SpecC System-level Static Scheduling

En-Shou Chang and Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

Technical Report #99–23

May 23,1999

Abstract

This report describes how to use SpecC System-Level Scheduling (SLS) tools, as well as definitions of SLS tools, restrictions of current release, and how to read the refined design generated by SLS tools.

For quick start, two simplified SLS tools provide basic scheduling functions. However, due to various situations in real designs, we suggest advance designers use combination of all SLS tools to utilize all features provided by SLS tools to obtain better results.

1 Introduction

The **SpecC System-Level Scheduler(SLS)** is comprised by a set of tools, each tool can perform a part of the scheduling job. These tools can be invoked in different combination and different order to meet a variety of scheduling goals.

Figure 1 depicts the basic scheduling flow. The SLS inputs a hierarchical SpecC description as shown in Figure 1(a). Before the scheduling, informations listed below have to be provided.

- Each leaf-behavior instance(defined in Section 4.2) is assigned to a specific type of PE, for example, Pentium-100Hz.
- Accurate or estimated execution time for each leaf-behavior instance is computed, for example, 312μS.
- Communication style and direction of each ports of each leaf-behavior instance are also determined.

The system-level static scheduling is performed in three major steps. The hierarchical description is first transformed into an **ETG(Extended Task Graph)** as shown in Figure 1(b). Hierarchy of the behaviors is turned into explicit precedences among the leaf-behavior instances. Implicit precedences caused by communications (the dashed arrows in Figure 1(a)) are also added into the ETG. The definition of ETG is given in Section 2.

Once the ETG is created, the SLS schedules all the nodes in the ETG according to the design goals and constraints given by the designer, then produces a schedule as shown in Figure 1(c). Different SLS scheduling tools can be involved here to obtain the best schedule for the given design goals and constraints.

Finally, according to the schedule obtained, the SLS refines the original SpecC description, creates necessary control signals, and modifies original variables and channel instances

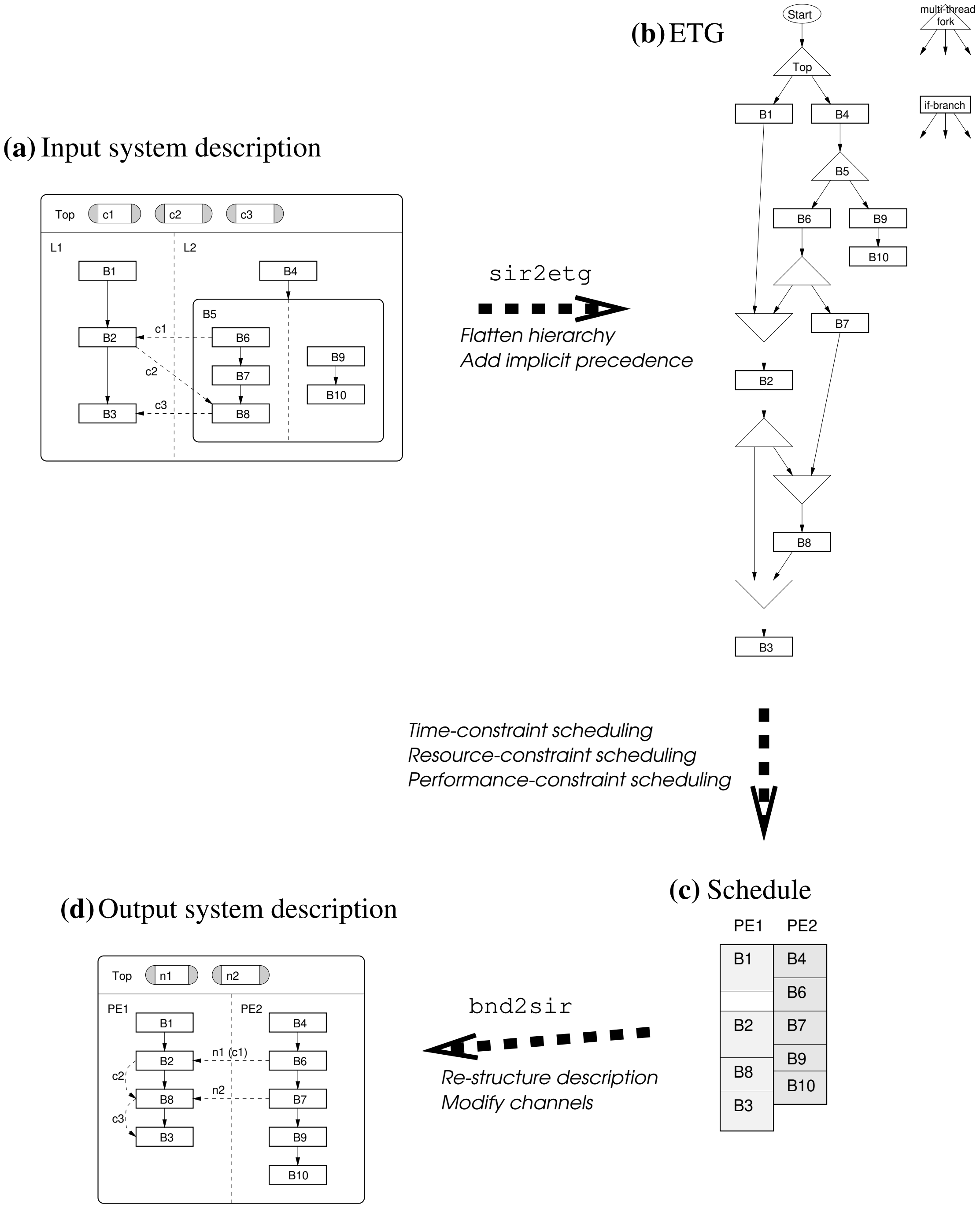


Figure 1: A basic SpecC system-level scheduling flow

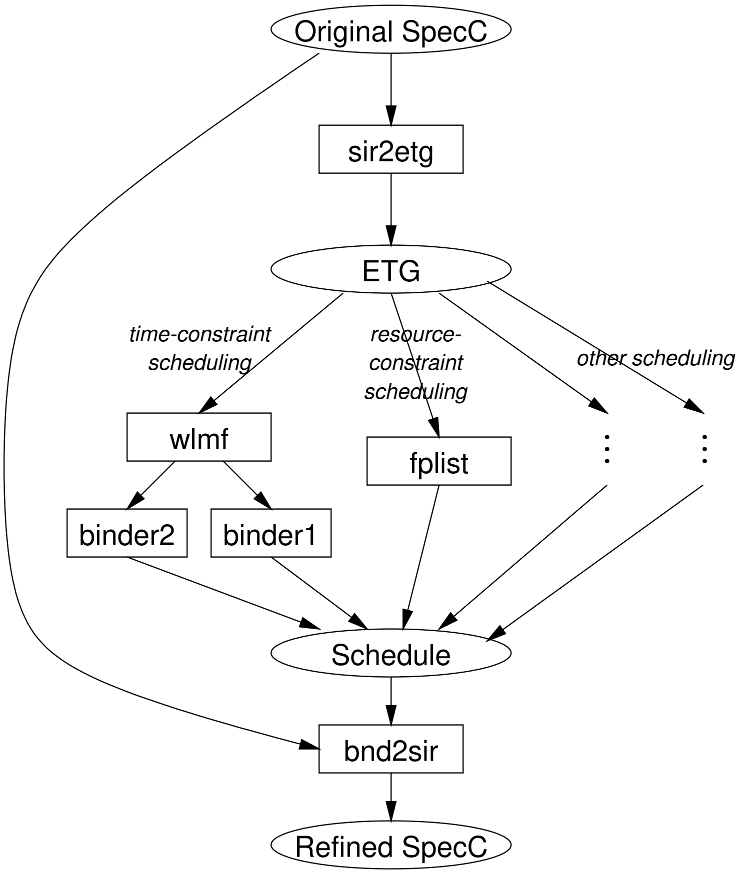


Figure 2: A basic SpecC system-level scheduling flow

to generate the refined SpecC description as shown in Figure 1(d). The output is then hand-over to the next synthesis stage.

Figure 2 depicts how the SLS tools are assembled to conquer varied scheduling goals. First, an SLS tool `sir2etg` inputs the original SpecC description and generates ETG. Then, different SLS tools which are implementation of a variety of system-level scheduling or binding algorithms are invoked to generate the best schedule. Finally, another SLS tool `bnd2sir` reads the original SpecC description as well as the schedule obtained and generates the refined SpecC description. Details of currently available SLS tools are given in Section 6

For quick start, two simplified tools, `sls_time` and `sls_resource`, are provided to conquer basic time-constraint and resource-constraint scheduling problems. However, to conquer varied scheduling problems, the designer can invoke combinations of all SLS tools to utilize all features provided by SLS.

This report is organized as following. In Section 2, we define ETG, which is a task graph specialized for system-level scheduling. In Section 3, we state some presumptions to settle some un-clarified issues in SpecC. In Section 4, we describe required **notations**[1] of SpecC descriptions which are going to to be fed into SLS tools and default conditions. In Section 5, we explain how to use two quick-start tools. In Section 6, we list all the SLS tools currently available and their features. In Section 7, we use an example to show how to use SLS tools. In Section 8, we list some design guidelines related to system-level scheduling which can lead to better designs. In Section 9, we state current restrictions on SLS tools. Finally, we discuss some technical details of SLS tools in Section 10 and Section 11..

2 ETG(Extended Task Graph)

The ETG is a task graph which is specialized for system-level scheduling. We define ETG as following.

Definition 1 An **ETG** is a graph $G = (V, E)$ where

- $V = V_T \cup V_F \cup V_J$;
- $E = E_{SEQ} \cup E_{SYNC}$;
- V_T is a set of **tasks** ;
- V_F is a set of **forks** ;

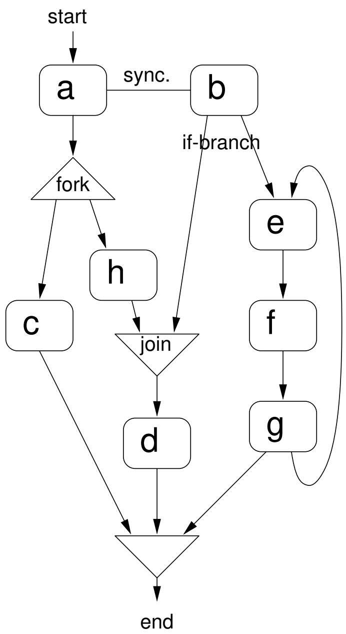


Figure 3: An example of ETG

- V_J is a set of **joins** ;
- E_{SEQ} is a set of **precedences**, which are directed arcs;
- E_{SYNC} is a set of **synchronizations**, which are undirected arcs.

Only one arc is allowed between any two nodes, either a precedence or a synchronization. When multiple precedences source a task, only one of the arcs can be true at run-time. When multiple precedences source a fork, all of them are true. Moreover, the sink of a join can not be true until all the sources of the join become true. Tasks connected by synchronizations have to be executed at the same time.

Figure 3 shows an example of ETG. Each box is a task. In applications, a **task** is a piece of work which occupies a specified type of hardware component for a certain amount of time. The up-right triangle is a fork and the inverted triangles are joins. Moreover, task a and b have to start at the same time, whereas task c and h don't have to.

3 Presumptions for SLS Tools

Some presumptions are made for un-clarified issues in SpecC documentations. We discuss these issues in the rest of this section.

3.1 Relaxed Specification Timing

The SpecC simulation engine assumes unlimited resources associated with ASAP (as soon as possible) scheduling for simulating. Each task will be executed as soon as it is ready. However, no scheduling is needed if the designer has already assumed ASAP scheduling. SLS tools assume relaxed timing in SpecC specification. Each task is not required to be executed as soon as it is ready. The scheduler can schedules each task to be executed at the best incidence time, according to the constraints and goals given by the designer.

SLS tools synthesize control signals in the refined design to synchronize all the tasks to be executed in correct order and satisfying the design constraint given by the designer. These control signals can also force each task to be executed at the scheduled incidence time by the simulation engine.

Under the relaxed timing assumption, there can be a time interval between a state and the next state in FSM. The value of a branch condition can change from time to time during this period. SLS tools assume all these values are valid. If only one of these values is valid, for example, the value at the time the current state complete, the designer should latch the valid valve in a variable.

4 Input Specification

The central idea of this section is to describe required notations of SpecC descriptions which are going to be fed into SLS tools. In addition, we discuss why these notations are vital for system-level scheduling and explain the default conditions for these annotations.

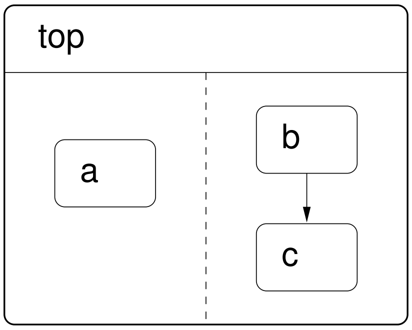
4.1 Usage of Behavior and Behavior Instances

The physical entities in a SpecC description are **behavior instances**[2], not **behaviors**[2]. A behavior description is the declaration of characteristics of a component. The component does not exist if no one invokes it, whereas the component may be duplicated if the behavior is invoked multiple times. For example, in Figure 4 there is only one mpeg behavior, whereas there are actually two mpeg chips needed in the system.

4.2 Leaf-behavior Instances

A **leaf-behavior** is defined as a behavior whose **BehaviorClass**[2] is `SIR_BHVR_LEAF`, `SIR_BHVR_EXTERN`, `SIR_BHVR_TRY` or `SIR_BHVR_OTHER`. The SLS tools treats leaf-behaviors as un-partitionable elements. Actually, a leaf-behavior may be partitionable, for example, a piece of software, or un-partitionable, for example, a hardware component. However, to well partition a leaf-behavior is beyond the scope of the scheduling job.

A notation `sls_bhvr_class = "leaf"` can be attached to any behavior instances to create super-nodes, which can be scheduled as leaf-behavior instances. For example, the static schedulers can not schedule an ETG which has unbounded loops. However, by the help of design experience and other knowledges, the designer can estimate the execution time of the loops or may know worse-case and average execution time of the loops. Thus, the designer can use notation `sls_bhvr_class = "leaf"` to force these loops to be processed as leaf-behavior instances, and put the above execution time as the `sls_dura` of these super-




```
behavior mpeg(inp_ch IN, out_ch OUT);
behavior top( ... )
{
  mpeg a(in1,out1),
       b(in2,out2),
       c(in3,out2);
  main() {
    par{
      a.main();
      {
        b.main();
        c.main();
      }
    }   // end par
  }   // end main
};   // end top
```

Figure 4: An example of behaviors and behavior instances

nodes. For details, please see `sir2etg` on-line manual page.

Though there are usually several behavior instances in a try-behavior, whose BehaviorClass is `SIR_BHVR_TRY`, all the resources which are needed to execute these behavior instances have to be reserved all the time while the try-behavior is executing, since any of these behavior instances can be executed at anytime. Thus, the scheduler can do nothing about these resources. As a result, a try-behavior instance is treated as single task by SLS tools. However, the behavior associated with each of these behavior instances can be scheduled as another top behavior.

4.3 Communication Entity and Task

Since SpecC is a hierarchical language, a **variable**[2] in the original description may represent multiple **real** variables. Figure 5 shows an example. There is only one variable `x` in the description, whereas three real variables are needed for `x` in the system desired. Similarly, the same situation applies on **channel instances**[2] and behavior instances. We define a **communication entity** as a real variable or real channel instance. Moreover, **port variables**[1] of the top behavior are also considered as communication entities, since they are accessed similar to variables and channel instances in the top behavior.

The SLS tool `sir2etg` flattens the original description into an ETG. In the ETG, each task is usually a real behavior instance as explained above, or can be a communication entity which exclusively occupies the hardware component it is bound to during its lifetime.

4.4 Behavior and Channel Instances

For each leaf-behavior instance, two notations are required for the scheduling stage, namely `sls_type` and `sls_dura`. `sls_type` specifies the assigned PE type for the behavior instance, and `sls_dura` specifies the execution time of the behavior instance. The execution time can be **worst case execution time(WCET)**, average case execution time, or any metrics

```
behavior BX()
{
  int x;
  ....
};

behavior Main()
{
  BX B1(),B2(),B3();
  main() {
    par {
      B1.main();
      B2.main();
      B3.main();
    }   // end par
  }   // end main
};   // end Main
```

Figure 5: An example of a SpecC variable which represents multiple real variables in the system desired

```
behavior L1_d( ... )
{
  io_channel c1;
    note c1.sls_type = "PCI";      // Channel type
  B1_d B1();
    note B1.sls_type = "P5-100";   // PE type
    note B1.sls_dura = 312;        // task duration
  ...
```

Figure 6: An example of behavior notations

related to execution time. It depends on which execution time the designer wants to optimize. Default `sls_type` is `"default"`. Default `sls_dura` is `0`. Figure 6 shows an example of notations for a leaf-behavior instance.

Channel instances can also be annotated `sls_type`, and the default is `"chnl_default"`. However, `sls_dura` makes no use for channel instances by current SLS tools.

4.5 Ports

For each **port**[1] of a leaf-behavior, a notation `sls_dir` is required for the scheduling stage. Based on scheduling view, communications can be categorized into three primary styles as following.

Synchronized communication The sender and the receiver have to be active concurrently. For example, two components communicate via hand-shaking.

Buffered communication The sender doesn't have to wait for the receiver. However, all the messages have to be received by the receiver. For example, two components communicate through FIFO. As a consequence, the receiver needs to be scheduled after the sender is finished, if no additional conditions are specified.

```
behavior L1_d( chan_in c1 , chan_update c2 )
{
  note c1.sls_dir = 'i';   // buffered input
  note c2.sls_dir = 'u';   // update
  ...
```

Figure 7: An example of external communication notations

Update communication The message can be ignored if no one is waiting for the message, whereas the receiver always uses the latest update and doesn't have to wait for a sender sending a message.

The communication style of an external(outside-behavior) data access can be annotated by `sls_dir` as shown in Figure 7. Four options are available, namely, `'s'`(synchronized), `'i'`(buffered input), `'o'`(buffered output), and `'u'`(update). Default `sls_dir` is `'u'`. Moreover, since global variable accesses which are not through any ports can't be annotated in current SpecC version[2], their communication style can only be `'u'`.

An external data access in SpecC can be any of three communication styles, depends on how the designer manages it. Figure 8 and Figure 9 show an example. The communication description in Figure 8 can be a buffered communication or an update communication. In case behavior `A_Spec` is declared as in Figure 9(a). The behavior instance `D` is guaranteed to receive *new_data* and can not be executed until `A` is finished. The communication in Figure 8 is a buffered communication. On the other hand, the same communication description is an update communication if behavior `A_Spec` is declared as in Figure 9(b). The behavior instance `D` can be executed even `A` is not finished. `D` uses *new_data* if it starts after `A` is finished; whereas `D` uses *old_data* instead if it starts before `A` is finished.

It is not feasible for the synthesis tools to tell which communication style the designer is using in each communication description. Thus, a notation to identify the communication

```
behavior R_Spec(bool s, event e, d_type data)
{
   A_Spec A(s,data);
   B_Spec B();
   main()
   {
      A.main();
      notify e;
      B.main();
   }
};    // end R_Spec

behavior L_Spec(bool s, event e, d_type data)
{
   C_Spec C();
   D_Spec D(data);
   main()
   {
      C.main();
      if(s) wait e;
      D.main();
   }
};    // end L_Spec

behavior top()
{
   bool s;
   d_type data;
   event e;
   R_Spec R(s,e,data);
   L_Spec L(s,e,data);
   main()
   {
      data = old_data ;
      ...
      par {
         R.main();
         L.main();
      }
   }
};    // end top
```

Figure 8: An example of un-determined communication style

(a)

```
behavior A_Spec(bool s, d_type data)
{
   main()
   {
      s = TRUE;
      ...
      data = new_data ;
      s = FALSE;
   }
} ;    // end A_Spec
```

(b)

```
behavior A_Spec(bool s, d_type data)
{
   main()
   {
      s = FALSE;
      ...
      data = new_data ;
   }
};    // end A_Spec
```

Figure 9: Design two different communication styles using the same communication description: (a) Buffered communication; (b) Update communication

style of each external data access is necessary.

4.6 Improper annotation

SLS tools can *always* generate correct refined descriptions corresponding to input SpecC descriptions, with or without notations described in this section. The notations play the roll of providing vital information for improving the design during this refinement(system-level scheduling). The quality of this refinement is closely depend on the quality of these notations.

In case the PE type is default, the schedulers assign the behavior instance to be executed by generic PE; whereas in case the duration is default, the schedulers automatically allocate one PE for the behavior instance itself and synthesize synchronization signals for the behavior instance to work correctly with the rest of the system. Therefore, in both default cases the refined description can still function correctly.

In some cases the original description may run well but the refined description generates incorrect output, especially when improper or incorrect values are annotated for SLS tools. This situation implies *the original description is not completely correct and will sometime generate good output but sometimes not.* Basically, what the system-level scheduler does is to add restrictions on the task graph to force task execution through better sequence. Since the bad notations lead SLS tools into worse cases, the refined description will tend to go through a worse execution sequence. Therefore, it is more often for the refined description to generate the bad output.

5 Quick Start

Two simplified SLS tools, `sls_time` and `sls_resource`, provide basic features to conquer time-constraint and resource-constraint scheduling problems respectly. We explain how to use them in the rest of this section.

5.1 Time-Constraint Scheduling

NAME

sls_time – SpecC System-Level(SLS) Time-Constraint Scheduling

SYNOPSIS

sls_time *input_file output_file top_bhvr* [*–p PE_cost_file*] [*–m mobility*] [*–b time-constraint*] [*–f*]

DESCRIPTION

sls_time reads an SIR(SpecC Internal Representation)[2] file *input_file* and outputs a refined description in SIR file *ouput_file.*

The *PE_cost_file* is required by **sls_time**. Use option *–p* to input the PE cost file name. The default PE cost file name is wlmf.pel .

In *output_file* , a refined top behavior _sls_#_top_behavior is generated by scheduling the original top behavior *top_bhvr* with time-constraint *time-constraint.* The # is a sequence number started at 1 and later generated top behaviors are associated with larger #. **sls_time** outputs the name of the refined top behavior to standard I/O.

The original top behavior is not replaced by the refined top behavior. A related tool **change_bhvr**, which is described in Section 5.3, can be used to change the behavior type of selected top behavior instance(s) to the refined top behavior.

ARGUMENTS

input_file input SIR file

output_file refined SIR file

top_bhvr top behavior name (default: Main)

OPTIONS

–*p PE_cost_file* PE cost file name (default: wlmf.pel)

–*m mobility* use mobility as the time-constraint

–*b time-constraint* use time-constraint

Option –*b* will overwrite –*m* (default: -*m 0*)

–*f* evaluate distribution boundaries only, result quicker but a little worse schedule

5.2 Resource-Constraint Scheduling

NAME

sls_resource – SpecC System-Level(SLS) Resource-Constraint Scheduling

SYNOPSIS

sls_resource *input_file output_file top_bhvr resource_priority_file*

DESCRIPTION

sls_resource reads an SIR(SpecC Internal Representation) file *input_file* and a file *resource_priority_file* which contains informations about resources assigned and priorities of tasks, then outputs a refined description in SIR file *output_file.*

In *output_file* , a refined top behavior _sls_#_top_behavior is generated by scheduling the original top behavior *top_bhvr* with the resources and priorities assigned. The # is a sequence number started at 1 and later generated top behaviors are associated with larger #. **sls_resource** outputs the name of the refined top behavior to standard I/O.

The original top behavior is not replaced by the refined top behavior. A related tool **change_bhvr**, which is described in Section 5.3, can be used to change the behavior type of selected top behavior instance(s) to the refined top behavior.

The *resource_priority_file* is a text file which contains a sequence of numbers. The first part of the sequence are informations regarding resources allocated. As shown below, the first number is the total number of types of resources. Following the first number are the numbers of resources allocated of each resource type.

$$\#types \;\; \#resource_0 \;\; \#resource_1 \;\; \#resource_2 \;\; \cdots \;\; \#resource_n$$

The resource type numbers are assigned by an SLS tool **sir2etg**. The type numbers can be found in the information displayed by **sir2etg** running in non-quiet mode (without option $-q$). Please see Section 10 for details.

The second part of the sequence are informations regarding scheduling priorities of nodes. As shown below, the first number is the total number of nodes which are assigned priorities. Following are nodes ranked by priority.

$$\#nodes\_with\_priority \;\; highest\_node \;\; second\_node \;\; third\_node \;\; \cdots$$

Nodes which are not appeared have the lowest priority. The node numbers are also assigned by **sir2etg**. The node numbers can be found in the information displayed by **sir2etg** running in non-quiet mode. Please also see Section 10 for details.

In void of the second part of the sequence, all nodes have the same priority.

ARGUMENTS

input_file input SIR file

output_file refined SIR file

top_bhvr top behavior name (default: Main)

resource_priority_file a file which contains the information about available resources and task priorities.

5.3 Change Type of Behavior Instances

NAME

change_bhvr – change behavior type of a behavior instance

SYNOPSIS

change_bhvr *original_SIR replacer* [*replacee behavior*] [–o *output_SIR*]

DESCRIPTION

change_bhvr reads an SIR file *original_SIR,* changes the behavior type of the selected behavior instance *replacee* located in behavior *behavior* to *replacer,* then outputs the SIR to file *output_SIR.* When no output file is specified, the output will over-write *original_SIR*

In case no *replacee* and *behavior* are specified, `behavior Main` will be replaced by the *replacer.*

ARGUMENTS

replacer new behavior name

replacee the behavior instance which will be changed

behavior the behavior which *replacee* is located in **OPTIONS**

–o output_SIR output SIR file name.

6 Currently Available Tools

Currently available SLS tools are described in this section. Please also see on-line manual pages for details.

6.1 Transformation from and to SIR

6.1.1 `sir2etg`

`sir2etg` reads an **SIR(SpecC Internal Representation)**[2] file and outputs the ETG of the selected **top behavior** to a binary file. Please see header file `etgdef.h` for details of the binary ETG file.

Since there can be multiple real entities for each behavior instance, variable, or channel instance, as discussed in Section 4.3, `sir2etg` assigns each real behavior entity a unique task identification number. Each communication entity (real channel or variable entity) is also assigned a unique communication entity identification number. In non-quiet(default) mode, `sir2etg` displays informations about traversing the SIR, communication entities, and the ETG. The task and communication entity identification numbers are included in the display. We explain these informations in Section 10.

6.1.2 bnd2sir

`bnd2sir` reads an SIR file and task-to-PE scheduling and binding information, then outputs the refined SpecC description in SIR. Please see `$SPECC/src/sls/api2/read_bnd.cc` for details about the binary file which contains the scheduling and binding information.

In non-quiet(default) mode, `bnd2sir` displays information about refinement as well as informations similar to what are displayed by `sir2etg`. Moreover, the refined SpecC description is explained in Section 11

The refined top behavior's name is `_sls_#_top_behavior`, where `#` is a sequence number started at 1 and later generated top behaviors are associated with larger `#`. `bnd2sir` outputs the newest top behavior name to the standard I/O. The original top behavior is not replaced by the refined top behavior. Some related tools e.g. `change_bhvr` can be used to change the behavior type of selected top behavior instance(s) to the refined top behavior.

6.2 Scheduling and Binding

6.2.1 wlmf

`wlmf` is a time-constraint scheduling program. It reads an ETG file, and outputs a schedule of the ETG to a binary file. Please see `$SPECC/src/sls/f1/read_sch.cc` for details about the binary file.

A PE cost file is required by `wlmf`. Use option `-p` to input PE cost file name. The default PE cost file name is `wlmf.pel`. Using option `-f` can obtain quicker but worse schedule.

6.2.2 `fplist`

`fplist` is a resource-constraint scheduling program. It implements a fix-priority LIST scheduling algorithm. It reads an ETG file and an ASCII file which contains the information about available resources and task priorities, then output a resource-constraint schedule to a binary file, in which each task is not only scheduled to a specific time but also bound to a specific PE.

The `resource_priority_file` is a text file which contains a sequence of numbers. The first part of the sequence are informations regarding resources allocated. As shown below, the first number is the total number of types of resources. Following the first number are the numbers of resources allocated of each resource type.

$$\#types \quad \#resource_0 \quad \#resource_1 \quad \#resource_2 \quad \cdots \quad \#resource_n$$

The resource type numbers are assigned by an SLS tool `sir2etg`. The type numbers can be found in the information displayed by `sir2etg` running in non-quiet mode (without option `-q`). Please see Section 10 for details.

The second part of the sequence are informations regarding scheduling priorities of nodes. As shown below, the first number is the total number of nodes which are assigned priorities. Following are nodes ranked by priority.

$$\#nodes\_with\_priority \quad highest\_node \quad second\_node \quad third\_node \quad \cdots$$

Nodes which are not appeared have the lowest priority. The node numbers are also assigned by `sir2etg`. The node numbers can be found in the information displayed by `sir2etg` running in non-quiet mode. Please also see Section 10 for details. In void of the second part of the sequence, all nodes have the same priority.

Please see `$SPECC/src/sls/api2/read_bnd.cc` for details about the binary file which contains the scheduling and binding information.

6.2.3 Limitation on Static Scheduling

As the nature of static scheduling, `wlmf` and `fplist` only can schedule acyclic ETGs. To schedule a ETG which contains loops, each loop needs to be either unrolled or annotated as a super-node as described in Section 4.2. However, the loop-body can be statically scheduled.

6.3 Other Tools

6.3.1 uid

`uid` assigns universal ID numbers for the behavior or channel instances. Both input and output are SIR files.

6.3.2 sch2pri

`sch2pri` converts output of `wlmf` to the resource and priority lists for input of `fplist` so `fplist` can be used as a binder of `wlmf`. `sch2pri` outputs to the standard I/O, which can be re-directed to a file for input of fplist.

6.3.3 change_bhvr

`change_bhvr` reads an SIR file, changes the behavior type of a selected behavior instance, then outputs the updated SIR. When no output file name is specified, the output will over-write the original SIR file.

7 A Synthesis Example

In this section, we use an example to show how the SLS tools are used to do general system-level scheduling. Figure 10 shows the schematic diagram of the input SpecC description. The input description can be found in `$SPECC/examples/sls/before.sc` . In Figure 10, each box is a behavior instance, and `c1, c2,` and `c3` are channel instances which all have

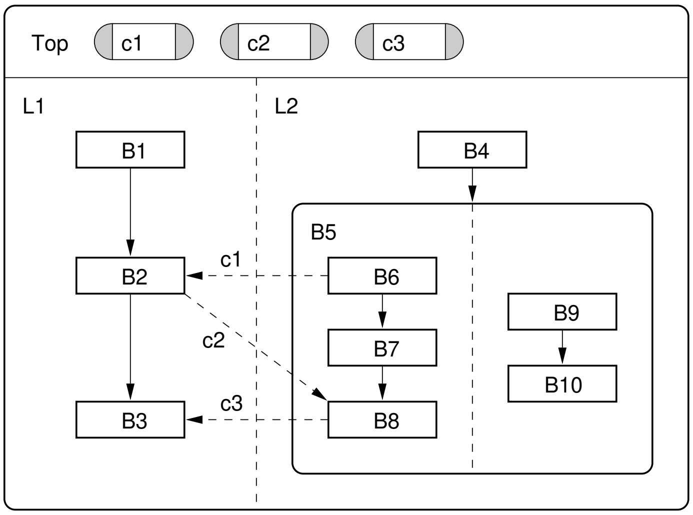


Figure 10: The schematic diagram of `before.sc`

the buffered communication style, as defined in Section 4.5. That is to say behavior instance `B2` has to wait for `B6` finished, and so do `B8` and `B3` to wait for `B2` and `B8` respectly. In the rest of this section, we show two basic SLS synthesis scripts, namely, a resource-constraint scheduling scripts and a time-constraint scheduling scripts.

7.1 Resource-Constraint Scheduling

Figure 11 shows a standard synthesis script for resource-constraint scheduling. SLS tools input SIR files which can be compiled by the scc compiler as `before` shown in line 1. The input SIR file can also be generated by other SpecC tools.

The first step of system-level scheduling is creating the task graph (ETG), as `before.etg` shown in line 2. Then, in line 3, we create an example resource-priority file `before.pri` for the LIST scheduler `fplist`. The `before.pri` describes 2 type number 4 PEs are allocated and task priority sequence are "22, 2, 3, ...". Task 22 has the highest priority, task 2 has the second highest priority, and so on. The task numbers and PE type numbers are assigned

```
1  scc before -sc2sir -vv -w
2  sir2etg before.sir -t Main -o before.etg
3  echo 5 0 0 0 0 2  22 2 3 4 5 7  8 11 14 15 18  19 20 21 0 1  \
   13 6 9 10 12  16 17 > before.pri
4  fplist before.etg before.pri -o before.bnd
5  bnd2sir before.sir before.bnd -t Main -o after.sir \
   -l $SPECC/sirlib/_sls_bnd2sirlib.sir
6  change_bhvr after.sir _sls_1_top_behavior
7  scc after -sir2sc -vvv -www
```

Figure 11: A standard synthesis script for resource-constraint scheduling

by `sir2etg` and can be found in the display of `sir2etg`. Please see Section 10 for details. Moreover, the resource-priority file can be input by the designer as shown here, as well as automatically generated by estimation or exploration tools.

Once both input data are prepared, the LIST scheduler `fplist` inputs `before.etg` and `before.pri`, and outputs the scheduling and binding result to `before.bnd` as shown in line 4. Then, in line 5, `bnd2sir` inputs the results and the original description `before.sir`, then generates the refined description and outputs it to the SIR file `after.sir`. The original top behavior `Main` is not overwritten by the newly synthesized top behavior. The new top behavior's name is output to standard I/O. We assume the new top behavior's name is `_sls_1_top_behavior` here. Figure 12 and Figure 13 show the schedule-and-binding diagram and the schematic diagram of the refined top behavior respectly. Again, the schedule-and-binding file can be generated by `fplist` as well as any available tools.

Since there could be several instances of the original top behavior, we give the designer

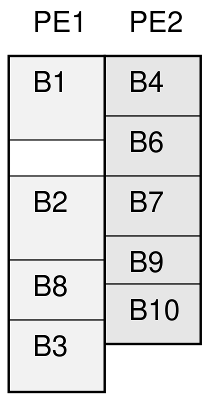


Figure 12: Input schedule-and-binding for `bnd2sir`

the power to pick which top behavior instances to be replaced by the refined top behavior. As shown in line 6, `change_bhvr` replaces the original top behavior `Main` by the new top behavior `_sls_1_top_behavior`. Finally, the refined description `after.sir` can be output to next refinement tools or deparsed by the scc compiler for human reading, as shown in line 7.

7.2 Time-Constraint Scheduling

Figure 14 shows a standard synthesis script for time-constraint scheduling. Similar to the synthesis script in Section 7.1, line 1 and line 2 prepare the ETG `before.etg` for the scheduler. Then, in line 3, the time-constraint scheduler `wlmf` schedules `before.etg` with given mobility `0.1` and outputs the schedule to the file `before.sch`.

`wlmf` is based on a scheduling algorithm similar to Force-Directed Scheduling[3], where each task is assigned a start time but not bound to a specific PE. As a consequence, we need to hand over the schedule to a binder to bind each task to a PE. In this example, we use `fplist` as the binder. In line 4, a bridge tool `sch2pri` transforms `before.sch` into the input data format of `fplist`. Then, in line 5 `fplist` inputs all the necessary information

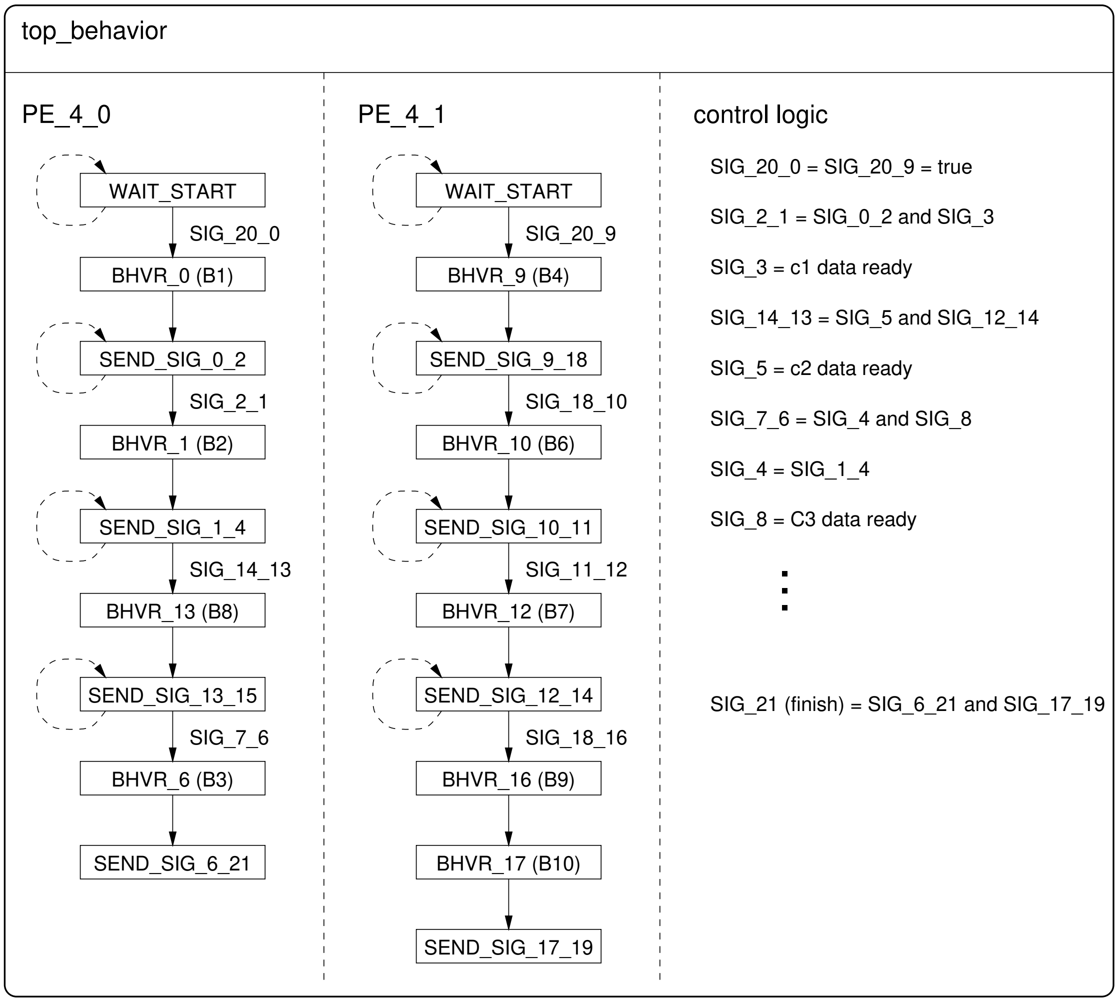


Figure 13: The schematic diagram of the refined top behavior

```
1  scc before -sc2sir -vv -w
2  sir2etg before.sir -t Main -o before.etg
3  wlmf before.etg -m 0.1 -o before.sch
4  sch2pri before.sch > before.pri
5  fplist before.etg before.pri -o before.bnd
6  bnd2sir before.sir before.bnd -t Main -o after.sir \
   -l $SPECC/sirlib/_sls_bnd2sirlib.sir
7  change_bhvr after.sir _sls_1_top_behavior
8  scc after -sir2sc -vvv -www
```

Figure 14: A standard synthesis script for time-constraint scheduling

and generates the schedule-and-binding result `before.bnd`.

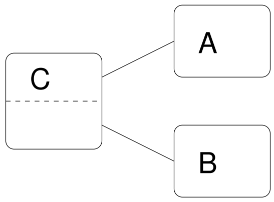
Finally, similar to the synthesis script in Section 7.1, `bnd2sir` synthesizes the refined description in line 6, and behavior `Main` is replaced by the synthesized top behavior `_sls_1_top_behavior` in line 7.

8 Design Guidelines

In this section, we list several guidelines which can help improve the design quality.

8.1 Partition Leaf-Behaviors

Properly partition leaf-behaviors can effectively improve scheduling quality. Figure 15 shows an example. The leaf-behavior instance `C` has synchronized communications with both leaf-behavior instances `A` and `B`. Without partitioning the behavior associated with `C`, both `A` and `B` need to be active with `C`, thus three processors are required as shown in Figure 15(a). Once we can partition the behavior associated with `C` into two new leaf-behaviors with



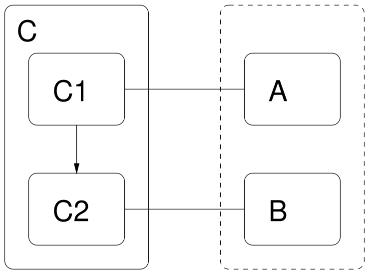


Figure 15: An example of refining leaf-behaviors: (a) Before partitioning the leaf-behavior associated with `C`, we need three processors to execute these tasks; (b) After partitioning the leaf-behavior associated with `C` into two new leaf-behaviors, it needs only two processors.

instantiations `C1` and `C2` respectly, `B` can be inactive until `C1` and `A` are finished, thus only two processors are needed as in Figure 15(b).

9 Current Limitations of SLS Tools

At the time we started developing SLS tools, some features in current SpecC version were not there. Thus, these features are not dealt in current SLS tools. In this section, we list these features which are not supported by current SLS tools.

9.1 Bit-vector in port-map[2]

Bit-vectors in port-map are not supported by current SLS tools. Figure 16 shows an example. SLS tools will display error messages and bail out when they encounter bit-vectors in port-map.

9.2 Channel with Ports

The idea of a channel which has ports is not fully understood at this time. Currently, SLS tools assume all accesses of ports of channels are update communications. Moreover, SLS

```
behavior xyz( ... );

behavior bit_vector_in_port_map( ... )
{
  xyz xyz_inst( var1[1:14]@var2[5:8] );

  main(){
    ...
  }
};
```

Figure 16: An example of bit-vectors in port-map

tools can only deal with channel instances in the top behavior and global channel instances to have ports. SLS tools will display error messages and bail out when they encounter channel instances which are not global or in the top behavior and have ports.

9.3 Behavior with Multiple Entries

A behavior has multiple member access entries is not supported by current SLS tools. Figure 17 shows an example of a behavior which has multiple member access entries. SLS tools will display error messages and bail out when they encounter a behavior member access other than `main()`. However, if all the non-leaf behaviors involved in the scheduling contain only behavior member accesses through `main()`, SLS tools will work correctly.

10 Read Display of SLS Tools

In non-quiet(default) mode, SLS tools display useful informations for the designer to verify the refinement and understand the refined description. In this section, we explain the display of `sir2etg`. Other SLS tools display similar informations upon the need of understanding their refinement.

```
interface abc
{
  void xyz( int x );
  main();
};

behavior multiple_entry( ... ) implement abc
{
  void xyz( int x ) {
    ...
  }

  main() {
    ...
  }
};

behavior Main()
{
  multiple_entry mei( ... );
  main() {
    mei.xyz( y );
    mei.main();
  }
};
```

Figure 17: An example of a behavior which has multiple member access entries

The issues discussed in Section 10 and Section 11 are closely related to SIR(SpecC Internal Representation)[2]. Please see[2] for the definitions of the SIR terminologies.

10.1 Traverse Information

Owing to SpecC Synthesis System is still under construction, SLS tools display trace while traversing through SIR. The trace is very helpful for verifying and debugging programs and designs. We use a segment of the trace in Section 10.1.1 along with a portion of related original SpecC description in Section 10.1.2 to illustrate the traverse information.

Each line of the trace in Section 10.1.1 contains the informations associated with an SIR_Definition which makes sense to scheduling. Based on scheduler's view, these SIR_Definitions can be categoried into to four groups.

1. **variables or channel instances**
2. **leaf-behavior instances**
3. **non-leaf-behavior instances**
4. **port-maps**

There can be multiple tasks associated with the one leaf-behavior instance and multiple communication entities associated with one variable or a channel instance, as explained in Section 4.3.

The informations associated with each of the SIR_Definitions include its name, type, task-ID(tid), mapping, and scheduling related notations. Most of these informations are names of behaviors, behavior instances, variables, types, channels, channel instances, interfaces, etc., which can be found easily in the original description, as shown in Section 10.1.2. We only explain those which are generated by SLS tools below.

A line in the trace leads with a number, for example, line 24 of the trace contains the informations associated with a behavior instance `L1` in line 235 in the original description. The leading number is enumerated BehaviorClass as in SIR. The indention of the lines of the trace indicates the ancestor-descendent relation. For example, line 24 is associated with behavior instance `L1` which has 3 ports, `c1`, `c2`, and `c3` associated with line 25,26, and 27 respectly, and 3 child behavior instances, `B1`, `B2`, and `B3` associated with line 28, 32, and 38.

A line in the trace leads with one name followed by a colon and a number, for example, line 35 contains the information associated with the second port-map of behavior instance `B2` (in line 32 of the trace and line 148 of the original description). The name is the name of the corresponding SIR_PortVar (`ci` in line 95 of the original description) and the number is the ID number of the communication entity which the port is mapped to. We explain details of the communication entity in Section 10.2.

The **task-ID(tid)** is generated by `sir2etg` and is unique when we flatten the hierarchical SpecC description. We also can use the tid to find the ancestors of the task.

10.1.1 A segment of the trace of traversing SIR

```
18  nb8 *  tid=nb8
19  nb9 *  tid=nb9
20  3 Main Main
21    c1 sig_ch  tid=Main.c1
22    c2 sig_ch  sls_type="pci":3  tid=Main.c2
23    c3 sig_ch  tid=Main.c3
24    2 L1 L1_d
25      c1:16
26      c2:17
27      c3:18
28      1 B1 NOIO
29        delay:0
30        mark:6
```

```
   sls_type="P5-100":4  sls_dura=1.0000e-01:0.1  tid=Main.L1.B1
  1 B2 IO
   delay:4
   mark:8
   ci:16  sls_dir='i':i
   co:17  sls_dir='o':o
   sls_type="P5-100":4  sls_dura=7.0000e-02:0.07  tid=Main.L1.B2
  1 B3 I
   delay:2
   mark:9
   ci:18  sls_dir='i':i
   sls_type="P5-100":4  sls_dura=5.0000e-02:0.05  tid=Main.L1.B3
 2 L2 L2_d
  c1:16
  c2:17
```

10.1.2 A portion of a SpecC design description

```
    *nb8 = "B8" ,
    *nb9 = "B9" ,
...
behavior NOIO( int delay , char *mark )
{
 void main()
  {
   printf( "%6lld s-%-7.7s\n" , now() , mark );
   waitfor( delay );
   printf( "%6lld e-%-7.7s\n" , now() , mark );
  }
};
...
behavior IO( int delay , char *mark , sig_in ci , sig_out co )
{
 note ci.sls_dir = 'i' ;
 note co.sls_dir = 'o' ;

```

```
  void main()
    {
      ci.read();
      printf( "%6lld s-%-7.7s\n" , now() , mark );
      waitfor( delay );
      printf( "%6lld e-%-7.7s\n" , now() , mark );
      co.write();
    }
};
...
behavior L1_d( sig_in c1 , sig_out c2 , sig_in c3 )
{
  NOIO B1(d100,nb1);
  note B1.sls_dura = 0.1;
  note B1.sls_type = "P5-100";
  IO   B2(d70,nb2,c1,c2);
  note B2.sls_dura = 0.07;
  note B2.sls_type = "P5-100";
  I    B3(d50,nb3,c3);
  note B3.sls_dura = 0.05;
  note B3.sls_type = "P5-100";

  void main()
    {
      B1.main();
      B2.main();
      B3.main();
    }
};
...
behavior Main()
{
  sig_ch c1(),c2(),c3();
  note  c2.sls_type = "pci" ;
  L1_d  L1(c1,c2,c3);
```

```
    L2_d  L2(c1,c2,c3);

    void main()
      {
        par {
          L1.main();
          L2.main();
        }
      }
  };
```

10.2 Communication Entities

Section 10.2.1 shows an example of listing of the communication entities. Each communication entity is assigned an ID number. The ID number is used as the major key for cross reference among SLS tools as well as in refined SpecC description output by SLS tools. The column entitled `#` contains the ID numbers.

Each number in the column entitled `GPL` indicates the origin of the associated communication entity. The communication entities associated with 0 are system-only variables which are invisible for SLS tools. The communication entities associated with 1 are global variables or channel instances, associated with 2 are port variables of the top behavior, and associated with 3 are variables and channel instances in the top behavior. The communication entities associated with 4 are real local variables (explained in Section 4.3) in all the descendent behavior instances of the top behavior.

Each number in the column entitled `PE` is the assigned hardware component type number of the associated communication entity. Each name in the column entitled `chnl_def` is the variable or channel type(or interface) of the associated communication entity.

The numbers in the column entitled `duration` are always zeros, since current SLS tools did not consider lifetime of communication entities yet.

10.2.1 An example of listing of communication entities

```
*** communication entity list
GPL: 0=system-only  1=global-var  2=parameter  3=top-bhvr-var  \
4=local-var
  # GPL PE  duration chnl_def tid -- (succ)== (pred):: (para)
   0  1  2   0.00000        * C --==::
   1  1  2   0.00000        * D --==::
   2  1  2   0.00000      int d100 --==::
   3  1  2   0.00000      int d40 --==::
   4  1  2   0.00000      int d50 --==::
   5  1  2   0.00000      int d60 --==::
   6  1  2   0.00000      int d70 --==::
   7  1  2   0.00000      int d80 --==::
   8  3  2   0.00000   sig_ch Main.c1 --==::
   9  3  3   0.00000   sig_ch Main.c2 --==::
  10  3  2   0.00000   sig_ch Main.c3 --==::
  11  4  2   0.00000      int Main.L1.B1.x --==::
  12  4  2   0.00000      int Main.L2.B4.x --==::
  13  4  2   0.00000      int Main.L2.B5.B5L.B7.x --==::
```

10.3 ETG Content

Section 10.3.1 shows a portion of listing of the nodes in the ETG. Similar to communication entity, each node is assigned an ID number.

Each number in the column entitled `NTP` indicates the type of the node. The node associated with 1 or 2 is a task, which is a real behavior instance or a communication entity with exclusive lifetime, respectly. The node associated 3 is a super-node which is a cluster of tasks connected by synchronizations. All the immediate successors of the super-node are those synchronized tasks. The node associated with 4 is a fork, and associated with 5 is a join.

The numbers in the columns entitled `PE`, `duration`, and `bhvr_def` are similar to that in communication entity listing, except the numbers in the `duration` column are certainly not always zeros.

At the end of each line are three groups of numbers which are separated by `--`, `==`, and `::`. The first group are successors of the node. The second group are predecessors of the nodes. The third group are port-maps of the node if its type is 1. The port-maps map the ports of the associated behavior instance to communication entities.

10.3.1 A portion of listing of of the nodes in an ETG

```
*** node dump
NTP: 1=behavior  2=channel  3=sync_group  4=fork  5=join
  # NTP PE  duration bhvr_def tid -- (succ)== (pred):: (para)
   0  1  4   0.10000     NOIO Main.L1.B1 -- 2== 20:: 2
   1  1  4   0.07000       IO Main.L1.B2 -- 4== 2:: 6 8 9
   2  5  0   0.00000          Main.L1.B2.in -- 1== 0 3::
   3  2  2   0.00000   sig_ch Main.c1 -- 2== 11::
   4  4  0   0.00000          Main.L1.B2.out -- 7 5== 1::
   5  2  3   0.00000   sig_ch Main.c2 -- 14== 4::
   6  1  4   0.05000        I Main.L1.B3 -- 21== 7:: 4 10
   7  5  0   0.00000          Main.L1.B3.in -- 6== 4 8::
   8  2  2   0.00000   sig_ch Main.c3 -- 7== 15::
   9  1  4   0.04000     NOIO Main.L2.B4 -- 18== 20:: 3
  10  1  4   0.05000        O Main.L2.B5.B5L.B6 -- 11== 18:: 4 8
```

11 Read Refined SpecC Description Output by `bnd2sir`

In this section, we only describe how to read refined SpecC descriptions output by `bnd2sir`. For details of how the refined descriptions are synthesized, please see[4]. We use a portion of a refined design description in Section 11.1 along with a portion of the display of `bnd2sir` in Section 11.2 to illustrate how to read the refined SpecC description generated by `bnd2sir`. The SpecC in Section 11.1 is created by using `scc` compiler to deparse the SIR generated by `bnd2sir`.

Every SIR_Definition generated by `bnd2sir` is prefixed with an *sls-header*. The *sls-header* is a string `_sls_#_`, where `#` is an integer. Since it may go through several iterations to refine the design, a SpecC description can be refined by `bnd2sir` several times. At each time, `bnd2sir` finds the biggest `#` of all the *sls-header*s in the description, then increases one as the new *sls-header*. If none of the `#` is found in the input SIR, `bnd2sir` uses `_sls_1_` as the *sls-header*;

`bnd2sir` generates a new top behavior, whose name is *sls-header* + `top_behavior`, as in line 643 in Section 11.1. The new top behavior has all the ports, variables, and channel instances of the original top behavior `Main`, as in line 485.

In the new top behavior, `bnd2sir` synthesizes each PE into a behavior instance. Each of the behavior instances is associated with name *sls-header* + `PE_#1_#2`, where `#1` is the PE type number and `#2` is the rank of the PE in that type of PEs. For example, `_sls_1_PE_4_0` in line 684 is a PE `P5-100` whose type number is 4, and so is `_sls_1_PE_4_1` in line 685. The behavior associated with each of these PE behavior instances is named *sls-header* + `PE_#1_#2_def`, for example, behavior `_sls_1_PE_4_0_def` in line 558.

Each PE behavior contains several tasks. `bnd2sir` synthesizes each task into a behavior instance associated with name *sls-header* + `BHVR_#`, where `#` is the node ID number of the ETG as described in Section 10.3. For example, node #6 in line 174 of the display in Section 11.2 is synthesized into behavior instance `_sls_1_BHVR_6` in line 564 of the refined description.

Similar to tasks, `bnd2sir` synthesizes each communication entity into a variable or channel instance associated with name *sls-header* + `CH_#` in the new top behavior, where `#` is the communication entity ID number as described in Section 10.2.1.

Variables in the new top behavior associated with names *sls-header* + `SIG_` + *anything*, for example, `_sls_1_SIG_0_2` in line 646, are signal wires which deliver execution sequence control signals. Behavior instances associated with names *sls-header* + `CTRL_` + *anything*, for example, `_sls_1_CTRL_19` in line 671, are glue logics or simply connections of signal wires. There are seven types of glue logics or connections needed for `bnd2sir` as shown in Section 11.3.

Each task with theoretical zero execution time is synthesized as an independent PE associated with name *sls-header* + `PE_dura0_#`, for example, `_sls_1_PE_dura0_1`. **Theoretical zero execution time(TZET)** means that it is so small that system-level scheduling can omit it, for example, the delay of a logic gate or a wire connection, or the designer knows it make no difference for scheduling so he sets `sls_dura` of the task to zero. `bnd2sir` synthesizes each TZET task into one PE so the refined description can produces correct simulation results, and these TZET can be further refined or moved into other PEs by the designer.

11.1 A portion of a refined design description

```
484  #line 231 "before.sc"
485  behavior Main ()
486  {
487
488  #line 489 "tmp.sc"
489          void main();
490
491  #line 233 "before.sc"
492          sig_ch c1(); sig_ch c2();
493          note c2.sls_type = "pci";
494
495  #line 233 "before.sc"
496          sig_ch c3();
497
498          L1_d L1(c1, c2, c3);
```

```
        L2_d L2(c1, c2, c3);

        void main()
                {
                par {
                L1.main();
                L2.main();}}
};
...
#line 558 "tmp.sc"
behavior _sls_1_PE_4_0_def (inout int CH_0_0, char *CH_6_1, \
inout int CH_4_2, char *CH_8_3, sig_in CH_16_4, sig_out CH_17_5, \
inout int CH_0_6, char *CH_14_7, sig_in CH_17_8, sig_out CH_18_9, \
inout int CH_2_10, char *CH_9_11, sig_in CH_18_12, \
in bool SIG_20_0, out bool SIG_0_2, in bool SIG_2_1, \
out bool SIG_1_4, in bool SIG_14_13, out bool SIG_13_15, \
in bool SIG_7_6, out bool SIG_6_21)
{
        void main(void );
        NOIO _sls_1_BHVR_0(CH_0_0, CH_6_1);
        IO _sls_1_BHVR_1(CH_4_2, CH_8_3, CH_16_4, CH_17_5);
        IO _sls_1_BHVR_13(CH_0_6, CH_14_7, CH_17_8, CH_18_9);
        I _sls_1_BHVR_6(CH_2_10, CH_9_11, CH_18_12);
        _sls_START _sls_1_SEND_SIG_0_2(SIG_0_2);
        _sls_START _sls_1_SEND_SIG_13_15(SIG_13_15);
        _sls_START _sls_1_SEND_SIG_1_4(SIG_1_4);
        _sls_START _sls_1_SEND_SIG_6_21(SIG_6_21);
        _sls_WAIT _sls_1_WAIT_START();
        void main(void )
        {
        fsm {
        _sls_1_WAIT_START: { if (SIG_20_0) goto _sls_1_BHVR_0;
        goto _sls_1_WAIT_START; }
        _sls_1_BHVR_0: { goto _sls_1_SEND_SIG_0_2; }
        _sls_1_SEND_SIG_0_2: { if (SIG_2_1) goto _sls_1_BHVR_1;
```

```
        goto _sls_1_SEND_SIG_0_2; }
        _sls_1_BHVR_1: { goto _sls_1_SEND_SIG_1_4; }
        _sls_1_SEND_SIG_1_4: { if (SIG_14_13) goto _sls_1_BHVR_13;
        goto _sls_1_SEND_SIG_1_4; }
        _sls_1_BHVR_13: { goto _sls_1_SEND_SIG_13_15; }
        _sls_1_SEND_SIG_13_15: { if (SIG_7_6) goto _sls_1_BHVR_6;
        goto _sls_1_SEND_SIG_13_15; }
        _sls_1_BHVR_6: { goto _sls_1_SEND_SIG_6_21; }
        _sls_1_SEND_SIG_6_21: { break; } }}
};
...
#line 643 "tmp.sc"
behavior _sls_1_top_behavior ()
{
void main();
bool _sls_1_SIG_0_2 = false;
bool _sls_1_SIG_10_11 = false;
bool _sls_1_SIG_11 = false;
bool _sls_1_SIG_12_14 = false;
bool _sls_1_SIG_13_15 = false;
bool _sls_1_SIG_14 = false;
bool _sls_1_SIG_15 = false;
bool _sls_1_SIG_17_19 = false;
bool _sls_1_SIG_18 = false;
bool _sls_1_SIG_19 = false;
bool _sls_1_SIG_1_4 = false;
bool _sls_1_SIG_2 = false;
bool _sls_1_SIG_20 = false;
bool _sls_1_SIG_21 = false;
bool _sls_1_SIG_3 = false;
bool _sls_1_SIG_4 = false;
bool _sls_1_SIG_5 = false;
bool _sls_1_SIG_6_21 = false;
bool _sls_1_SIG_7 = false;
bool _sls_1_SIG_8 = false;
```

```
bool _sls_1_SIG_9_18 = false;
_sls_OR_1 _sls_1_CTRL_11(_sls_1_SIG_10_11, _sls_1_SIG_11);
_sls_AND_2 _sls_1_CTRL_14(_sls_1_SIG_5, _sls_1_SIG_12_14, \
_sls_1_SIG_14);
_sls_OR_1 _sls_1_CTRL_15(_sls_1_SIG_13_15, _sls_1_SIG_15);
_sls_OR_1 _sls_1_CTRL_18(_sls_1_SIG_9_18, _sls_1_SIG_18);
_sls_AND_2 _sls_1_CTRL_19(_sls_1_SIG_15, _sls_1_SIG_17_19, \
_sls_1_SIG_19);
_sls_AND_2 _sls_1_CTRL_2(_sls_1_SIG_0_2, _sls_1_SIG_3, \
_sls_1_SIG_2);
_sls_START _sls_1_CTRL_20(_sls_1_SIG_20);
_sls_AND_2 _sls_1_CTRL_21(_sls_1_SIG_6_21, _sls_1_SIG_19, \
_sls_1_SIG_21);
_sls_OR_1 _sls_1_CTRL_3(_sls_1_SIG_11, _sls_1_SIG_3);
_sls_OR_1 _sls_1_CTRL_4(_sls_1_SIG_1_4, _sls_1_SIG_4);
_sls_OR_1 _sls_1_CTRL_5(_sls_1_SIG_4, _sls_1_SIG_5);
_sls_AND_2 _sls_1_CTRL_7(_sls_1_SIG_4, _sls_1_SIG_8, _sls_1_SIG_7);
_sls_OR_1 _sls_1_CTRL_8(_sls_1_SIG_15, _sls_1_SIG_8);
sig_ch c1();
sig_ch c2();
note c2.sls_type = "pci";
sig_ch c3();
_sls_1_PE_4_0_def _sls_1_PE_4_0(d100, nb1, d70, nb2, c1, c2, \
d100, nb8, c2, c3, d50, nb3, c3, _sls_1_SIG_20, _sls_1_SIG_0_2, \
_sls_1_SIG_2, _sls_1_SIG_1_4, _sls_1_SIG_14, _sls_1_SIG_13_15, \
_sls_1_SIG_7, _sls_1_SIG_6_21);
_sls_1_PE_4_1_def _sls_1_PE_4_1(d40, nb4, d50, nb6, c1, d70, nb7, \
d50, nb9, d60, nb10, _sls_1_SIG_20, _sls_1_SIG_9_18, \
_sls_1_SIG_18, _sls_1_SIG_10_11, _sls_1_SIG_11, _sls_1_SIG_12_14, \
_sls_1_SIG_18, _sls_1_SIG_17_19);
void main()
        {
        par {
        _sls_1_CTRL_2.main();
        _sls_1_CTRL_3.main();
```

```
        _sls_1_CTRL_4.main();
        _sls_1_CTRL_5.main();
        _sls_1_CTRL_7.main();
        _sls_1_CTRL_8.main();
        _sls_1_CTRL_11.main();
        _sls_1_CTRL_14.main();
        _sls_1_CTRL_15.main();
        _sls_1_CTRL_18.main();
        _sls_1_CTRL_19.main();
        _sls_1_CTRL_20.main();
        _sls_1_CTRL_21.main();
        _sls_1_PE_4_0.main();
        _sls_1_PE_4_1.main();}}
};
```

11.2 Display of bnd2sir

```
 PE type#  quantity  name
         0    0
         1    0 default
         2    0 chnl_default
         3    0 pci
         4    2 P5-100

TOP BEHAVIOR: Main

*** communication entity list
GPL: 0=system-only  1=global-var  2=parameter  3=top-bhvr-var  \
4=local-var
  # GPL   PE       chnl_def tid -- (succ)== (pred) ::(ref)
   0  1 chnl_defau      int d100
   1  1 chnl_defau      int d40
   2  1 chnl_defau      int d50
   3  1 chnl_defau      int d60
   4  1 chnl_defau      int d70
   5  1 chnl_defau      int d80
```

```
  6  1 chnl_defau        * nb1
  7  1 chnl_defau        * nb10
  8  1 chnl_defau        * nb2
  9  1 chnl_defau        * nb3
 10  1 chnl_defau        * nb4
 11  1 chnl_defau        * nb5
 12  1 chnl_defau        * nb6
 13  1 chnl_defau        * nb7
 14  1 chnl_defau        * nb8
 15  1 chnl_defau        * nb9
 16  3 chnl_defau   sig_ch Main.c1
 17  3 pci          sig_ch Main.c2
 18  3 chnl_defau   sig_ch Main.c3

*** node dump
NTP: 1=behavior  2=channel  3=sync_group  4=fork  5=join
  # NTP   PE      PE# schedule bhvr_def tid -- (succ)== (pred) ::(ref)
   0  1 P5-100     0   0.00000     NOIO Main.L1.B1 -- 2== 20:: 0 6
   1  1 P5-100     0   0.10000       IO Main.L1.B2 -- 4== 2:: 4 8 16 17
   2  5            c   0.10000          Main.L1.B2.in -- 1== 0 3::
   3  2 chnl_defau -   0.09000   sig_ch Main.c1 -- 2== 11::
   4  4            c   0.17000          Main.L1.B2.out -- 7 5== 1::
   5  2 pci        -   0.17000   sig_ch Main.c2 -- 14== 4::
   6  1 P5-100     0   0.27000        I Main.L1.B3 -- 21== 7:: 2 9 18
   7  5            c   0.27000          Main.L1.B3.in -- 6== 4 8::
   8  2 chnl_defau -   0.27000   sig_ch Main.c3 -- 7== 15::
   9  1 P5-100     1   0.00000     NOIO Main.L2.B4 -- 18== 20:: 1 10
  10  1 P5-100     1   0.04000        O Main.L2.B5.B5L.B6 -- 11== \
18:: 2 12 16
  11  4            c   0.09000          Main.L2.B5.B5L.B6.out -- \
12 3== 10::
  12  1 P5-100     1   0.09000     NOIO Main.L2.B5.B5L.B7 -- 14== \
11:: 4 13
  13  1 P5-100     0   0.17000       IO Main.L2.B5.B5L.B8 -- 15== \
14:: 0 14 17 18
```

```
  14  5            c   0.17000         Main.L2.B5.B5L.B8.in -- \
13== 5 12::
  15  4            c   0.27000         Main.L2.B5.B5L.B8.out -- \
19 8== 13::
  16  1 P5-100     1   0.16000    NOIO Main.L2.B5.B5R.B9 -- 17== \
18:: 2 15
  17  1 P5-100     1   0.21000    NOIO Main.L2.B5.B5R.B10 -- 19== \
16:: 3 7
  18  4            c   0.04000         Main.L2.B5.fork -- 10 16== 9::
  19  5            c   0.27000         Main.L2.B5.join -- 21== 15 17::
  20  4            c   0.00000         Main.fork -- 0 9==::
  21  5            c   0.32000         Main.join --== 6 19::

New top behavior: _sls_1_top_behavior

*** control box: 2 3 4 5 7 8 11 14 15 18 19 20 21

*** behavior instance -- tasks in the behavior
_sls_1_PE_4_0 -- 0 1 13 6
_sls_1_PE_4_1 -- 9 10 12 16 17
```

11.3 Behaviors of Glue Logics and Connections Needed by `bnd2sir`

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

behavior _sls_WAIT()
{
  bool first_time=true;
  void main(){
    if(first_time) first_time=false;
    else wait _sls_event;
  }
};
```

```
behavior _sls_START(out bool OUT)
{
  bool first_time=true;
  void main(){
    if(first_time){
      first_time=false;
      OUT=true;
      notify _sls_event;
    }
    else wait _sls_event;
  }
};

behavior _sls_SYNC_START(out bool RDY,in bool START)
{
  void main(){
    RDY=true;
    notify _sls_event;
    while(!START)
      wait _sls_event;
  }
};

// # is a parameter
behavior _sls_SYNC_#(in bool IN,out bool OUT,
             in bool RDY_0,in bool RDY_1, ... in bool RDY_(#-1),
             out bool START)
{
  void main(){
    while(!IN) wait _sls_event;
    OUT=true;
    notify _sls_event;
    while(!(RDY_0 && RDY_1 && ... && RDY_(#-1)))
      wait _sls_event;
```

```
    START=true;
    notify _sls_event;
  }
};

// # is a parameter
behavior _sls_AND_#(in bool IN_0,in bool IN_1, ... in bool IN_(#-1),
            out bool OUT)
{
  void main(){
    while(!(IN_0 && IN_1 && ... & IN_(#-1)))
      wait _sls_event;
    OUT=true;
    notify _sls_event;
  }
};

// # is a parameter
behavior _sls_OR_#(in bool IN_0,in bool IN_1, ... in bool IN_(#-1),
           out bool OUT)
{
  void main(){
    while(!(IN_0 || IN_1 || ... || IN_(#-1)))
      wait _sls_event;
    OUT=true;
    notify _sls_event;
  }
};
```

References

[1] R. Dömer, J. Zhu, and D. D. Gajski, "The specc language reference manual." UC Irvine, Dept. of ICS, Technical Report 98-13,March 1998.

[2] R. Dömer, "The specc internal representation." UC Irvine, Dept. of ICS, Technical Report 99-??,January 1999.

[3] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Transactions on Computer-Aided Design*, June 1989.

[4] E.-S. Chang, *Algorithms for System Synthesis.* PhD thesis, University of California, Irvine, maybe 1999.