Title
The SpeC language reference manual

Authors
Domer, Rainer
Zhu, Jianwen
Gajski, Daniel D.

Publication Date
1998-03-31

Peer reviewed

ICS

TECHNICAL REPORT

The SpecC Language Reference Manual

Rainer Dömer
Jianwen Zhu
Daniel D. Gajski

Technical Report ICS-98-13
March 1998
(Version as of March 31, 1998)

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

doemer@ics.uci.edu
jzhu@ics.uci.edu
gajski@ics.uci.edu

Abstract

This Language Reference Manual defines the syntax and semantics of the SpecC language. For each SpecC construct the syntax, purpose, and semantics are defined and an explaining example is given. Also the full SpecC grammar is included using a formal notation in lex and yacc style.

Information and Computer Science
University of California, Irvine

Contents

1 Introduction

The SpecC language was introduced in [3, 4, 5] as a system level language for modeling embedded systems. This report contains the SpecC Language Reference Manual and formally defines the syntax and the semantics of the language.

The SpecC language is an extension of the C programming language and is based on the ANSI-C standard [1]. In this report the ANSI-C language is assumed to be known. The report only describes features of the SpecC language that are not found in ANSI-C.

In order to support important concepts needed for modelling embedded systems SpecC adds new keywords and constructs to the C language. Section 2 defines these special data types, statements and constructs of the SpecC language.

As a reference the appendix contains the full SpecC grammar formally defined using a `lex` and `yacc` style notation.

2 Semantics of the SpecC language

This section defines all types, statements, operations and constructs of the SpecC language that are not part of ANSI-C.

2.1 Boolean Type

Purpose: Explicit support for Boolean data types

Synopsis:

```
basic_type_name:                    /* BasicType */
        ...
        | bool

constant:                           /* Expression */
        ...
        | false
        | true
```

Example:

```
bool f(bool b1, int a)
{
    bool b2;

    if (b1 == true)
       { b2 = b1 || (a > 0);
       }
    else
       { b2 = !b1;
       }
    return(b2);
}
```

Semantics:

A Boolean value, the type **bool**, can have two values: **true** or **false**. It can be used to express the result of logical operations. If converted to an integer type, **true** becomes 1, **false** becomes 0.

2.2 Bitvector Type

Purpose: Direct support of bitvectors of arbitrary length

Synopsis:

```
bindigit        [01]
binary          {bindigit}+
bitvector       {binary}("b"|"B")
bitvector_u     {binary}("ub"|"bu"|"uB"|"Bu"|"Ub"|"bU"|"UB"|"BU")

basic_type_name:                  /* BasicType */
        ...
        | bit '[' constant_expression ':' constant_expression ']'

constant:                         /* Expression */
        ...
        | bitvector
        | bitvector_u

postfix_expression :              /* Expression */
        ...
        | postfix_expression '[' comma_expression ':' comma_expression ']'

concat_expression :               /* Expression */
        cast_expression
        | concat_expression '@' cast_expression
```

Example:

```
typedef bit[3:0]           nibble_type;
nibble_type                a;
unsigned bit[15:0]         c;

void f(nibble_type b, bit[16:1] d)
{
    a = 1101B;                        /* bitvector assignment */
    c = 1110001111100011ub;

    b = c[2:5];                       /* bitvector slicing */
    d = a @ b @ c[0:15];              /* bitvector concatenation */
    b += 42 + a * 12;                 /* arithmetic operations */
    d = ~(b | 10101010B);             /* logic operations */
}
```

Semantics:

A bitvector represents an integral data type of arbitrary length. It can be used as any other integral type (eg. **int** is equivalent to **bit**$[sizeof(int) * 8 - 1 : 0]$). Implicit promotion to (**unsigned**) **int**, **long**, or **long long** is performed when necessary. Automatic conversion (signed or unsigned extension, slicing) is supported as with any other integral type. No explicit type casting is necessary.

A bitvector can be thought of as a parameterized type whose bounds are defined in the name of the type. The left and right bounds of a bitvector can be constant expressions but will be evaluated at compile time. Therefore the length of any bitvector is always known at compile time.

Bitvector constants are noted as a sequence of zeros and ones followed by a suffix indicating the bitvector type (see the synopsis above).

In addition to all standard C operations a concatenation operation, noted as @, and an extraction operation, noted as $[lb : rb]$, are supported (see lines 10 and 11 in the example above). Both can be applied to bitvectors as well as to any other integral type (which will be treated as bitvector of suitable length).

2.3 Event Type

Purpose: Support for events as a mechanism for synchronization and exception handling

Synopsis:

```
basic_type_name :                    /* BasicType */
        ...
        | event

wait_statement :                     /* Statement */
        wait paren_event_list ';'

notify_statement :                   /* Statement */
        notify paren_event_list ';'
        | notifyone paren_event_list ';'
```

Example:

```
int     d;
event   e;

void send(int x)
{
    d = x;
    notify e;
}

int receive(void)
{
    wait e;
    return(d);
}
```

Semantics:

An event type is a special type to support synchronization of concurrent executing behaviors and exception handling.

An event does *not* have a value. Therefore, an event must not be used in any expression.

Events can only be used with **wait** and **notify** statements (see the example above or Section 2.15), or with the **try-trap-interrupt** construct described in Section 2.14.

2.4 Time Type

Purpose: Simulation time with support of timed (hardware) and untimed behavior (software)

Synopsis:

```
primary_expression :                    /* Expression */
        ...
        | delta

waitfor_statement :                     /* Statement */
        waitfor time ';'

time :                                  /* Expression */
        constant_expression
```

Example:

```
extern void        f(void);
const long int     CycleTime = 15; /* ns */

void Timed(void)
{
    while(true)
        { f();
          waitfor(CycleTime);
        }
}

void Untimed(void)
{
    while(true)
        { f();
          waitfor(delta);
        }
}
```

Semantics:

The time type represents the type of simulation time. Time is not an explicit type. It is an implementation dependent integral type (eg. **unsigned long long**).

The time type is used only with the **waitfor** statement and in the **do-timing** construct (see Section 2.16).

For untimed behavior (behavior with unknown timing, eg. software) the **delta** time variable is supported. The **delta** variable is of type time and is implementation and simulator dependent. At simulation time it is evaluated to the time spend executing the current behavior on the host machine since the last simulator entry. *NOTE: This section needs more work!*

2.5 Behavior Class

Purpose: Construct for specification of behavioral objects; container for functionality

Synopsis:

```
behavior_declaration:               /* void */
        behavior_specifier  port_list_opt  ';'

behavior_definition :               /* void */
        behavior_specifier  port_list_opt  '{'  internal_def._list_opt  '}' ';'

behavior_specifier :                /* Declarator */
        behavior identifier
```

Example:

```
behavior B (in int p1, out int p2)
{
   int  a, b;

   int f(int x)
      {
        return(x * x);
      }

   void main(void)
      {
        a = p1;              /* read data from input port */
        b = f(a);            /* compute */
        p2 = b;              /* output to output port */
      }
};
```

Semantics:

In SpecC the functionality of a system is described by a hierarchical network of behaviors. A **behavior** is a class that consists of an optional set of ports, an optional set of instantiations, an optional set of local variables and functions, and a main function.

Through its ports a behavior can communicate with other behaviors. This is described in detail in Section 2.8.

A behavior is called a composite behavior if it contains instantiations of other behaviors (as described in Section 2.9). Otherwise it is called a leaf behavior.

Local variables and functions, as a, b, and f in the example above, can be used to conveniently program a behaviors functionality. The function `main` of a behavior is the only one that can be called from outside the behavior (callback functions of instantiated interfaces, as described in Section 2.7, are an exception to this rule). The `main` function is called whenever an instantiated behavior is executed and the completion of `main` determines the completion of the behaviors execution.

A behavior is compatible with another behavior if the number and the types of the behavior ports match. Compatibility of behaviors is important for reuse and "plug-and-play".

The example above shows a simple leaf behavior. For typical composite behaviors please refer to Sections 2.10 to 2.14.

2.6 Channel Class

Purpose: Construct for specification of channel objects; container for communication protocols

Synopsis:

```
channel_declaration :                /* void */
        channel_specifier  port_list_opt  implements_interface_opt  ';'

channel_definition :                 /* void */
        channel_specifier  port_list_opt  implements_interface_opt
                '{'  internal_definition_list_opt  '}'  ';'

channel_specifier :                  /* Declarator */
        channel  identifier

implements_interface_opt :           /* SymbolPtrList */
        /* nothing */
        | implements  interface_list

interface_list :                     /* SymbolPtrList */
        interface_name
        | interface_list  ','  interface_name
```

Example:

```
interface I;

channel C (void) implements I
{
    int  data;

    void send(int x)
       {
          data = x;
       }

    int  receive(void)
       {
          return(data);
       }
};
```

Semantics:

Communication between behaviors can be encapsulated in channels. A **channel** is a class that consists of an optional set of ports, an optional set of instantiations, and an optional set of local variables and functions called methods. Also, a channel can have a list of supported interfaces specified after the **implements** keyword.

A channel can include a list of ports through which it can communicate with other channels or behaviors (although channel ports are rarely used). Ports are described in detail in Section 2.8.

A channel is called a hierarchical channel if it contains instantiations of other channels (as described in Section 2.9). A channel is called a wrapper if it instantiates behaviors.

In general variables and functions (methods) defined in a channel can be accessed from outside (just like members of structures). By using interfaces (defined in Section 2.7) only a subset of the internal methods can be made public. The **implements** keyword declares the list of implemented interfaces. All the methods of the implemented interfaces must be defined inside the channel.

A channel is compatible with another channel if the number and the types of the channel ports, and the list of the implemented interfaces match.

The example above shows a simple channel providing a simple communication via an integer variable.

2.7 Interface Class

Purpose: Link between behaviors and channels; support for "Plug-and-Play"

Synopsis:

```
interface_declaration :            /* void */
        interface_specifier ';'

interface_definition :             /* void */
        interface_specifier '{' internal_declaration_list_opt '}' ';'

interface_specifier :              /* Declarator */
        interface identifier

internal_declaration_list_opt :    /* void */
        /* nothing */
        | internal_declaration
        | internal_declaration_list internal_declaration

internal_declaration :             /* void */
        declaration
        | callback declaration
        | note_definition
```

Example:

```
interface I
{
   void send(int x);
   int receive(void);
};

interface I2
{
   void send_block(void);
   callback int get_data(void);

   void receive_block(void);
   callback void put_data(int d);
};
```

Semantics:

Interfaces can be used to connect behaviors with channels in a way so that both the behaviors and the channels are easily exchangable with compatible components ("plug-and-play"). An **interface** is a class that consists of a set of variable or function declarations. The definitions of these declarations are contained in a channel that **implements** the interface.

A typical use of an interface is a behavior with a port of interface type. Via an interface port a behavior has access to all communication methods declared in that interface. For each interface multiple channels can provide an implementation of the declared communication functions and each of these channels then can be connected to the behavior with the interface port.

The example above shows an **interface** I declaring two functions *send* and *receive*. It is this interface that the channel C in the example from Section 2.6 implements.

The **interface** $I2$ in the example above defines a communication scheme involving **callback** functions. A **callback** function is a method that must be supplied by the class instantiating the interface (eg. the behavior, not the channel). So a behavior with an interface port $I2$ must contain definitions of the functions *get_data* and *put_data*, whereas a channel implementing interface $I2$ must define only the functions *send_block* and *receive_block* which can call back the functions *get_data* and *put_data*.

2.8 Ports

Purpose: Specification of connectors of behaviors and channels

Synopsis:

```
port_list_opt :                    /* ParameterList */
        /* nothing */
        | '(' ')'
        | '(' port_list ')'

port_list :                        /* ParameterList */
        port_declaration
        | port_list ',' port_declaration

port_declaration :                 /* Parameter */
        port_direction parameter_declaration
        | interface_name
        | interface_name identifier
        | channel_name
        | channel_name identifier

port_direction :                   /* Direction */
        /* nothing */
        | in
        | out
        | inout
```

Example:

```
interface I;

behavior B1 (in int p1, out int p2, in event clk);

behavior B2 (I i, inout event clk);

channel  C  (inout bool f) implements I;
```

Semantics:

Behavior and channel classes can have a list of ports through which they communicate. These ports are defined with the definition of the behavior or channel they are attached to (exactly like function parameters are defined with the function definition).

A port can be of three types: standard, interface, or channel type. A standard type port can be of any type, but may include its direction as an additional type modifier. A port direction can be **in**, **out**, or **inout**, and is handled as an access restriction to that port. An **in** port only allows read-access from inside the class (write-access from outside), an **out** port only allows write-access from the inside (read-access from outside). An **inout** port can be accessed bi-directionally, as can a port without any direction modifier.

An interface or channel type port enables access to the methods of that interface or channel class. Via such a port a behavior or a channel can call the methods of the ports class.

2.9 Class Instantiation

Purpose: Specification of behavioral hierarchy and connectivity among behaviors and channels

Synopsis:

```
instance_declaring_list :            /* DeclarationSpec */
        behavior_or_channel instance_declarator
        | instance_declaring_list ',' instance_declarator

instance_declarator :                /* Declarator */
        identifier port_mapping_opt

behavior_or_channel :                /* DeclarationSpec */
        behavior_name
        | channel_name

port_mapping_opt :                   /* ParameterList */
        /* nothing */
        | '(' ')'
        | '(' port_mapping_list ')'

port_mapping_list :                  /* ParameterList */
        identifier
        | port_mapping_list ',' identifier
```

Example:

```
interface I;
channel   C  (inout bool f) implements I;
behavior  B1 (in int p1, out float p2, in event clk);
behavior  B2 (I i, out event clk);

behavior B (float f1, float f2)
{
   bool          b;
   int           i;
   event         e;

   C    c(b);             /* instantiate c as channel C */
   B1   b1(i, f1, e),     /* instantiate b1 and b3 as behavior B1 */
        b3(i, f2, e);
   B2   b2(c, e);         /* instantiate b2 as behavior B2 */
};
```

Semantics:

SpecC supports behavioral hierarchy by allowing (sub-) behaviors and (sub-) channels, called components, to be instantiated inside compound behaviors and channels. The instantiation of behaviors and channels also defines the connectivity among the instantiated components and with the compound class.

An instantiation defines its connections by use of a port mapping list. Each port of the instantiated class must be mapped onto a corresponding variable or port of suitable type. A port must match the type of the mapped variable or port, just as the types or arguments to a function call must match the types of the function parameters. Also, the port directions must be compatible. As a rule, for each connection there can be at most one driver (no two **out** ports can be connected, whereas two **in** ports can).

The example above contains four class instantiations. In line 12 a channel c is instantiated as type channel C. Its only port of type **bool** is mapped to the Boolean variable b.

Lines 13 and 14 instantiate two behaviors $b1$ and $b3$ (of type behavior $B1$) which are both connected to integer i and event e. The second port of $b1$ is connected to the first port of B, whereas the second port of $b3$ is mapped to the second port of B.

In line 15 $b2$ is instantiated as a $B2$ type behavior. Its ports are mapped to the channel c (instantiated in line 12) and event e.

2.10 Sequential Execution

Purpose: Specification of sequential control flow

Synopsis:

```
statement:                                  /* Statement */
        labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        | spec_c_statement

spec_c_statement:                           /* Statement */
        concurrent_statement
        | fsm_statement
        | exception_statement
        | timing_statement
        | wait_statement
        | waitfor_statement
        | notify_statement
```

Example:

```
behavior B;

behavior B_seq(void)
{
    B       b1, b2, b3;

    void main(void)
    {
        b1.main();
        b2.main();
        b3.main();
    }
};
```

Semantics:

Sequential execution of statements and behaviors is the same as in standard C. The sequential control flow can be programmed using the standard C constructs **if-then-else**, **switch-case**, **goto**, **for**, **while**, etc.

The example above shows the trivial case of sequential, unconditional execution of three behaviors.

2.11 Parallel Execution

Purpose: Specification of concurrency

Synopsis:

```
concurrent_statement:                    /* Statement */
        ...
        | par compound_statement

compound_statement:                      /* Statement */
        '{' '}'
        | '{' declaration_list '}'
        | '{' statement_list '}'
        | '{' declaration_list statement_list '}'
```

Example:

```
behavior B;

behavior B_par(void)
{
    B     b1, b2, b3;

    void main(void)
    {
        par{ b1.main();
             b2.main();
             b3.main();
           }
    }
};
```

Semantics:

Concurrent execution of statements can be specified with the **par** statement. Every statement in the compound statement block following the **par** keyword formes a new thread of control and is executed in parallel. The execution of the **par** statement completes when each thread of control has finished its execution.

Usually concurrent execution is used in the behavioral hierarchy in order to execute instantiated behaviors in parallel. This is shown in the example above where the behaviors *b1*, *b2* and *b3* are running concurrently. The compound behavior *B_par* finishes when *b1*, *b2* and *b3* have completed.

Note that in a simulation concurrent threads of control are not really executed in parallel. Instead the scheduler, which is part of the simulation run-time system, always executes one thread at a time, and decides when to suspend and when to resume a thread depending on simulation time advance and synchronization points.

2.12 Pipelined Execution

Purpose: Explicit support for specification of pipelining

Synopsis:

```
storage_class :                          /* BasicType */
        ...
        | piped
        | storage_class piped

concurrent_statement :                   /* Statement */
        ...
        | pipe compound_statement

compound_statement:                      /* Statement */
        '{' '}'
        | '{' declaration_list '}'
        | '{' statement_list '}'
        | '{' declaration_list statement_list '}'
```

Example:

```
behavior B(in int p1, out int p2);

behavior B_pipe(in int a, out int b)
{
    int          x;
    piped int    y;
    B            b1(a, x),
                 b2(x, y),
                 b3(y, b);

    void main(void)
    {
        pipe{ b1.main();
              b2.main();
              b3.main();
            }
    }
};
```

Semantics:

Pipelined execution specified by the **pipe** statement is a special form of concurrent execution. As is with the **par** statement all statements in the compound statement block after the **pipe** keyword form a new thread of control. They are executed in a pipelined fashion (in parallel but obey the specification order). The **pipe** statement never finishes (except through abortion which is described in Section 2.14).

For example, as shown above, the behaviors $b1$, $b2$ and $b3$ form a pipeline of behaviors. In the first iteration only $b1$ is executed. When $b1$ finishes the second iteration starts and $b1$ and $b2$ are executed in parallel. In the third iteration, after $b1$ and $b2$ have completed, $b3$ also is executed in parallel with $b1$ and $b2$. Every next iteration is the same as the third iteration (iteration three is repeated forever).

In order to support buffered communication in pipelines the **piped** storage class can be specified for variables connecting pipeline stages. A variable with **piped** storage class can be thought of as a variable with two storages. Write access always writes to the first storage, read access reads from the second storage. The contents of the first storage are shifted to the second storage whenever a new iteration starts in the **pipe** statement.

In the example above x is a standard variable connecting $b1$ (pipeline stage 1) with $b2$ (stage 2). This variable is not buffered, every access from stage 1 is immediately reflected in stage 2. On the other hand, variable y connecting $b2$ and $b3$ is buffered. A result that is computed by behavior $b2$ and stored in y is available for processing by $b3$ in the next iteration when $b2$ already produces new data.

Note that the **piped** storage class can be specified n times defining a variable with n buffers. This can be used to transfer data over n stages synchronously with the pipeline.

2.13 Finite State Machine Execution

Purpose: Explicit support for specification of finite state machines and their state transitions

Synopsis:

```
fsm_statement:                    /* Statement */
        fsm '{' '}'
        | fsm '{' transition_list '}'

transition_list:                  /* TransitionList */
        transition
        | transition_list transition

transition:                       /* Transition */
        identifier ':'
        | identifier ':' cond_branch_list
        | identifier ':' '{' '}'
        | identifier ':' '{' cond_branch_list '}'

cond_branch_list:                 /* TransitionList */
        cond_branch
        | cond_branch_list cond_branch

cond_branch:                      /* Transition */
        if '(' comma_expression ')' goto identifier ';'
        | goto identifier ';'
        | if '(' comma_expression ')' break ';'
        | break ';'
```

Example:

```
behavior B;

behavior B_fsm(in int a, in int b)
{
   B     b1, b2, b3;

   void main(void)
   {
     fsm{ b1: { if (b < 0)  break;
                if (b >= 0) goto b2;
              }
          b2: { if (a > 0)  goto b1;
                goto b3;
              }
```

```
            b3: { break;
                }
        }
    }
};
```

Semantics:

Finite State Machine (FSM) execution is a special form of sequential execution allowing explicit specification of state transitions. Both Mealy and Moore type finite state machines can be modeled with the **fsm** construct.

As shown in the synopsis section above the **fsm** construct specifies a list of state transitions where the states are instantiated behaviors. A state transition is a triple $\langle current\_state, condition, next\_state \rangle$. The *current_state* and the *next_state* take the form of labels and denote behavior instances. The *condition* is an expression which has to be evaluated as **true** for the transition to become valid.

The execution of a **fsm** construct starts with the execution of the first behavior that is listed in the transition list (eg. $b1$). Once the behavior has finished, its transitions determine the next behavior to be executed. The conditions of the transitions are evaluated in the order they are specified (first $b < 0$, then $b \geq 0$) and as soon as a condition is **true** the specified next behavior is started (eg. $b2$ for $b = 1$). If none of the conditions is true the next behavior defaults to the next behavior listed (similar to a **case** statement without **break**). A **break** statement terminates the execution of the **fsm** construct.

Note that the body of the **fsm** construct does not allow arbitrary statements. As specified in the synopsis section the grammar limits the transitions to well-defined triples.

2.14 Exception Handling

Purpose: Support for (premature) abortion of execution and interrupt handling

Synopsis:

```
exception_statement:                /* Statement */
        try compound_statement exception_list

exception_list:                     /* ExceptionList */
        exception
        | exception_list exception

exception:                          /* Exception */
        trap paren_event_list compound_statement
        | interrupt paren_event_list compound_statement

paren_event_list:                   /* SymbolPtrList */
        event_list
        | '(' event_list ')'

event_list:                         /* SymbolPtrList */
        identifier
        | event_list ',' identifier
```

Example:

```
behavior B;

behavior B_except(in event e1, in event e2)
{
   B      b1, b2;

   void main(void)
      {
         try { b1.main(); }
             interrupt (e1) { b2.main(); }
             trap (e2)      { b1.main(); }
      }
};
```

Semantics:

The **try-trap-interrupt** construct deals with two types of exception handling: abortion (or trap) and interrupt.

With **try** a behavior is made sensitive to the events listed with the **trap** and **interrupt** declarations. Whenever such an event occurs while executing the **try** behavior its execution is immediately suspended. For an **interrupt** event the specified interrupt handler is executed and after its completion the execution of the **try** behavior is resumed. For a **trap** event the suspended execution is aborted and the trap handler takes over the execution.

In the example above, whenever event $e1$ is notified during execution of behavior $b1$, the execution of $b1$ is immediately suspended and behavior $b2$ is started. When $b2$ finishes the execution of behavior $b1$ is resumed. Note that during execution of $b2$ the event $e1$ is ignored (the interrupt does not interrupt itself). Also, as soon as event $e2$ occurs while executing behavior $b1$ the current execution is aborted and $b1$ is restarted. If **try** and **trap** denote the same behavior, as is in this case $b1$, effectively a reset is modeled.

As a rule, **interrupt** and **trap** declarations are prioritized in the order they are listed. Always only the first listed exception that matches an event is executed.

2.15 Synchronization

Purpose: Support for synchronization of concurrent behaviors

Synopsis:

```
wait_statement :                      /* Statement */
        wait paren_event_list ';'

notify_statement :                    /* Statement */
        notify paren_event_list ';'
        | notifyone paren_event_list ';'

paren_event_list :                    /* SymbolPtrList */
        event_list
        | '(' event_list ')'

event_list :                          /* SymbolPtrList */
        identifier
        | event_list ',' identifier
```

Example:

```
event   e;

behavior b1(int x, event s)
{
    void main(void)
    {
        x = 42;
        notify s;
        ...
        notify(e, s);
    }
}

behavior b2(int x, event r)
{
    void main(void)
    {
        wait(r);
        printf("%d", x);
        ...
        wait(e, s);
    }
}
```

Semantics:

There are three statements to support synchronization between concurrent executing behaviors: **wait**, **notify** and **notifyone**. Each of these statements takes a list of events (described in Section 2.3) as argument.

The **wait** statement suspends the current behavior from execution until one of the specified events is notified by another behavior. The execution of the waiting behavior is then resumed.

Note that when waiting for a list of events the **wait** statement provides no information to determine which of the specified events actually was notified. This limitation is not a bug, it is a feature of pure event semantics.

The **notify** statement triggers all specified events so that all behaviors waiting on one of those events can continue their execution. If currently no other behavior is waiting on the notified events the notification is ignored.

The **notifyone** statement acts similar as the **notify** statement but notifies exactly one behavior from all behaviors waiting on the specified events. Again, if there is no behavior waiting the notification has no effect.

2.16 Timing Specification

Purpose: Explicit specification of execution time, delay and timing constraints

Synopsis:

```
waitfor_statement :                 /* Statement */
        waitfor time ';'

timing_statement :                  /* Statement */
        do statement timing '{' constraint_list '}'

constraint_list :                   /* ConstraintList */
        constraint
        | constraint_list constraint

constraint :                        /* Constraint */
        range '(' any_name ';' any_name ';' time_opt ';' time_opt ')' ';'

time_opt :                          /* Constant */
        /* nothing */
        | time

time :                              /* Expression */
        constant_expression
```

Example:

```
void ClockGen(int *clk, int *clk2)
{
    do { t1: { *clk = 1; *clk2 = 1; }
         t2: { *clk = 0; }
         t3: { *clk = 1; *clk2 = 0; }
         t4: { *clk = 0; break; }
       }
    timing
       { range(t1; t2; 110; 112);
         range(t2; t3; 110; 112);
         range(t3; t4; 110; 112);
         range(t1; t4; 332;    );
       }
}
```

Semantics:

There are two constructs that support the specification of timing (simulation time).

First, the **waitfor** statement specifies delay or execution time. Whenever the simulator reaches a **waitfor** statement, the execution of the current behavior is suspended. As soon as the simulation time is increased by the number of time units specified in the argument the execution of the current behavior resumes.

Second, the **do-timing** construct can be used to specify timing constraints in terms of minimum and maximum times. In the construct the **do** block defines labeled statements which will be executed according to the constraints specified in the **timing** block. The order of the execution of the labeled statements is determined solely by the constraints. The execution of a **do-timing** construct completes when a **break** statement is executed.

Timing constraints are specified with **range** statements. Each constraint consists of two labels linking the constraint to its actions, and a minimum and maximum time value. The minimum and maximum times are optional constant expressions which will be evaluated at compile time. If unspecified the minimum time is taken as $-\infty$, the maximum time as $+\infty$.

The semantics of a statement **range**$(l1, l2, min, max)$ is the following: The statement labeled $l1$ is to be executed at least min time steps before, but not more than max time steps after the statement labeled with $l2$.

Note that the **do-timing** construct is not directly executable. In order to get a simulatable model each **do-timing** construct has to be scheduled so that all constraints are satisfied. Therefore the SpecC compiler performs an ASAP scheduling for each **do-timing** construct and generates code containing **waitfor** statements instead.

2.17 Binary Import

Purpose: Fast and easy reuse of library components

Synopsis:

```
import_definition :                /* void */
        import string_literal_list ';'

string_literal_list :              /* String */
        string
        | string_literal_list string
```

Example:

```
#include <stdio.h>
#include <stdlib.h>

import "Interfaces/I1";
import "Interfaces/I2";
import "Channels/PCI_Bus";
import "Components/MPEG_II";
```

Semantics:

For using objects declared or defined in separate files of SpecC source code two constructs are provided. First, the `#include` statement known from the standard C language can be used. It will be evaluated at preprocessing time by the C preprocessor.

Second, the **import** declaration provides an efficient way to incorporate already compiled components. Using the SpecC compiler any SpecC source description can be compiled into a binary file (with suffix `.sir`) containing the SpecC internal representation. Such a file can be included using the **import** declaration which effectively integrates all declarations and definitions from the binary file into the current design representation.

The string argument of the **import** declaration denotes the file name of the binary component to be integrated. The actual search for the binary file is implementation dependent but usually involves applying the suffix `.sir` and searching in a list of specified directories.

2.18 Persistent Annotation

Purpose: Support of persistent design annotation; easy data exchange between refinement tools

Synopsis:

```
any_declaration:                    /* void */
        ...
        | note_definition

any_definition:                     /* void */
        ...
        | note_definition

note_definition:                    /* void */
        note any_name '=' annotation ';'
        | note any_name '.' any_name '=' annotation ';'

annotation:                         /* Constant */
        constant_expression

any_name:                           /* Name */
        identifier
        | typedef_name
        | behavior_name
        | channel_name
        | interface_name
```

Example:

```
/* C style comment, not persistent */
// C++ style comment, not persistent

note Author      = "Rainer␣Doemer";
note Date        = "Thu␣Mar␣26␣13:46:30␣PST␣1998";

const int x = 42;
struct S { int a, b; float f; };

note x.Size      = sizeof(x);
note S.Bits      = sizeof(struct S) * 8;

behavior B(in int a, out int b)
{
   note Version   = 1.1;

```

```
17     void main(void)
18     {
19        l1: b = 2 * a;
20            waitfor(10);
21        l2: b = 3 * a;
22
23        note NumOps  = 3;
24        note l1.OpID = 1;
25        note l2.OpID = 3;
26     }
27 };
28 note B.AreaCost = 12345;
```

Semantics:

SpecC, as does any other programming language, allows comments in the source code to annotate the description. In particular SpecC supports the same comment styles as C++, which are comments enclosed in `/*` and `*/` delimiters as well as comments after `//` up to the end of the line (see lines 1 and 2 in the example above). These comments are not persistent, which means they are not stored in the SpecC internal representation.

Using the **note** definition a persistent annotation can be attached to any symbol, label, and user-defined type. An annotation consists of a name and a note. The note can be any type of constant or constant expression (evaluated at compile time).

Names of notes have their own name space. There is no name conflict possible with symbols, user-defined types or labels.

There are two ways to define an annotation. First, a note can be attached to the current scope. This way global notes (lines 4 and 5 in the example), notes at classes (line 15), notes at functions (line 23), and notes at user-defined types can be defined.

Second, the object a note will be attached to can be named explicitly by preceding the note name with the object name and a dot. In the example above this style is used to define the notes at variable x (line 10), structure S (line 11), and labels $l1$ and $l2$ (lines 24 and 25).

3 Summary

The SpecC Language Reference Manual defines the syntax and semantics of the SpecC language.

The SpecC language is designed to model embedded systems at system level. It is based on the ANSI-C programming language and uses additional constructs to support the requirements of modelling embedded systems. In Section 2 these additional constructs were enumerated and formally defined. In summary these constructs add support for modelling behavioral hierarchy, concurrency, state transitions, timing, and exception handling.

For further information on the SpecC language please refer to [3, 4, 5].

A SpecC Keywords and Tokens

In this section a complete list of the SpecC keywords and tokens is defined. The following subsections use the `lex` syntax as the formal notation.

A.1 Lexical Rules

The following lexical rules are used to make the definitions below more understandable.

```
delimiter           [ \t\b\r]
newline             [\n\f\v]
whitespace          {delimiter}+
ws                  {delimiter}*
ucletter            [A-Z]
lcletter            [a-z]
letter              ({ucletter}|{lcletter})
digit               [0-9]
bindigit            [01]
octdigit            [0-7]
hexdigit            [0-9a-fA-F]
identifier          (({letter}|"_")({letter}|{digit}|"_")*)
integer             {digit}+
binary              {bindigit}+
decinteger          [1-9]{digit}*
octinteger          "0"{octdigit}*
hexinteger          "0"[xX]{hexdigit}+
decinteger_u        {decinteger}[uU]
octinteger_u        {octinteger}[uU]
hexinteger_u        {hexinteger}[uU]
decinteger_l        {decinteger}[lL]
octinteger_l        {octinteger}[lL]
hexinteger_l        {hexinteger}[lL]
decinteger_ul       {decinteger}("ul"|"lu"|"uL"|"Lu"|"Ul"|"lU"|"UL"|"LU")
octinteger_ul       {octinteger}("ul"|"lu"|"uL"|"Lu"|"Ul"|"lU"|"UL"|"LU")
hexinteger_ul       {hexinteger}("ul"|"lu"|"uL"|"Lu"|"Ul"|"lU"|"UL"|"LU")
decinteger_ll       {decinteger}("ll"|"LL")
octinteger_ll       {octinteger}("ll"|"LL")
hexinteger_ll       {hexinteger}("ll"|"LL")
decinteger_ull      {decinteger}("ull"|"llu"|"uLL"|"LLu"|"Ull"|"llU"|"ULL"|"LLU")
octinteger_ull      {octinteger}("ull"|"llu"|"uLL"|"LLu"|"Ull"|"llU"|"ULL"|"LLU")
hexinteger_ull      {hexinteger}("ull"|"llu"|"uLL"|"LLu"|"Ull"|"llU"|"ULL"|"LLU")
octchar             "\\"{octdigit}{1,3}
hexchar             "\\x"{hexdigit}+
exponent            [eE][+-]?{integer}
fraction            {integer}
float1              {integer}"."{fraction}?({exponent})?
```

```
float2          "."{fraction}({exponent})?
float3          {integer}{exponent}
float           {float1}|{float2}|{float3}
float_f         {float}[fF]
float_l         {float}[lL]
bitvector       {binary}("b"|"B")
bitvector_u     {binary}("ub"|"bu"|"uB"|"Bu"|"Ub"|"bU"|"UB"|"BU")
cppstart        {ws}"#"{ws}
cppflag         {whitespace}[1-4]
cppfile         "\""[ !#-~]*"\""
cpplineno       ^{cppstart}{integer}{whitespace}{cppfile}{cppflag}*{ws}{newline}
cpppragma       ^{cppstart}"pragma"{ws}.*
cppdirective    ^{cppstart}.*
```

A.2 Comments

In addition to the standard C style comments the SpecC language also supports C++ style comments. Everything following two slash-characters is ignored until the end of the line.

```
"/*" <anything> "*/"     /* ignore comment */
"//" <anything> "\n"     /* ignore comment */
```

A.3 String and Character Constants

SpecC follows the standard C/C++ conventions for encoding character and string constants. The following escape sequences are recognized:

```
"\n"            /* newline              (0x0a) */
"\t"            /* tabulator            (0x09) */
"\v"            /* vertical tabulator   (0x0b) */
"\b"            /* backspace            (0x08) */
"\r"            /* carriage return      (0x0d) */
"\f"            /* form feed            (0x0c) */
"\a"            /* bell                 (0x07) */
{octchar}       /* octal encoded character */
{hexchar}       /* hexadecimal encoded character */
```

A.4 White space and Preprocessor Directives

As usual white space in the source code is ignored. Preprocessor directives are handled by the C preprocessor (`cpp`) and are therefore eliminated from the SpecC source code when it is read by the scanner. As a special case `pragma` directives which are still left after preprocessing are simply ignored.

```
{newline}        /* skip */
{whitespace}     /* skip */
{cpplineno}      /* acknowledge */
{cpppragma}      /* ignore */
{cppdirective}   /* error */
```

A.5 Keywords

The SpecC language recognizes the following ANSI-C keywords:

```
"auto"          { TOK_AUTO }
"break"         { TOK_BREAK }
"case"          { TOK_CASE }
"char"          { TOK_CHAR }
"const"         { TOK_CONST }
"continue"      { TOK_CONTINUE }
"default"       { TOK_DEFAULT }
"do"            { TOK_DO }
"double"        { TOK_DOUBLE }
"else"          { TOK_ELSE }
"enum"          { TOK_ENUM }
"extern"        { TOK_EXTERN }
"float"         { TOK_FLOAT }
"for"           { TOK_FOR }
"goto"          { TOK_GOTO }
"if"            { TOK_IF }
"int"           { TOK_INT }
"long"          { TOK_LONG }
"register"      { TOK_REGISTER }
"return"        { TOK_RETURN }
"short"         { TOK_SHORT }
"signed"        { TOK_SIGNED }
"sizeof"        { TOK_SIZEOF }
"static"        { TOK_STATIC }
"struct"        { TOK_STRUCT }
"switch"        { TOK_SWITCH }
"typedef"       { TOK_TYPEDEF }
"union"         { TOK_UNION }
"unsigned"      { TOK_UNSIGNED }
"void"          { TOK_VOID }
"volatile"      { TOK_VOLATILE }
"while"         { TOK_WHILE }
```

In addition the following SpecC keywords are recognized:

```
"behavior"      { TOK_BEHAVIOR }
"bit"           { TOK_BIT }
"bool"          { TOK_BOOL }
"callback"      { TOK_CALLBACK }
"channel"       { TOK_CHANNEL }
"delta"         { TOK_DELTA }
"event"         { TOK_EVENT }
"false"         { TOK_FALSE }
"fsm"           { TOK_FSM }
"implements"    { TOK_IMPLEMENTS }
"import"        { TOK_IMPORT }
"in"            { TOK_IN }
"inout"         { TOK_INOUT }
"interface"     { TOK_INTERFACE }
"interrupt"     { TOK_INTERRUPT }
"note"          { TOK_NOTE }
"notify"        { TOK_NOTIFY }
"notifyone"     { TOK_NOTIFYONE }
"out"           { TOK_OUT }
"par"           { TOK_PAR }
"pipe"          { TOK_PIPE }
"piped"         { TOK_PIPED }
"range"         { TOK_RANGE }
"timing"        { TOK_TIMING }
"trap"          { TOK_TRAP }
"true"          { TOK_TRUE }
"try"           { TOK_TRY }
"wait"          { TOK_WAIT }
"waitfor"       { TOK_WAITFOR }
```

SpecC supports all standard ANSI-C operators. The following multi-character operators are recognized as keywords:

```
"->"            { TOK_ARROW }
"++"            { TOK_INCR }
"--"            { TOK_DECR }
"<<"            { TOK_SHIFTLEFT }
">>"            { TOK_SHIFTRIGHT }
"<="            { TOK_LE }
">="            { TOK_GE }
"=="            { TOK_EQ }
"!="            { TOK_NE }
"&&"            { TOK_ANDAND }
"||"            { TOK_OROR }
"..."           { TOK_ELLIPSIS }
"*="            { TOK_MULTASSIGN }
"/="            { TOK_DIVASSIGN }
```

```
"%="            { TOK_MODASSIGN }
"+="            { TOK_PLUSASSIGN }
"-="            { TOK_MINUSASSIGN }
"<<="           { TOK_SLASSIGN }
">>="           { TOK_SRASSIGN }
"&="            { TOK_ANDASSIGN }
"^="            { TOK_EORASSIGN }
"|="            { TOK_ORASSIGN }
.               { <single character> }
```

A.6 Tokens with Values

The following is a complete list of all tokens that carry values.

```
{identifier}          { TOK_IDENTIFIER }
{decinteger}          { TOK_INTEGER }
{octinteger}          { TOK_INTEGER }
{hexinteger}          { TOK_INTEGER }
{decinteger_u}        { TOK_INTEGER }
{octinteger_u}        { TOK_INTEGER }
{hexinteger_u}        { TOK_INTEGER }
{decinteger_l}        { TOK_INTEGER }
{octinteger_l}        { TOK_INTEGER }
{hexinteger_l}        { TOK_INTEGER }
{decinteger_ul}       { TOK_INTEGER }
{octinteger_ul}       { TOK_INTEGER }
{hexinteger_ul}       { TOK_INTEGER }
{decinteger_ll}       { TOK_INTEGER }
{octinteger_ll}       { TOK_INTEGER }
{hexinteger_ll}       { TOK_INTEGER }
{decinteger_ull}      { TOK_INTEGER }
{octinteger_ull}      { TOK_INTEGER }
{hexinteger_ull}      { TOK_INTEGER }
{float}               { TOK_FLOATING }
{float_f}             { TOK_FLOATING }
{float_l}             { TOK_FLOATING }
{bitvector}           { TOK_BITVECTOR }
{bitvector_u}         { TOK_BITVECTOR }
<any_character>       { TOK_CHARACTER }
```

B The SpecC Grammar

This section contains the complete grammar of the SpecC language. For formal notation the `yacc` syntax style is used. Note that most of this grammar actually describes the C programming language[1]. Rules in the grammar added to support SpecC constructs are marked with comments.

B.1 Token with Values

```
identifier:                       /* Name */
        TOK_IDENTIFIER

typedef_name:                     /* Name */
        TOK_TYPEDEFNAME

behavior_name:                    /* Name */
        TOK_BEHAVIORNAME

channel_name:                     /* Name */
        TOK_CHANNELNAME

interface_name:                   /* Name */
        TOK_INTERFACENAME

integer:                          /* Const */
        TOK_INTEGER

floating:                         /* Const */
        TOK_FLOATING

character:                        /* Const */
        TOK_CHARACTER

string:                           /* String */
        TOK_STRING

bitvector:                        /* Const */
        TOK_BITVECTOR
```

B.2 Constants

```
constant:                         /* Expression */
```

[1]The SpecC grammar presented here is based on a ANSI-C grammar developed by J. A. Roskind. Use of that grammar is permitted if the following statement is preserved: "Portions Copyright (c) 1989, 1990 James A. Roskind".

```
        integer
        | floating
        | character
        /*** SpecC-only: boolean constants ***/
        | TOK_FALSE
        | TOK_TRUE
        /*** SpecC-only: bitvector constant ***/
        | bitvector

string_literal_list :           /* String */
        string
        | string_literal_list string
```

B.3 Expressions

```
primary_expression :            /* Expression */
        identifier
        | constant
        | string_literal_list
        | '(' comma_expression ')'
        /*** SpecC-only: untimed timing ***/
        | TOK_DELTA

postfix_expression :            /* Expression */
        primary_expression
        | postfix_expression '[' comma_expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' member_name
        | postfix_expression TOK_ARROW member_name
        | postfix_expression TOK_INCR
        | postfix_expression TOK_DECR
        /*** SpecC-only: bitvector slicing ***/
        | postfix_expression '[' comma_expression ':' comma_expression ']'

member_name:                    /* Name */
        identifier
        | typedef_name

argument_expression_list :      /* ExpressionList */
        assignment_expression
        | argument_expression_list ',' assignment_expression

unary_expression :              /* Expression */
        postfix_expression
        | TOK_INCR unary_expression
```

```
        | TOK_DECR unary_expression
        | unary_operator cast_expression
        | TOK_SIZEOF unary_expression
        | TOK_SIZEOF '(' type_name ')'

unary_operator:                 /* ExpressionType */
        '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'

cast_expression:                /* Expression */
        unary_expression
        | '(' type_name ')' cast_expression

        /*** SpecC-only: bitvector concatenation ***/
concat_expression:              /* Expression */
        cast_expression
        | concat_expression '@' cast_expression

multiplicative_expression:      /* Expression */
        concat_expression
        | multiplicative_expression '*' concat_expression
        | multiplicative_expression '/' concat_expression
        | multiplicative_expression '%' concat_expression

additive_expression:            /* Expression */
        multiplicative_expression
        | additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression

shift_expression:               /* Expression */
        additive_expression
        | shift_expression TOK_SHIFTLEFT additive_expression
        | shift_expression TOK_SHIFTRIGHT additive_expression

relational_expression:          /* Expression */
        shift_expression
        | relational_expression '<' shift_expression
        | relational_expression '>' shift_expression
        | relational_expression TOK_LE shift_expression
        | relational_expression TOK_GE shift_expression

equality_expression:            /* Expression */
        relational_expression
        | equality_expression TOK_EQ relational_expression
```

```
        | equality_expression TOK_NE relational_expression

and_expression:                     /* Expression */
        equality_expression
        | and_expression '&' equality_expression

exclusive_or_expression:            /* Expression */
        and_expression
        | exclusive_or_expression '^' and_expression

inclusive_or_expression:            /* Expression */
        exclusive_or_expression
        | inclusive_or_expression '|' exclusive_or_expression

logical_and_expression:             /* Expression */
        inclusive_or_expression
        | logical_and_expression TOK_ANDAND inclusive_or_expression

logical_or_expression:              /* Expression */
        logical_and_expression
        | logical_or_expression TOK_OROR logical_and_expression

conditional_expression:             /* Expression */
        logical_or_expression
        | logical_or_expression '?' comma_expression ':' conditional_expression

assignment_expression:              /* Expression */
        conditional_expression
        | unary_expression assignment_operator assignment_expression

assignment_operator:                /* ExpressionType */
        '='
        | TOK_MULTASSIGN
        | TOK_DIVASSIGN
        | TOK_MODASSIGN
        | TOK_PLUSASSIGN
        | TOK_MINUSASSIGN
        | TOK_SLASSIGN
        | TOK_SRASSIGN
        | TOK_ANDASSIGN
        | TOK_EORASSIGN
        | TOK_ORASSIGN

comma_expression:                   /* Expression */
        assignment_expression
        | comma_expression ',' assignment_expression

constant_expression:                /* Expression */
```

```
        conditional_expression

comma_expression_opt :              /* Expression */
        /* nothing */
        | comma_expression
```

B.4 Declarations

```
declaration :                       /* void */
        sue_declaration_specifier ';'
        | sue_type_specifier ';'
        | declaring_list ';'
        | default_declaring_list ';'

default_declaring_list :            /* DeclarationSpec */
        declaration_qualifier_list identifier_declarator initializer_opt
        | type_qualifier_list identifier_declarator initializer_opt
        | default_declaring_list ',' identifier_declarator initializer_opt

declaring_list :                    /* DeclarationSpec */
        declaration_specifier declarator initializer_opt
        | type_specifier declarator initializer_opt
        | declaring_list ',' declarator initializer_opt

declaration_specifier :             /* DeclarationSpec */
        basic_declaration_specifier
        | sue_declaration_specifier
        | typedef_declaration_specifier

type_specifier :                    /* Type */
        basic_type_specifier
        | sue_type_specifier
        | typedef_type_specifier

declaration_qualifier_list :        /* BasicType */
        storage_class
        | type_qualifier_list storage_class
        | declaration_qualifier_list declaration_qualifier

type_qualifier_list :               /* BasicType */
        type_qualifier
        | type_qualifier_list type_qualifier

declaration_qualifier :             /* BasicType */
        storage_class
        | type_qualifier
```

```
type_qualifier :                    /* BasicType */
        TOK_CONST
        | TOK_VOLATILE

basic_declaration_specifier :       /* BasicType */
        declaration_qualifier_list basic_type_name
        | basic_type_specifier storage_class
        | basic_declaration_specifier declaration_qualifier
        | basic_declaration_specifier basic_type_name

basic_type_specifier :              /* BasicType */
        basic_type_name
        | type_qualifier_list basic_type_name
        | basic_type_specifier type_qualifier
        | basic_type_specifier basic_type_name

sue_declaration_specifier :         /* DeclarationSpec */
        declaration_qualifier_list elaborated_type_name
        | sue_type_specifier storage_class
        | sue_declaration_specifier declaration_qualifier

sue_type_specifier :                /* Type */
        elaborated_type_name
        | type_qualifier_list elaborated_type_name
        | sue_type_specifier type_qualifier

typedef_declaration_specifier :     /* DeclarationSpec */
        typedef_type_specifier storage_class
        | declaration_qualifier_list typedef_name
        | typedef_declaration_specifier declaration_qualifier

typedef_type_specifier :            /* Type */
        typedef_name
        | type_qualifier_list typedef_name
        | typedef_type_specifier type_qualifier

storage_class :                     /* BasicType */
        TOK_TYPEDEF
        | TOK_EXTERN
        | TOK_STATIC
        | TOK_AUTO
        | TOK_REGISTER
        /*** SpecC-only: piped modifier ***/
        | TOK_PIPED

basic_type_name :                   /* BasicType */
        TOK_INT
```

```
        | TOK_CHAR
        | TOK_SHORT
        | TOK_LONG
        | TOK_FLOAT
        | TOK_DOUBLE
        | TOK_SIGNED
        | TOK_UNSIGNED
        | TOK_VOID
        /*** SpecC-only: boolean type ***/
        | TOK_BOOL
        /*** SpecC-only: bit(vector) type ***/
        | TOK_BIT '[' constant_expression ':' constant_expression ']'
        /*** SpecC-only: event type ***/
        | TOK_EVENT

elaborated_type_name:               /* Type */
        aggregate_name
        | enum_name

aggregate_name:                     /* Type */
        aggregate_key '{' member_declaration_list '}'
        | aggregate_key identifier_or_typedef_name '{' member_declaration_list
                '}'
        | aggregate_key identifier_or_typedef_name

aggregate_key:                      /* UserTypeClass */
        TOK_STRUCT
        | TOK_UNION

member_declaration_list:            /* MemberList */
        member_declaration
        | member_declaration_list member_declaration

member_declaration:                 /* MemberList */
        member_declaring_list ';'
        | member_default_declaring_list ';'
        /*** SpecC-only: note definition in member list ***/
        | note_definition

member_default_declaring_list:      /* MemberDeclSpec */
        type_qualifier_list member_identifier_declarator
        | member_default_declaring_list ',' member_identifier_declarator

member_declaring_list:              /* MemberDeclSpec */
        type_specifier member_declarator
        | member_declaring_list ',' member_declarator

member_declarator:                  /* MmbrDeclarator */
```

```
        declarator bit_field_size_opt
        | bit_field_size

member_identifier_declarator:   /* MmbrDeclarator */
        identifier_declarator bit_field_size_opt
        | bit_field_size

bit_field_size_opt:             /* Expression */
        /* nothing */
        | bit_field_size

bit_field_size:                 /* Expression */
        ':' constant_expression

enum_name:                      /* Type */
        TOK_ENUM '{' enumerator_list '}'
        | TOK_ENUM identifier_or_typedef_name '{' enumerator_list '}'
        | TOK_ENUM identifier_or_typedef_name

enumerator_list:                /* MemberList */
        identifier_or_typedef_name enumerator_value_opt
        | enumerator_list ',' identifier_or_typedef_name enumerator_value_opt

enumerator_value_opt:           /* Expression */
        /* nothing */
        | '=' constant_expression

parameter_type_list:            /* ParameterList */
        parameter_list
        | parameter_list ',' TOK_ELLIPSIS

parameter_list:                 /* ParameterList */
        parameter_declaration
        | parameter_list ',' parameter_declaration

parameter_declaration:          /* Parameter */
        declaration_specifier
        | declaration_specifier abstract_declarator
        | declaration_specifier identifier_declarator
        | declaration_specifier parameter_typedef_declarator
        | declaration_qualifier_list
        | declaration_qualifier_list abstract_declarator
        | declaration_qualifier_list identifier_declarator
        | type_specifier
        | type_specifier abstract_declarator
        | type_specifier identifier_declarator
        | type_specifier parameter_typedef_declarator
        | type_qualifier_list
```

```
        | type_qualifier_list  abstract_declarator
        | type_qualifier_list  identifier_declarator

identifier_or_typedef_name :        /* Name */
        identifier
        | typedef_name

type_name:                          /* Type */
        type_specifier
        | type_specifier  abstract_declarator
        | type_qualifier_list
        | type_qualifier_list  abstract_declarator

initializer_opt :                   /* Initializer */
        /* nothing */
        | '=' initializer

initializer :                       /* Initializer */
        '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        | constant_expression

initializer_list :                  /* InitializerList */
        initializer
        | initializer_list ',' initializer
```

B.5 Statements

```
statement:                          /* Statement */
        labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        /*** SpecC-only: SpecC statements ***/
        | spec_c_statement

labeled_statement:                  /* Statement */
        any_name ':' statement
        | TOK_CASE constant_expression ':' statement
        | TOK_DEFAULT ':' statement

compound_statement:                 /* Statement */
        compound_scope '{' '}'
        | compound_scope '{' declaration_list '}'
```

```
        | compound_scope '{' statement_list '}'
        | compound_scope '{' declaration_list statement_list '}'

compound_scope:                    /* Scope */
        /* nothing */

declaration_list :                 /* void */
        declaration
        | declaration_list declaration
        /*** SpecC-only: note definitions in compound statements ***/
        | note_definition
        | declaration_list note_definition

statement_list :                   /* StatementList */
        statement
        | statement_list statement
        /*** SpecC-only: note definitions in compound statements ***/
        | statement_list note_definition

expression_statement :             /* Statement */
        comma_expression_opt ';'

selection_statement :              /* Statement */
          TOK_IF '(' comma_expression ')' statement
        | TOK_IF '(' comma_expression ')' statement TOK_ELSE statement
        | TOK_SWITCH '(' comma_expression ')' statement

iteration_statement :              /* Statement */
        TOK_WHILE '(' comma_expression_opt ')' statement
        | TOK_DO statement TOK_WHILE '(' comma_expression ')' ';'
        | TOK_FOR '(' comma_expression_opt ';' comma_expression_opt ';'
                comma_expression_opt ')' statement

jump_statement:                    /* Statement */
        TOK_GOTO any_name ';'
        | TOK_CONTINUE ';'
        | TOK_BREAK ';'
        | TOK_RETURN comma_expression_opt ';'
```

B.6 External Definitions

```
translation_unit :                 /* void */
        external_definition
        | translation_unit external_definition

external_definition :              /* void */
```

```
        function_definition
        | declaration
        /*** SpecC-only: SpecC specific definitions ***/
        | spec_c_definition

function_definition:            /* void */
        identifier_declarator compound_statement
        | declaration_specifier identifier_declarator compound_statement
        | type_specifier identifier_declarator compound_statement
        | declaration_qualifier_list identifier_declarator compound_statement
        | type_qualifier_list identifier_declarator compound_statement

declarator:                     /* Declarator */
        identifier_declarator
        | typedef_declarator

typedef_declarator:             /* Declarator */
        paren_typedef_declarator
        | parameter_typedef_declarator

parameter_typedef_declarator:   /* Declarator */
        typedef_name
        | typedef_name postfixing_abstract_declarator
        | clean_typedef_declarator

clean_typedef_declarator:       /* Declarator */
        clean_postfix_typedef_declarator
        | '*' parameter_typedef_declarator
        | '*' type_qualifier_list parameter_typedef_declarator

clean_postfix_typedef_declarator:       /* Declarator */
        '(' clean_typedef_declarator ')'
        | '(' clean_typedef_declarator ')' postfixing_abstract_declarator

paren_typedef_declarator:       /* Declarator */
        paren_postfix_typedef_declarator
        | '*' '(' simple_paren_typedef_declarator ')'
        | '*' type_qualifier_list '(' simple_paren_typedef_declarator ')'
        | '*' paren_typedef_declarator
        | '*' type_qualifier_list paren_typedef_declarator

paren_postfix_typedef_declarator:       /* Declarator */
        '(' paren_typedef_declarator ')'
        | '(' simple_paren_typedef_declarator postfixing_abstract_declarator ')'
        | '(' paren_typedef_declarator ')' postfixing_abstract_declarator

simple_paren_typedef_declarator:        /* Declarator */
        typedef_name
```

```
        | '(' simple_paren_typedef_declarator ')'

identifier_declarator:              /* Declarator */
        unary_identifier_declarator
        | paren_identifier_declarator

unary_identifier_declarator:        /* Declarator */
        postfix_identifier_declarator
        | '*' identifier_declarator
        | '*' type_qualifier_list identifier_declarator

postfix_identifier_declarator:      /* Declarator */
        paren_identifier_declarator postfixing_abstract_declarator
        | '(' unary_identifier_declarator ')'
        | '(' unary_identifier_declarator ')' postfixing_abstract_declarator

paren_identifier_declarator:        /* Declarator */
        identifier
        | '(' paren_identifier_declarator ')'

abstract_declarator:                /* AbstrDeclarator */
        unary_abstract_declarator
        | postfix_abstract_declarator
        | postfixing_abstract_declarator

postfixing_abstract_declarator:     /* AbstrDeclarator */
        array_abstract_declarator
        | '(' ')'
        | '(' parameter_type_list ')'

array_abstract_declarator:          /* AbstrDeclarator */
        '[' ']'
        | '[' constant_expression ']'
        | array_abstract_declarator '[' constant_expression ']'

unary_abstract_declarator:          /* AbstrDeclarator */
        '*'
        | '*' type_qualifier_list
        | '*' abstract_declarator
        | '*' type_qualifier_list abstract_declarator

postfix_abstract_declarator:        /* AbstrDeclarator */
        '(' unary_abstract_declarator ')'
        | '(' postfix_abstract_declarator ')'
        | '(' postfixing_abstract_declarator ')'
        | '(' unary_abstract_declarator ')' postfixing_abstract_declarator
```

B.7 SpecC Constructs

```
spec_c_definition :                /* void */
        import_definition
        | behavior_declaration
        | behavior_definition
        | channel_declaration
        | channel_definition
        | interface_declaration
        | interface_definition
        | note_definition

import_definition :                /* void */
        TOK_IMPORT string_literal_list ';'

behavior_declaration :             /* void */
        behavior_specifier port_list_opt ';'

behavior_definition :              /* void */
        behavior_specifier port_list_opt '{' internal_definition_list_opt
                '}' ';'

behavior_specifier :               /* Declarator */
        TOK_BEHAVIOR identifier

channel_declaration :              /* void */
        channel_specifier port_list_opt implements_interface_opt ';'

channel_definition :               /* void */
        channel_specifier port_list_opt implements_interface_opt
                '{' internal_definition_list_opt '}' ';'

channel_specifier :                /* Declarator */
        TOK_CHANNEL identifier

port_list_opt :                    /* ParameterList */
        /* nothing */
        | '(' ')'
        | '(' port_list ')'

port_list :                        /* ParameterList */
        port_declaration
        | port_list ',' port_declaration

port_declaration :                 /* Parameter */
        port_direction parameter_declaration
        | interface_name
```

```
        | interface_name identifier
        | channel_name
        | channel_name identifier

port_direction :                    /* Direction */
        /* nothing */
        | TOK_IN
        | TOK_OUT
        | TOK_INOUT

implements_interface_opt :          /* SymbolPtrList */
        /* nothing */
        | TOK_IMPLEMENTS interface_list

interface_list :                    /* SymbolPtrList */
        interface_name
        | interface_list ',' interface_name

internal_definition_list_opt :      /* void */
        /* nothing */
        | internal_definition_list

internal_definition_list :          /* void */
        internal_definition
        | internal_definition_list internal_definition

internal_definition :               /* void */
        function_definition
        | declaration
        | instantiation
        | note_definition

instantiation :                     /* void */
        instance_declaring_list ';'

instance_declaring_list :           /* DeclarationSpec */
        behavior_or_channel instance_declarator
        | instance_declaring_list ',' instance_declarator

instance_declarator :               /* Declarator */
        identifier port_mapping_opt

behavior_or_channel :               /* DeclarationSpec */
        behavior_name
        | channel_name

port_mapping_opt :                  /* ParameterList */
        /* nothing */
```

```
        | '(' ')'
        | '(' port_mapping_list ')'

port_mapping_list :                /* ParameterList */
        port_mapping
        | port_mapping_list ',' port_mapping

port_mapping :                     /* Parameter */
        identifier

interface_declaration :            /* void */
        interface_specifier ';'

interface_definition :             /* void */
        interface_specifier '{' internal_declaration_list_opt '}' ';'

interface_specifier :              /* Declarator */
        TOK_INTERFACE identifier

internal_declaration_list_opt :    /* void */
        /* nothing */
        | internal_declaration_list

internal_declaration_list :        /* void */
        internal_declaration
        | internal_declaration_list internal_declaration

internal_declaration :             /* void */
        declaration
        | TOK_CALLBACK declaration
        | note_definition

note_definition :                  /* void */
        TOK_NOTE any_name '=' note ';'
        | TOK_NOTE any_name '.' any_name '=' note ';'

note:                              /* Constant */
        constant_expression

any_name:                          /* Name */
        identifier
        | typedef_name
        | behavior_name
        | channel_name
        | interface_name
```

B.8 SpecC Statements

```
spec_c_statement :                  /* Statement */
        concurrent_statement
        | fsm_statement
        | exception_statement
        | timing_statement
        | wait_statement
        | waitfor_statement
        | notify_statement

concurrent_statement :              /* Statement */
        TOK_PAR compound_statement
        | TOK_PIPE compound_statement

fsm_statement :                     /* Statement */
        TOK_FSM '{' '}'
        | TOK_FSM '{' transition_list '}'

transition_list :                   /* TransitionList */
        transition
        | transition_list transition

transition :                        /* Transition */
        identifier ':'
        | identifier ':' cond_branch_list
        | identifier ':' '{' '}'
        | identifier ':' '{' cond_branch_list '}'

cond_branch_list :                  /* TransitionList */
        cond_branch
        | cond_branch_list cond_branch

cond_branch :                       /* Transition */
        TOK_IF '(' comma_expression ')' TOK_GOTO identifier ';'
        | TOK_GOTO identifier ';'
        | TOK_IF '(' comma_expression ')' TOK_BREAK ';'
        | TOK_BREAK ';'

exception_statement :               /* Statement */
        TOK_TRY compound_statement exception_list

exception_list :                    /* ExceptionList */
        exception
        | exception_list exception

exception :                         /* Exception */
```

```
        TOK_TRAP paren_event_list compound_statement
        | TOK_INTERRUPT paren_event_list compound_statement

paren_event_list :                  /* SymbolPtrList */
        event_list
        | '(' event_list ')'

event_list :                        /* SymbolPtrList */
        event
        | event_list ',' event

event :                             /* SymbolPtr */
        identifier

timing_statement :                  /* Statement */
        TOK_DO statement TOK_TIMING '{' constraint_list '}'

constraint_list :                   /* ConstraintList */
        constraint
        | constraint_list constraint

constraint :                        /* Constraint */
        TOK_RANGE '(' any_name ';' any_name ';' time_opt ';' time_opt ')' ';'

time_opt :                          /* Constant */
        /* nothing */
        | time

time :                              /* Expression */
        constant_expression

wait_statement :                    /* Statement */
        TOK_WAIT paren_event_list ';'

waitfor_statement :                 /* Statement */
        TOK_WAITFOR time ';'

notify_statement :                  /* Statement */
        TOK_NOTIFY paren_event_list ';'
        | TOK_NOTIFYONE paren_event_list ';'
```

C References

[1] X3 Secretariat. *The C Language.* X3J11/90-013, ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association, Washington, DC, USA, 1990.

[2] M. A. Ellis, B. Stroustrup. *The annotated C++ Reference Manual.* Addison-Wesley, Reading, Mass., 1990.

[3] D. Gajski, J. Zhu, R. Dömer. *The SpecC+ Language.* University of California, Irvine, Technical Report ICS-TR-97-15, April 15, 1997.

[4] J. Zhu, R. Dömer, D. Gajski. *Syntax and Semantics of the SpecC+ Language.* University of California, Irvine, Technical Report ICS-TR-97-16, April 1997.

[5] D. Gajski, J. Zhu, R. Dömer. *Essential Issues in Codesign.* University of California, Irvine, Technical Report ICS-TR-97-26, June 1997.

[6] J. Zhu, R. Dömer, D. Gajski. *Syntax and Semantics of the SpecC Language.* Proceedings of the Synthesis and System Integration of Mixed Technologies 1997, Osaka, Japan, December 1997.

[7] D. Gajski, G. Aggarwal, E.-S. Chang, R. Dömer, T. Ishii, J. Kleinsmith, J. Zhu. *Methodology for Design of Embedded Systems.* University of California, Irvine, Technical Report ICS-TR-98-07, March 1998.

Index