.

4.2.4 Arbitration Protocol

If the bus supports multiple masters connected to the bus, it has to supply an arbitration protocol that is used to regulate accesses to the shared bus wires. In a centralized arbitration scheme, the master side of the arbitration protocol instantiated in each master communicates with the slave side of the arbitration protocol instantiated in an additional arbiter component attached to the bus. In a distributed arbitration scheme, there is no slave side of the arbitration protocol and the master sides of the protocol in each master regulate accesses among themselves.

The master side of the arbitration protocol can either be provided as a separate physical layer protocol model or it can be a built-in part of the data transfer protocol in which case the arbitration master annotation (_BUS_MASTER_ACCESS, see Section 4.1.1) points back to the data transfer protocol model. In case of a separate arbitration protocol, the master side arbitration protocol model has to implement the *i_semaphore* interface out of the standard SpecC channel library for acquiring and releasing access to the bus. In either case, arbitration has to be made available for each PE that will act as a data transfer protocol master if multiple masters sharing the bus are supported.

4.3 Media Access Layer

Media access layer models abstract accesses to the actual physical medium through the protocol into canonical interfaces for regulated, non-conflicting exchange or communication of data of arbitrary size and type. Hence, the media access layer regulates conflicting bus accesses in case the bus supports multiple masters through the bus arbitration protocol, and it slices data chunks into bus words or frames that are transmitted using the primitives (and possibly choosing among modes) of the bus data transfer protocol. Note that the media access layer does not implement any additional synchronization (e.g. through the synchronization protocol) but rather inherits the synchronization semantics from the underlying data transfer protocol.

The media access layer consists of two parts, models for implementation of bi-directional data links and models for accesses to shared memories connected to the bus. Mandatory data link models provide primitives to create point-to-point logical links for exchanging data between two communication partners attached to the bus. Optional memory access models are required if the bus supports shared memories and addressing of and access to PE storage. In both cases, media access layer models can consist of separate implementations for use in bus master and bus slave type PEs.

4.3.1 General Format

```
mac_model =
    mac_channel
        | mac_behavior

mac_channel =
    channel mac_name '(' port_list ')'
        implements interface_name
            '{' internal_declarations_opt '}' ';'

mac_behavior =
    behavior mac_name '(' port_list ')'
        implements interface_name
            '{' internal_declarations_opt '}' ';'
```

Figure 13: Format of media access layer models.

The general format of media access layer models is shown in Figure 13. In the same manner as physical layer models (see Section 4.2.1), active or passive media access layer models can be created through SpecC behavior or channel definitions, respectively. Media access models connect to underlying physical layer models for data transfers and arbitration through ports of corresponding interface type. For use by higher-level models created within SCE, a media access model has to define a set of methods in its body and export them through an implemented interface.

Since they will be accessed and used by automatically generated code, media access layer interfaces have to adhere to a certain canonical format. As defined in Figure 14, a media access layer interface defines a set of methods for transferring a block of data over the bus using a bus address to distinguish between different logical connections made over the same physical bus. Consequently, a media access method has no return value and takes three parameters: a bus address of integral type corresponding to the range of addresses available on the bus, a pointer to a data block, and the size of the data block in bytes. A media access layer interface has to define exactly two methods for reading and writing a block of data from/to the bus where the names of the methods have to match the corresponding annotations at the bus channel (see Section 4.1.1).

An example of a media access layer interface is shown in Listing 18. The interface has the two required methods *read* and *write* that each take address, data pointer, and length parameters. Corresponding to a 32-bit bus address range, address and length parameters are of `unsigned long` type.

4.3.2 Data Links

The data link part of the media access layer is used to transfer streams of data packets between logical endpoints inside PEs attached to the bus where two logical endpoints define a bi-directional, point-to-point logical link. Since streams only support sequential access (no random access), bus addresses are used to distinguish among different logical links on the bus only, i.e. data link models use the same bus address supplied as parameter for all bytes in a packet. Addresses supplied to data link model methods are used as addresses on the bus where addresses can be assumed to be aligned on bus word boundaries as defined in the bus channel annotations (see Section 4.1.1).

The data link part of the media access layer can provide different master and slave sides of the model if the underlying data transfer protocol in the physical layer differentiates between bus masters and slaves and if separate functionality is needed. For example, acquiring and releasing access to the bus through calls to the arbitration protocol is only needed on the master side, if supported by the bus at all.

Listing 19 shows and example of a data link media access layer model. This example shows a model that can be used for both masters and slaves assuming that no arbitration is present. If arbitration is supported by the physical layer, the calls to data protocol primitives in the data link model on the master side would have to be enclosed in arbitration method calls. Apart from arbitration, the data link model slices the packet of data supplied as a parameter to the *read* and *write* methods into bus words using data transfer protocol primitives for byte and long word transfers to transmit as much data as possible in each bus cycle. The same bus address supplied as method parameter is used for all transfers of the packet.

4.3.3 Memory Access

The memory access part of the media access layer provides methods for accessing bytes of data stored in a shared memory PE attached to the bus. Since memories need to support random access, data bytes in all memories attached to the bus have to be individually distinguishable. Therefore, bus addresses are used to select among different characters stored in memory where each character holds a certain amount of bytes as defined by the bus (see Section 4.1.1) and where consecutive bytes in memory are accessed as consecutive characters on the bus. Consequently, for each memory access the address supplied is the address of the first character in the block of data to be accessed and the length of the block divided by the bus character size determines the range of bus addresses accessed. Generally, addresses supplied by higher layers generated through SCE cannot be assumed to be properly aligned in any way, i.e. the memory access model has to take care of properly aligning data on the bus.

A media access layer memory model consists of master and slave sides for initiating and serving shared memory accesses over the bus. The master side provides methods with names matching the corresponding bus channel annotations (Section 4.1.1) for reading and writing blocks of data bytes from/to memory used in PEs that access shared storage over the bus. The slave side, on the other hand, provides methods for serving incoming random memory accesses used in PEs that provide shared storage (e.g. dedicated shared memory PEs).

An example of a memory access client model is shown in Listing 20. As in the example of the data link model shown in the previous section, the memory access model slices the block of data into bus words and transfer the data using primitives provided by the physical layer data transfer protocol model. Similarly, in case the physical layer of the bus supports arbitration, data protocol primitives have to be enclosed in arbitration call to regulate bus accesses on the bus master side. In contrast to the data link model,

```
mac_interface =
        '{' mac_declaration_list '}' ';'

mac_declaration_list =
        mac_declaration
        | mac_declaration_list mac_declaration

mac_declaration =
        note_definition
        | mac_method_declaration

mac_method_declaration =
        void method_name '(' addr_param ',' data_param ',' len_param ')' ';'

addr_param =
        integral_type_specifier param_name

data_param =
        void* param_name

len_param =
        integral_type_specifier param_name
```

Figure 14: Format of media access layer interfaces.

```
interface IMyDSP_Bus_Access
{
  void write(unsigned long addr, void* data, unsigned long len);
  void read(unsigned long addr, void* data, unsigned long len);
};
```

Listing 18: Example of media access layer interface.

```
import "MyDSP_Bus_Protocol";     // physical layer

import "IMyDSP_Bus_Access";      // MAC interface

channel MyDSP_Bus_LinkAccess(IMyDSP_Bus_Protocol protocol)
  implements IMyDSP_Bus_Access
{
  void write(unsigned long addr, void* data, unsigned long len) {
    unsigned char* p;

    for(p = (unsigned char*)data ; len >= 4; len -= 4, p += 4)
      protocol.writeLong(addr[31:2], (*p)[7:0] @ (*(p+1))[7:0] @
                                   (*(p+2))[7:0] @ (*(p+3))[7:0]);
    for( ; len; len--, p++)
      protocol.writeByte(addr, *p);
  }

  void read(unsigned long addr, void* data, unsigned long len) {
    unsigned char* p;
    bit[31:0] l;
    bit[7:0]  b;

    for(p = (unsigned char*)data ; len >= 4; len -= 4, p += 4) {
      protocol.readLong(addr[31:2], &l);
      *p = l[31:24]; *(p+1) = l[23:16]; *(p+2) = l[15:8]; *(p+2) = l[7:0];
    }
    for( ; len; len--, p++) {
      protocol.readByte(addr, &b);
      *p = b;
    }
  }
};
```

Listing 19: Example of media access layer data link model.

```
import "MyDSP_Bus_Protocol";     // physical layer

import "IMyDSP_Bus_Access";      // MAC interface

channel MyDSP_Bus_MemAccess(IMyDSP_Bus_Protocol protocol)
  implements IMyDSP_Bus_Access
{
  void write(unsigned long addr, void* data, unsigned long len) {
    unsigned char *p;

    for(p = (unsigned char*)data; len && (addr % 4); p++, len--)
      protocol.writeByte(addr++, *p);
    for(; len >= 4; p += 4, len -= 4)
      protocol.writeLong(addr[31:2]++, (*p)[7:0] @ (*(p+1))[7:0] @
                                     (*(p+2))[7:0] @ (*(p+3))[7:0]);
    for(; len; p++, len--)
      protocol.writeByte(addr++, *p);
  }

  void read(unsigned long addr, void* data, unsigned long len) {
    unsigned char *p;
    bit[31:0] l;
    bit[7:0]  b;

    for(p = (unsigned char*)data; len && (addr % 4); p++, len--){
      protocol.readByte(addr++, &b);
      *p = b;
    }
    for(; len >= 4; p += 4, len -= 4) {
      protocol.readLong(addr[31:2]++, &l);
      *p = l[31:24]; *(p+1) = l[23:16]; *(p+2) = l[15:8]; *(p+2) = l[7:0];
    }
    for(; len; p++, len--){
      protocol.readByte(addr++, &b);
      *p = b;
    }
  }
};
```

Listing 20: Example of media access layer memory access master model.

however, the memory access model also chooses data protocol primitives such that proper alignment is ensured. Furthermore, the model generates correct bus addresses by incrementing addresses for consecutive characters in the data block.

5 RTL Units

The RTL database contains models of register transfer level units such as functional units, storage units and local busses. RTL units are represented by SpecC behaviors in the RTL database. For each RTL unit, a corresponding behavior declaration needs to exist in the database.

The RTL units will be used for RTL component allocation and the generation of the final RTL netlist. During the RTL synthesis process, operations, variables and data transfers in the behavioral description of the design are bound to these RTL units.

5.1 RTL Behaviors

RTL behaviors represent the RTL units in the RTL database. For each RTL unit, there must be one RTL behavior which describes its basic characteristics in form of attributes and parameters.

5.1.1 General Format

An RTL behavior is created through a special SpecC behavior definition. The general format of RTL units is shown in Figure 15. The name of the behavior defines the name of the RTL unit in the RTL database. RTL behaviors should accurately reflect the component ports in terms of number and order.

RTL behaviors carry a number of annotations for general database management, attributes (Section 5.1.2), and optional parameters (see Section 2.2.2).

Table 25 lists the general annotations that are attached to RTL behaviors for database management and in order to describe basic information about the corresponding RTL unit. In detail, the following annotations have to be supplied:

_RT_LIBRARY Name describing the RTL database the behavior is a member of. Used to distinguish between RTL and other behaviors, i.e. required for RTL behaviors.

_RT_CATEGORY Name of the category the RTL unit belongs to. RTL unit categories are mandatory and are used to classify RTL units into different groups for RTL allocation and selection.

```
rtl_behavior =
    behavior unit_name ports body_opt ';'

body_opt =
    <nothing>
    | '{' internal_declarations_opt '}' ';'

ports =
    '(' port_list ')'

port_list =
    port
    | port_list ',' port

port =
    port_direction bit_sign bit_vector port_name

port_direction =
    <nothing>
    | in
    | out
    | inout

bit_sign =
    <nothing>
    | signed
    | unsigned

bit_vector =
    bit '[' const_expression ':' const_expression ']'
    | bit '[' const_expression ']'
```

Figure 15: Format of RTL unit behavior models.

Name	Description	Type	Default
_RT_LIBRARY	Name of parent RTL database	String	–
_RT_CATEGORY	Name of RTL unit category	String	””
_RT_COMMENT	RTL unit description	String	””
_RT_CLASS	RTL unit classification (see Table 26)	uint	0u
_RT_DATATYPE	Type of data processed by RTL unit (see Table 27)	uint	0u

Table 25: General annotations for RTL unit behavior.

_RT_COMMENT Brief description of the RTL unit that will be used and displayed as an aid during selection and allocation.

_RT_CLASS Class the RTL unit belongs to, i.e. type of unit. See Table 26.

_RT_DATATYPE Data type of the RTL unit, i.e. type of data the unit can process. See Table 27.

Value	Class
0	Functional unit
1	Memory
2	Register
3	Register file
4	Bus

Table 26: Classification of RTL units.

RTL units must be classified into 5 different categories: functional units, memories, registers, register files and local busses. Functional units can perform operations like addition, subtraction, multiplication, and so on. A function unit may take more than one clock cycle to perform an operation (multi-cycle operation), or can be pipelined to reduce the clock period.

Secondly, storage units can store data which will be used in the next computation. These include memories, registers, and register files.

Finally, local busses are wires used to transfer data between functional units and storage units.

Value	Datatype
0	Integral
1	Floating point
2	Fixed point

Table 27: RTL unit data types.

The behavioral description of a design contains different data types of variables. These data types are classified as: integral type, floating point type, and fixed point type as shown in Table 27. RTL units perform operations with these types of operands and store results in the same type.

5.1.2 Attributes

Basic characteristics and metrics of RTL units are described through attribute annotations attached to the RTL behaviors. Attribute annotations can be given using the format defined previously in Section 2.2.1. Specifically, the following attributes can be attached to an RTL behavior (Table 28):

_RT_COST Cost of the RTL unit. The cost unit depends on the database (e.g. price or area).

_RT_POWER Power consumption rating of the RTL unit, i.e. usually average power consumption when not idling. The power unit is W.

_RT_AREA Area of the the RTL unit. The area unit is mm^2

_RT_SIZE The size of the storage unit in words, i.e. number of words which the storage unit can store.

_RT_BITWIDTH The bit width of the RTL unit, i.e. how wide the RTL unit is.

_RT_DECIMAL_WIDTH The bit width for mantissa of floating point type unit or for fractional part of fixed point type unit.

_RT_DELAY Worst case delay for the RTL unit. The unit is *ns*.

_RT_INPUT_DELAY The delay from the input to the internal data buffer for multi-cycle/pipelined units. The default value is 0 *ns*. It also indicates the write time to write a value to the memory. This attribute is associated with clocked units.

_RT_OUTPUT_DELAY The delay to obtain the output data. For storage units, it indicates the read time to read a value from the memory. This attribute is associated with clocked units.

_RT_STAGES This indicates how many pipeline stages the RTL unit has. The default value is 0u.

_RT_INTERVAL This indicates how often the RTL unit can receive input data in number of clock cycle (data introduction interval). The default value is 0u

Name	Description	Unit	Type	Default
`_RT_COST`	Cost		double	0.0
`_RT_POWER`	Power consumption	W	double	0.0
`_RT_AREA`	Physical size	mm^2	double	0.0
`_RT_SIZE`	Storage size	words	ulong	0ul
`_RT_BITWIDTH`	Data width	bits	uint	8
`_RT_DECIMAL_WIDTH`	Fractional part width	bits	uint	0u
`_RT_DELAY`	Worst case stage delay	ns	double	0.0
`_RT_INPUT_DELAY`	First stage delay	ns	double	0.0
`_RT_OUTPUT_DELAY`	Last stage delay	ns	double	0.0
`_RT_STAGES`	Number of pipeline stages		uint	0u
`_RT_INTERVAL`	Data introduction interval		uint	0u

Table 28: RTL behavior attribute annotations.

Name	Description	Type	Default
`_RT_PIN_TYPE`	port type	unsigned int	0u
`_RT_ACTIVE_HIGH`	the port is active high or not	bool	true
`_RT_CLK_POS`	clock polarity of capturing data	bool	true

Table 29: RTL unit port attribute annotations.

Also, the ports of RTL units have some attributes as shown in Table 29.

`_RT_PIN_TYPE` Type of port. See Table 30.

`_RT_ACTIVE_HIGH` This indicates whether the port is active high or not. e.g. if reset is asserted to high and the design is reset, then reset port is active high.

`_RT_CLK_POS` The clock polarity when the input data is sampled. This attribute is associated with each data port. e.g. if data is sampled at the positive edge of the clock, this attribute for the data port should be set to true.

Value	Port type
0	Data
1	Control
2	Address
3	Clock
4	Reset

Table 30: RTL unit port types.

The port types of an RTL unit are classified as: data type, control type, address type, clock type, and reset type as listed in Table 30.

5.1.3 Operations

The annotation `_RT_OPERATIONS` defines the operations which the RTL unit can perform and the associated I/O ports and control ports. The operations in the RTL input description will be bound to the RTL units which can perform these operations. Table 31 lists all operations which can be used in operation field.

In addition, we should have a way to describe IP units like MAC, DCT, and so on. Because the functionality of an IP unit can't be described by a primitive SpecC operator, we use a function call to point to the IP unit in the operation field (string in Table 31. In the design, if the designer wants to map a function call `mac` to MAC unit, the MAC unit should have `mac` in the operation field of `_RT_OPERATIONS`. Note that only global functions are supported for this mapping.

Listing 21 shows `_RT_OPERATIONS` of an ALU unit which can perform the arithmetic operations addition ($+$), subtraction ($-$), increment ($++$), decrement ($--$) and comparison ($==$, $!=$), as well as the logic operations logic and/or (&&, $||$). These operations and the associated I/O ports and control ports are defined by `_RT_OPERATIONS` in the ALU unit. For example, as shown in line 12 in Listing 21, control input `ctrl` of ALU should be set to 0x0000 for the ALU to perform addition of values at input port `a` and `b`. The result of the addition will come out of the output port `sum`.

5.2 Examples

In this section, we will take a closer look at how we can describe RTL components with examples. We will describes how functional units and storae units and busses look like.

```
_RT_OPERATIONS =
        '{' operation_list '}'

operation_list =
        operation
        | operation_list ',' operation

operation =
        '{' name ',' '{' ports_list '}' '}'

ports_list =
        ports
        | ports_list ',' ports

ports =
        '{' output_ports ',' input_ports ',' '{' control_list '}' '}'

output_ports =
        port_name
        | '{' port_name_list '}'

input_ports =
        port_name
        | '{' port_name_list '}'

port_name_list =
        port_name
        | port_name_list ',' port_name

control_list =
        control
        | control_list ',' control

control =
        '{' port_name ',' const_expression '}'
```

Figure 16: Format of _RT_OPERATIONS annotation.

Operators	Description	Operators	Description
++	post-increment	−−	post-decrement
++:	pre-increment	−−:	pre-decrement
+:	positive	−:	negative
∼	not	∗	multiplication
/	division	%	modulo
+	addition	−	subtraction
<<	shift left	>>	shift right
<	less than	>	greater than
<=	less than or equal to	>=	greater than or equal to
==	equal to	!=	not equal to
&	bitwise and	∧	bitwise exclusive or
|	bitwise or	&&	logical and
||	logical or	!	logical not
=	assignment (write data to storage unit)	[]	array access (read data from storage unit)
string	name of function call for IP	@	concatenation

Table 31: operators in _RT_OPERATIONS.

```
#ifndef BITWIDTH
#define BITWIDTH 32u
#endif

behavior alu(out signal unsigned bit[BITWIDTH-1:0] sum,
             out signal unsigned bit[0:0]          status,
             in  signal unsigned bit[BITWIDTH-1:0] a,
             in  signal unsigned bit[BITWIDTH-1:0] b,
             in  signal unsigned bit[3:0]          ctrl);

note alu._RT_OPERATIONS = {
 { "+",   {{{"sum"},    {"a", "b"}, {{"ctrl", 0000ub} }}}},
 { "++",  {{{"sum"},    {"a"},      {{"ctrl", 0000ub}, {"b", 0001ub}}},
           {{"sum"},    {"b"},      {{"ctrl", 0000ub}, {"a", 0001ub}}}}},
 { "++:", {{{"sum"},    {"a"},      {{"ctrl", 0000ub}, {"b", 0001ub}}},
           {{"sum"},    {"b"},      {{"ctrl", 0000ub}, {"a", 0001ub}}}}},
 { "-",   {{{"sum"},    {"a", "b"}, {{"ctrl", 00001ub} } } } },
 { "--",  {{{"sum"},    {"a"},      {{"ctrl", 0001ub}, {"b", 0001ub}}}}},
 { "--:", {{{"sum"},    {"a"},      {{"ctrl", 0001ub}, {"b", 0001ub}}}}},
 { "-:",  {{{"sum"},    {"b"},      {{"ctrl", 0001ub}, {"a", 0000ub}}}}},
 { "~",   {{{"sum"},    {"a"},      {{"ctrl", 0010ub} }}}},
 { "&",   {{{"sum"},    {"a", "b"}, {{"ctrl", 0011ub} }}}},
 { "|",   {{{"sum"},    {"a", "b"}, {{"ctrl", 0100ub} }}}},
 { "^",   {{{"sum"},    {"a", "b"}, {{"ctrl", 0101ub} }}}},
 { ">=",  {{{"status"}, {"a", "b"}, {{"ctrl", 0110ub} }}}},
 { ">",   {{{"status"}, {"a", "b"}, {{"ctrl", 0111ub} }}}},
 { "==",  {{{"status"}, {"a", "b"}, {{"ctrl", 1000ub} }}}},
 { "!=",  {{{"status"}, {"a", "b"}, {{"ctrl", 1001ub} }}}},
 { "<=",  {{{"status"}, {"a", "b"}, {{"ctrl", 1010ub} }}}},
 { "<",   {{{"status"}, {"a", "b"}, {{"ctrl", 1011ub} }}}}
};
```

Listing 21: Example of `_RT_OPERATIONS` annotation.

5.2.1 Functional units

Listing 22 shows an example of a simple behavior for an RTL unit. The behavior named `alu` is part of the ”rtl” library and belongs to the ”Functional Unit” category. The delay, area, cost and power of `alu` are configurable attributes and are scaled by the bit width. The `alu` is not pipelined as shown in line 23. The `main` function body describes the functionality of the `alu` including timing. It contains `while` statement to describe that `alu` runs for good and `wait` statement with sensitivity list for simulation. Input ports in combinational circuit can be included in a set of sensitivity list. To describe the delay of the unit, we use `waitfor` statement as shown in line 41. Note that the value in `waitfor` statement is as same as that of the `RT_DELAY` attribute.

Now, we will show how to describe a pipeline functional unit. Listing 23 shows an example of a 3-stage pipline multiplier unit which is named to behavior `mult_p`. In order to describe the behavior of pipeline stages, we may use sub-behaviors such as `stage1`, `stage2` and `stage3` in the `mult_p`. These sub-behaviors are runnig in parallel as shown in Listing 23. In the behavior `stage1`, the value of input ports is latched to internal registers. The behavior `stage2` performs multiplication and stores its result to an internal register. In the behavior `stage3`, the result of the multiplication will be latched to output port. This unit is working at rising edge of clock.

5.2.2 Storage units

Listing 24 shows a simple register `reg` which samples data at the rising edge of a clock when `load` is asserted and otherwise, outputs the data after output delay. The clock `clk` is included in sensitivity list because this unit is edge-triggered sequential circuit.

Listing 25 shows a register file `reg` which has two read ports and one write port. It samples data at the rising edge of a clock and stores it into an internal memory `buf` with an address. It also outputs the data from the memory with the addres. This unit has 3 configurable parameters such as bit width, address width and size.

5.2.3 Busses

Listing 26 shows a bus which is a wire in RTL design. Even though the bus unit is defined in RTL component library to contain its attributes, it will be instantiated in a design. Instead, we will use signal variables to represent busses in the design.

References

[1] A. Gerstlauer, R. Dömer, J. Peng, D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[2] R. Dömer, A. Gerstlauer, D. Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium (STOC), Japan, December 2002.

[3] SpecC Technology Open Consortium. http://www.specc.org.

```
#define BITWIDTH_DFLT    32u

#ifndef BITWIDTH
#define BITWIDTH BITWIDTH_DFLT
#endif

behavior alu(out signal unsigned bit[BITWIDTH-1:0] sum,
             out signal unsigned bit[0:0]          status,
             in  signal unsigned bit[BITWIDTH-1:0] a,
             in  signal unsigned bit[BITWIDTH-1:0] b,
             in  signal unsigned bit[3:0]          ctrl);

note alu._RT_LIBRARY    = "rtl";
note alu._RT_CATEGORY   = "Functional Unit";
note alu._RT_CLASS      = 0u;
note alu._RT_DATATYPE   = 0u;
note alu._RT_BITWIDTH   = BITWIDTH;
note alu._RT_DELAY      = 0.52 * (BITWIDTH/4);
note alu._RT_STAGES     = 0u;
note alu._RT_AREA       = BITWIDTH * 0.000297;
note alu._RT_COST       = BITWIDTH * 2.8;
note alu._RT_POWER      = BITWIDTH * 0.00005;
note alu._RT_OPERATIONS = {
  { "+",   { { {"sum"}, {"a", "b"}, { {"ctrl", 0000ub} } } } }
  // ...
};

note alu._SIR_PARAMETERS = {
  { "BITWIDTH", BITWIDTH_DFLT, { {2u, 64u} }, "Bitwidth", "bits" }
};
```

(a) Declaration

Listing 22: Example of RTL unit.

```
#include "alu.sh"        // declaration

behavior alu(out signal unsigned bit[BITWIDTH-1:0] sum,
             out signal unsigned bit[0:0]          status,
             in  signal unsigned bit[BITWIDTH-1:0] a,
             in  signal unsigned bit[BITWIDTH-1:0] b,
             in  signal unsigned bit[3:0]          ctrl)
{
  note ctrl._RT_PIN_TYPE = 1u;

  void main(void)
  {
    unsigned bit[BITWIDTH-1:0] res;

    while (1)
    {
      wait(a, b, ctrl);
      switch(ctrl)
      {
        case 0000ub: res = a + b;
                     break;
        // ...
      }
      waitfor 0.52*(BITWIDTH/4);
      sum = res;
    }
  }
};
```

(b) Definition

Listing 22: Example of RTL unit (continued).

```
#define BITWIDTH_DFLT     32u

#ifndef BITWIDTH
#define BITWIDTH BITWIDTH_DFLT
#endif
```

(a) Macros

```
#include "mult.sh"        // macros

behavior mult (out signal unsigned bit[BITWIDTH-1:0] result ,
               in  signal unsigned bit[BITWIDTH-1:0] a,
               in  signal unsigned bit[BITWIDTH-1:0] b,
               in  signal unsigned bit[0:0]          clk);

note mult._RT_LIBRARY     = "rtl";
note mult._RT_CATEGORY    = "Functional␣Unit";
note mult._RT_CLASS       = 0u;
note mult._RT_DATATYPE    = 0u;
note mult._RT_BITWIDTH    = BITWIDTH;
note mult._RT_DELAY       = 0.60 * (BITWIDTH / 4);
note mult._RT_STAGES      = 3u;
note mult._RT_AREA        = BITWIDTH * BITWIDTH * 0.000297;
note mult._RT_COST        = BITWIDTH * BITWIDTH * 2.8;
note mult._RT_POWER       = BITWIDTH * BITWIDTH * 0.00005;
note mult._RT_OPERATIONS = {
  { "*", { {"result"}, {"a", "b"}, { } } }
};

note mult._SIR_PARAMETERS = {
  { "BITWIDTH", BITWIDTH_DFLT, { {2u, 64u} }, "Bitwidth", "bits" }
};
```

(b) Declaration

Listing 23: Example of 3-stage pipelined multiplier.

```
#include "mult.sh"      // macros
behavior stage1 (out signal unsigned bit[BITWIDTH-1:0] reg_a,
                 out signal unsigned bit[BITWIDTH-1:0] reg_b,
                 in  signal unsigned bit[BITWIDTH-1:0] a,
                 in  signal unsigned bit[BITWIDTH-1:0] b,
                 in  signal unsigned bit[0:0]          clk)    {
  void main(void) {
    while (1) {
      wait (clk rising);        reg_a = a; reg_b = b;
  } }
};
```

(c) First stage

```
#include "mult.sh"      // macros
behavior stage2 (out signal unsigned bit[BITWIDTH-1:0] reg_c,
                 in  signal unsigned bit[BITWIDTH-1:0] reg_a,
                 in  signal unsigned bit[BITWIDTH-1:0] reg_b,
                 in  signal unsigned bit[0:0]          clk)    {
  void main(void) {
    while (1) {
      wait (clk rising);        reg_c = reg_a * reg_b;
  } }
};
```

(d) Second stage

```
#include "mult.sh"      // macros
behavior stage3 (out signal unsigned bit[BITWIDTH-1:0] result,
                 in  signal unsigned bit[BITWIDTH-1:0] reg_c,
                 in  signal unsigned bit[0:0]          clk)    {
  void main(void) {
    while (1) {
      wait (clk rising);        result = reg_c;
  } }
};
```

(e) Third stage

Listing 23: Example of 3-stage pipelined multiplier (continued).

```
#include "mult_p.sh"     // declaration

import "mult_p1";        // stage 1
import "mult_p2";        // stage 2
import "mult_p3";        // stage 3

behavior mult (out signal unsigned bit[BITWIDTH-1:0] result,
               in  signal unsigned bit[BITWIDTH-1:0] a,
               in  signal unsigned bit[BITWIDTH-1:0] b,
               in  signal unsigned bit[0:0]          clk)
{
  note clk._RT_PIN_TYPE = 3u;

  signal unsigned bit[BITWIDTH-1:0] reg_a, reg_b, reg_c;

  stage1 U1 (reg_a, reg_b, a, b, clk);
  stage2 U2 (reg_c, reg_a, reg_b, clk);
  stage3 U3 (result, reg_c, clk);

  void main(void) {
    par {
        U1.main();
        U2.main();
        U3.main();
      }
  }
};
```

(f) Definition

Listing 23: Example of 3-stage pipelined multiplier (continued).

```
#define BITWIDTH_DFLT    32u

#ifndef BITWIDTH
#define BITWIDTH BITWIDTH_DFLT
#endif

behavior reg(out signal unsigned bit[BITWIDTH-1:0] outport,
             in  signal unsigned bit[BITWIDTH-1:0] inport,
             in  signal unsigned bit[0:0]          load,
             in  signal unsigned bit[0:0]          clk);

note reg._RT_LIBRARY       = "rtl";
note reg._RT_CATEGORY      = "Register";
note reg._RT_CLASS         = 2u;
note reg._RT_DATATYPE      = 0u;
note reg._RT_BITWIDTH      = BITWIDTH;
note reg._RT_INPUT_DELAY   = 0.4 + (BITWIDTH * 0.01);
note reg._RT_OUTPUT_DELAY = 0.2 + (BITWIDTH * 0.01);
note reg._RT_AREA          = 0.00006 * BITWIDTH;
note reg._RT_COST          = 1.0 * BITWIDTH;
note reg._RT_POWER         = 0.000001 * BITWIDTH;
note reg._RT_OPERATIONS    = {
  { "[]", { { {"outport"}, {},        { {"load", 1ub} } } } },
  { "=",  { { {},          {"inport"}, { } } } }
};

note reg._SIR_PARAMETERS = {
  { "BITWIDTH", BITWIDTH_DFLT, { {1u, 128u} }, "Bitwidth", "bits" }
};
```

(a) Declaration

Listing 24: Example of register.

```
#include "reg.sh"        // declaration

behavior reg(out signal unsigned bit[BITWIDTH-1:0] outport,
             in  signal unsigned bit[BITWIDTH-1:0] inport,
             in  signal unsigned bit[0:0]          load,
             in  signal unsigned bit[0:0]          clk)
{
  note load._RT_PIN_TYPE = 1u;
  note clk._RT_PIN_TYPE = 3u;

  void main(void)
  {
    unsigned bit[BITWIDTH-1:0] buf;

    while(1)
    {
      wait(clk rising);
      if (load == 1ub) {
        waitfor 0.4+(BITWIDTH*0.01);  // input delay
        buf = inport;
      }
      else {
        waitfor 0.2+(BITWIDTH*0.01);  // output delay
        outport = buf;
      }
    }
  }
};
```

(b) Definition

Listing 24: Example of register (continued).

```
#define BITWIDTH_DFLT    32u
#define SIZE_DFLT        16ul
#define ADDRWIDTH_DFLT   4u

#ifndef BITWIDTH
#define BITWIDTH BITWIDTH_DFLT
#endif

#ifndef SIZE
#define SIZE SIZE_DFLT
#endif

#ifndef ADDRWIDTH
#define ADDRWIDTH ADDRWIDTH_DFLT
#endif
```

(a) Macros

Listing 25: Example of register file.

```
#include "rf.sh"          // macros

behavior RF(out signal unsigned bit[BITWIDTH-1:0]  outA,
            out signal unsigned bit[BITWIDTH-1:0]  outB,
            out signal unsigned bit[BITWIDTH-1:0]  outC,
            in  signal unsigned bit[BITWIDTH-1:0]  inport,
            in  signal unsigned bit[ADDRWIDTH-1:0] raA,
            in  signal unsigned bit[ADDRWIDTH-1:0] raB,
            in  signal unsigned bit[ADDRWIDTH-1:0] wa,
            in  signal unsigned bit[0:0]           reA,
            in  signal unsigned bit[0:0]           reB,
            in  signal unsigned bit[0:0]           we,
            in  signal unsigned bit[0:0]           clk);

note RF._RT_LIBRARY      = "rtl";
note RF._RT_CATEGORY     = "Register File";
note RF._RT_CLASS        = 3u;
note RF._RT_DATATYPE     = 0u;
note RF._RT_SIZE         = SIZE;
note RF._RT_BITWIDTH     = BITWIDTH;
note RF._RT_INPUT_DELAY  = 0.5 + (SIZE * 0.001) + (BITWIDTH * 0.01);
note RF._RT_OUTPUT_DELAY = 1.0;
note RF._RT_STAGES       = 0u;
note RF._RT_CLOCK        = 2;
note RF._RT_AREA         = 0.00006 * (BITWIDTH/2) * (SIZE/4);
note RF._RT_COST         = 6.0 * (BITWIDTH/2) * (SIZE/4);
note RF._RT_POWER        = 0.00002 * (BITWIDTH/2) * (SIZE/4);
note RF._RT_OPERATIONS   = {
  { "[]", { { {"outA"}, {},        { {"reA", 1ub} } },
            { {"outB"}, {},        { {"reB", 1ub} } } } },
  { "=",  { { {},       {"inport"}, { {"we", 1ub} } } } }
};

note _SIR_PARAMETERS = {
  { "BITWIDTH", BITWIDTH_DFLT, { {1u, 128u} }, "Bitwidth", "bits" },
  { "SIZE", SIZE_DFLT, { {1ul, 128ul} }, "Size", "words" },
  { "ADDRWIDTH", ADDRWIDTH_DFLT, { {1u, 128u} }, "Address width", "bits" }
};
```

(b) Declaration

Listing 25: Example of register file (continued).

```
#include "RF.sh"         // declaration

behavior RF(out signal unsigned bit[BITWIDTH-1:0]  outA,
            out signal unsigned bit[BITWIDTH-1:0]  outB,
            out signal unsigned bit[BITWIDTH-1:0]  outC,
            in  signal unsigned bit[BITWIDTH-1:0]  inport,
            in  signal unsigned bit[ADDRWIDTH-1:0] raA,
            in  signal unsigned bit[ADDRWIDTH-1:0] raB,
            in  signal unsigned bit[ADDRWIDTH-1:0] wa,
            in  signal unsigned bit[0:0]           reA,
            in  signal unsigned bit[0:0]           reB,
            in  signal unsigned bit[0:0]           we,
            in  signal unsigned bit[0:0]           clk)
{
  note raA._RT_PIN_TYPE = 2u;
  note raB._RT_PIN_TYPE = 2u;
  note wa._RT_PIN_TYPE  = 2u;
  note reA._RT_PIN_TYPE = 1u;
  note reB._RT_PIN_TYPE = 1u;
  note we._RT_PIN_TYPE  = 1u;
  note clk._RT_PIN_TYPE = 3u;
  note reA._RT_ACTIVE_HIGH = true;
  note reB._RT_ACTIVE_HIGH = true;
  note we._RT_ACTIVE_HIGH  = true;
```

(c) Definition

Listing 25: Example of register file (continued).

```
  void main(void)
  {
    unsigned bit[BITWIDTH-1:0] buf[SIZE];

    while(1)
    {
      wait(clk rising);
      if(we ==1) {
        waitfor 0.5+(SIZE*0.001)+(BITWIDTH*0.01); // input delay
        buf[wa] = inport;
      }
      else if(reA == 1) {
        waitfor 1.0; // output delay
        outA = buf[raA];
      }
      else if(reB == 1) {
        waitfor 1.0; // output delay
        outB = buf[raB];
      }
    }
  }
};
```

(d) Definition (continued)

Listing 25: Example of register file (continued).

```
#define BITWIDTH_DFLT    32u

#ifndef BITWIDTH
#define BITWIDTH BITWIDTH_DFLT
#endif

behavior bus(inout signal unsigned bit[BITWIDTH-1:0] data)
{
  note bus.._RT_LIBRARY   = "rtl";
  note bus.._RT_CATEGORY = "Bus";
  note bus.._RT_CLASS     = 4u;
  note bus.._RT_DELAY     = 1.0;
  note bus.._RT_AREA      = BITWIDTH * 0.0000001;
  note bus.._RT_POWER     = BITWIDTH * 0.000001;
  note bus.._RT_COST      = BITWIDTH * 2.0;
  note bus.._RT_PROTOCOL = {"read", "write"};
  note bus.._RT_BITWIDTH  = BITWIDTH;

  note bus.._SIR_PARAMETERS = {
    {"BITWIDTH", BITWIDTH_DFLT, { {2u, 64u} }, "Bitwidth", "bits" }};

  unsigned bit[BITWIDTH-1:0] buf;

  void recv(unsigned bit[BITWIDTH-1:0] idata) {
    buf = idata;
  }
  unsigned bit[BITWIDTH-1:0] send(){
    return buf;
  }

  void main(void) {
    while(1) {
      waitfor 1.0;
      recv(data);
      data = send();
    }
  }
};
```

Listing 26: Example of bus.

A Manual Pages

This appendix contains the documentation in the form of manual pages for tools included in SCE that are used for database management.

A.1 `scc` - SpecC Compiler

NAME

scc – SpecC Compiler

SYNOPSIS

scc *–h*

scc *design* [*command*] [*options*]

DESCRIPTION

scc is the compiler for the SpecC language. The main purpose of **scc** is to compile a SpecC source program into an executable program for simulation. Furthermore, **scc** serves as a general tool to translate SpecC code from various input to various output formats which include SpecC source text, SpecC binary files in SpecC Internal Representation format, and other compiler intermediate files.

Using the first command syntax as shown in the synopsis above, a brief usage information and the compiler version are printed to standard output and the program exits. Using the second command syntax, the specified *design* is compiled. By default, **scc** reads a SpecC source file, performs preprocessing and builds the SpecC Internal Representation (SIR). Then, C++ code is generated, compiled and linked into an executable file to be used for simulation. However, the subtasks performed by **scc** are controlled by the given *command* so that, for example, only partial compilation is performed with the specified *design.*

On successful completion, the exit value 0 is returned. In case of errors during processing, an error code with a brief diagnostic message is written to standard error and the program execution is aborted with the exit value 10.

For preprocessing and C++ compilation, **scc** relies on the availability of an external C++ compiler which is used automatically in the background. By default, the GNU compiler **gcc/g++** is used.

ARGUMENTS

design specifies the name of the design; by default, this name is used as base name for the input file and all output files;

COMMAND

The *command* has the format - *suffix1* 2 *suffix2*, where *suffix1* and *suffix2* specify the format of the main input and output file, respectively. This command also implies the compilation steps being performed. By default, the command –sc2out is used which specifies reading a SpecC source file (e.g. design.sc) and generating an executable file (e.g. a.out) for simulation. All necessary intermediate files (e.g. design.cc, design.o) are generated automatically.

Legal command suffixes are:

sc	SpecC source file (default: *design.sc)*
si	preprocessed SpecC source file (default: *design.si)*
sir	binary SIR file in SpecC Internal Representation format (default: *design.sir)*
cc	C++ simulation source file (default: *design.cc)*

h C++ simulation header file (default: *design.h)*

cch both, C++ simulation source file and C++ header file (default: *design.cc* and *design.h)*

o linker object file (default: *design.o)*

out executable file for simulation (default: *design);* however, with the –ip option, a shared library will be produced (default: *libdesign.so)*

OPTIONS

–v | –vv | -vvv increase the verbosity level so that all tasks performed are logged to standard error (default: be silent); at level 1, informative messages for each task performed are displayed; at level 2, additionally input and output file names are listed; at level 3, very detailed information about each executed task is printed;

–w | –ww | -www increase the warning level so that warning messages are enabled (default: warnings are disabled); four levels are supported ranging from only important warnings (level 1) to pedantic warnings (level 4); for most cases, warning level 2 is recommended (–ww);

–g enable debugging of the generated simulation code (default: no debugging code); this option disables optimization;

–O enable optimization of the generated simulation code (default: no optimization); this option disables debugging;

–ip enable intellectual property (IP) mode; when generating a SIR binary or SpecC text file, only declarations of symbols marked public will be included (the public interface of an IP is created); when generating C++ code, non-public symbols will be output so that they will be invisible outside the file scope; when compiling or linking, the compiler and linker are instructed to create a shared library instead of an executable file (creation of an IP simulation library);

–n suppress creation of new log information when generating the output SIR file (default: update log information); see also section ANNOTATIONS below;

–sl suppress source line information (preprocessor directives) when generating SpecC or C++ source code (default: include source line directives);

–sn suppress all annotations when generating SpecC source code (default: include annotations);

–st tabulator stepping set the tabulator stepping for SpecC/C++ code generation; this setting is used for code indentation; a value of 0 will disable the indentation of the generated code (default: 4);

–sT system tabulator stepping set the system tabulator stepping (\t) for SpecC/C++ code generation; if set, tab characters will be used for indentation; if a value of 0 is specified, only spaces will be used for indentation (default: 8);

–sw line wrapping set the column for line wrapping; in code generation, any line longer than this value is subject to line wrapping; if a value of 0 is specified, no line wrapping will be performed (default: 70);

–i input file specify the name of the input file explicitly (default: *design.suffix1);* the name '-' can be used to specify reading from standard input;

–o output file specify the name of the final output file explicitly (default: *design.suffix2);* the name '-' can be used to specify writing to standard output;

–D	do not define any standard macros; by default, the macro __SPECC__ is defined automatically (it is set to 1); furthermore, implementation dependent macros may be defined; this option suppresses the definition of all these macros;
–Dmacrodef	define the preprocessor macro *macrodef* to be passed to the preprocessor;
–U	do not undefine any macros; by default, few macros are undefined automatically (in order to allow C/C++ standard header files to be used); this option is implementation dependent;
–Uundef	undefine the preprocessor macro *undef* which will be passed to the preprocessor as being undefined; the macro *undef* will be undefined after the definition of all command-line macros; this allows to selectively suppress macros from being defined in the preprocessing stage;
–I	clear the standard include path; by default, the standard include path consists of the directory $SPECC/inc; this option suppresses the default include path;
–Idir	append *dir* to the include path (extend the list of directories to be searched for including source files); include directories are searched in the order of their specification; unless suppressed by option –I, the standard include path is automatically appended to this list; by default, only the standard include directories are searched;
–L	clear the standard library path; by default, the standard library path consists of the directory $SPECC/lib; this option suppresses the default library path;
–Ldir	append *dir* to the library path (extend the list of directories to be searched for linker libraries); the library path is searched in the specified order; unless suppressed by option –L, the standard library path is automatically appended to this list; by default, only the standard library path is searched;
–l	when linking, do not use any standard libraries; the default libraries are displayed when calling the compiler with the –h option; the –l option suppresses linking against theses standard libraries;
–llib	pass *lib* as a library to the linker so that the executable is linked against *lib;* libraries are linked in the specified order; unless suppressed by option –l, the standard libraries are automatically appended to this list; by default, only standard libraries are used;
–P	reset the import path; clear the list of directories to be searched for importing files; by default, the current directory is searched first, followed by the standard import directory $SPECC/import; this option suppresses this standard import path;
–Pdir	append *dir* to the import path, extending the list of directories to be searched for importing files; import directories are searched in the order of their specification; unless suppressed by option –P, the standard search path is automatically appended to this list; by default, only the standard import path is searched;
–xpp preprocessor_call	redefine the command to be used for calling the C preprocessor (default: ”g++ -E -x c %p %i -o %o”); the preprocessor call must contain three markers %p, %i and %o, which indicate the options and file names used in the call; in the specified string, the %p marker will be replaced with the list of specified preprocessor options; the %i and %o markers will be replaced with the actual input and output filenames, respectively;
–xcc compiler_call	redefine the command to be used for calling the C/C++ compiler (default: ”g++ -c %c %i -o %o”); the compiler call must contain three markers %c, %i and %o, which indicate the options and file names used in the call; in the specified string, the %c marker will be replaced with the list of specified compiler options; the %i and %o markers will be replaced with the actual input and output filenames, respectively;

–xld linker_call redefine the command to be used for calling the linker (default: ”g++ %i -o %o %l”); the linker call must contain three markers %l, %i and %o, which indicate the options and file names used in the call; in the specified string, the %l marker will be replaced with the list of specified linker options; the %i and %o markers will be replaced with the actual input and output filenames, respectively;

–xp preprocessor_option pass an option directly to the C/C++ preprocessor (default: none);

–xc compiler_option pass an option directly to the C/C++ compiler (default: none);

–xl linker_option pass an option directly to the linker (default: none);

ENVIRONMENT

SPECC is used to determine the installation directory of the SpecC environment where SpecC standard include files (directory $SPECC/inc), SpecC standard import files (directory $SPECC/import), and SpecC system libraries (directory $SPECC/lib) are located.

SPECC_LICENSE_FILE determines the license file (path and file name) to be used by the SpecC environment; if undefined, the environment variable *SPECC* is used as the path to the license file called ”license.sce”; if neither *SPECC_LICENSE_FILE* nor *SPECC* exist, the file ”license.sce” is searched in the current directory;

ANNOTATIONS

The following SpecC annotations are recognized by the compiler:

_SCE_LOG contains the log information of the SIR file; this global annotation is created and maintained automatically by the SpecC compiler and the SpecC tool set and can be used to determine the origin and the operations performed on the design model; *_SCE_LOG* is a composite annotation consisting of a list of log entries, ordered by time of creation; each log entry consists of a time stamp, command line, source file, version info, and an optional comment;

_SCC_RESERVED_SIZE for external behaviors and channels (IP components), this indicates the size reserved in the C++ class for internal use; the annotation type is unsigned int; if found at class definitions, this annotation is checked automatically for reasonable values; for IP declarations, the annotation can be created automatically with the –ip option;

_SCC_PUBLIC for global symbols, this annotation indicates whether the symbol is public and will be visible in a shared library; the annotation type is bool; this annotation only is recognized with the –ip option;

VERSION

The SpecC compiler **scc** is version 2.2.b.

AUTHOR

Rainer Doemer <doemer@ics.uci.edu>

COPYRIGHT

(c) 1997-2003 CECS, University of California, Irvine

SEE ALSO

gcc(1), **g++**(1), **sir_delete**(l), **sir_depend**(l), **sir_import**(l), **sir_isolate**(l), **sir_list**(l), **sir_note**(l), **sir_rename**(l), **sir_strip**(l), **sir_tree**(l), **sir_wrap**(l)

BUGS, LIMITATIONS

Variables of enumerator type cannot be initialized at the time of their declaration. The SpecC compiler issues a (false) error message in this case. As a simple work-around, however, enumerator variables can be initialized by use of standard assignment statements at the beginning of their lifetimes.

A.2 `sir_gen` - SpecC Design Generator

NAME

sir_gen – part of the SpecC SIR tool set

SYNOPSIS

sir_gen [*options*] *design* [*parameter...*]

DESCRIPTION

sir_gen generates a parametrized design from a SpecC source template by creating a SIR file that contains the SpecC Internal Representation of the design.

sir_gen reads the SpecC source file for the given *design*, applies the *parameters* given on the command line, optionally performs name mangling, and writes the resulting SIR file for the *design* or for its mangled design name. Parameters are applied to the design by defining corresponding preprocessor macros when reading in the SpecC source template. Input and output file names can be overwritten using the –i and –o options.

On successful completion, the exit value 0 is returned. In case of errors, an error code with a diagnostic message is written to standard error and the program execution is aborted with the exit value 10. In this case, no output is produced.

ARGUMENTS

design specifies the main design name to generate; if no –i option is specificed, the file name for the input source template is deduced by appending the suffix '.sc' to this name; if no –o option is specified, the generated design will be written into a SIR file with the (possibly mangled, see –m option) name of the design plus the suffix '.sir';

parameter specified a parameter to be applied when generating the design; syntactically, a *parameter* is a key/value pair separated by an assignment character ('=', no whitespace); the *key* specifies the name of the parameter, while the *value* specifies the value to apply to the parameter; a *value* is given using standard SpecC syntax for constants; each key/value pair will create a corresponding preprocessor definition while reading of the *design* source file, i.e. the source code is templated using the capabilities of the preprocessor to modify code generation based on the parameter definitions;

OPTIONS

–h prints a short usage and version information and then quits;

–i input file specifies the name of the input source file template explicitly; the name '-' can be used to specify reading from standard input;

–m enables name mangling; the design name and the names of all global, non-imported definitions (behaviors, channels, interfaces, functions, and variables) are mangled by appending a unique suffix to the name; by default, the suffix will be generated from the given ordered *parameter* set; note that mangling of the design name will change the name of the output file accordingly;

–msuffix enables name mangling and specifies the *suffix* used to mangle names explicitly;

–n suppresses the creation of new log information when generating the output SIR file; by default, log information in the main design file is updated automatically (see also section ANNOTATIONS below);

–o output file specifies the name of the output design file explicitly; the name '-' can be used to specify writing to standard output;

–v | *–vv* | *–vvv* set the verbosity level so that actions performed are logged to standard error (default: be silent);

–w | ... | *–wwww* set the warning level so that warning messages are enabled (default: standard warnings are displayed); four levels are supported ranging from only important warnings (level 1) to pedantic warnings (level 4);

–D do not define any standard macros; by default, the macro __SPECC__ is defined automatically (it is set to 1); furthermore, implementation dependent macros may be defined; this option suppresses the definition of all these macros;

–Dmacrodef define the preprocessor macro *macrodef* to be passed to the preprocessor;

–U do not undefine any macros; by default, few macros are undefined automatically (in order to allow C/C++ standard header files to be used); this option is implementation dependent;

–Uundef undefine the preprocessor macro *undef* which will be passed to the preprocessor as being undefined; the macro *undef* will be undefined after the definition of all command-line macros; this allows to selectively suppress macros from being defined in the preprocessing stage;

–I clear the standard include path; by default, the standard include path consists of the directory $SPECC/inc; this option suppresses the default include path;

–Idir append *dir* to the include path (extend the list of directories to be searched for including source files); include directories are searched in the order of their specification; unless suppressed by option –I, the standard include path is automatically appended to this list; by default, only the standard include directories are searched;

–P resets the import path; the list of directories to be searched for import is cleared; by default, the current directory is searched first, followed by the standard import directory $SPECC/import; this option suppresses this standard import path;

–Pdir appends *dir* to the import path, extending the list of directories to be searched for importing files; import directories are searched in the order of their specification; unless suppressed by option –P, the standard search path is automatically appended to this list; by default, only the standard import path is searched;

ENVIRONMENT

SPECC is used to determine the installation directory of the SIR tool set where SpecC standard include files (directory $SPECC/inc), and SpecC standard import files (directory $SPECC/import) are located.

SPECC_LICENSE_FILE determines the license file (path and file name) to be used by the SIR tool set; if undefined, the environment variable *SPECC* is used as the path to the license file called "license.sce"; if neither *SPECC_LICENSE_FILE* nor *SPECC* exist, the file "license.sce" is searched in the current directory;

ANNOTATIONS

The following SpecC annotations are recognized by **sir_gen:**

_SCE_LOG contains the log information of the SIR file; this global annotation is created and maintained automatically by the SpecC compiler and the SpecC tool set and can be used to determine the origin and the operations performed on the design model; *_SCE_LOG* is a composite annotation consisting of a list of log entries, ordered by time of creation; each log entry consists of a time stamp, command line, source file, version info, and an optional comment;

_SIR_PARAMETERS specifies the set of valid parameters that can be applied to the global definition to which this annotation is attached; *_SIR_PARAMETERS* is a composite annotation consisting of a list of template parameters; each entry consists of the parameter name, the parameter default value, and the parameter range; the parameter range is given as a list that can contain both discrete values and pairs of min/max values;

MACROS

During processing of the SpecC source template input file, **sir_gen** will define preprocessor macros that correspond to the key/value pairs given as parameters on the command line.

In addition, the following preprocessor macros are set by **sir_gen** during reading of the source file:

_SIR_MANGLED_SUFFIX contains the name mangling suffix chosen via the –m option; if name mangling is disabled, the macro is not defined;

VERSION

The SpecC SIR tool set is version 2.2.b.

AUTHOR

Andreas Gerstlauer <gerstl@ics.uci.edu>

COPYRIGHT

(c) 1997-2003 CECS, University of California, Irvine

SEE ALSO

scc(l), **sir_delete**(l), **sir_depend**(l), **sir_import**(l), **sir_isolate**(l), **sir_list**(l), **sir_note**(l), **sir_rename**(l), **sir_stats**(l), **sir_strip**(l), **sir_tree**(l), **sir_wrap**(l)

BUGS, LIMITATIONS

It is generally not possible to perform name mangling in a fully canonical fashion (e.g. mangling of strings or mangling in the presence of multiple, disparate template definitions in the design). **sir_gen** only performs limited mangling by default. Therefore, the user should explicitly choose the mangling scheme.