SpecC Language Reference Manual

Version 2.0

Approved December 12, 2002

SpecC Technology Open Consortium

`www.specc.org`

Authors:

Rainer Dömer
Andreas Gerstlauer
Daniel Gajski

Contents

Abstract

This SpecC language reference manual (LRM), version 2.0, defines the syntax and the semantics of the SpecC language 2.0. This document is based on the SpecC LRM version 1.0, dated March 6, 2001. It has been modified and extended according to the results of the work done by the SpecC language specification working group (LS-WG) of the SpecC Technology Open Consortium (STOC).

The SpecC language is defined as extension of the ANSI-C programming language. This document describes the syntax and semantics of the SpecC constructs that were added to the ANSI-C language.

For each SpecC construct, its purpose, its syntax, and its semantics are defined. In addition, each SpecC construct is illustrated by an example. The SpecC execution semantics are formally defined by use of a time interval formalism and an abstract simulation algorithm.

In the appendix, the complete SpecC grammar is included by use of an extended Backus-Naur form (EBNF), and the contents of the SpecC standard library are defined.

Chapter 1

Introduction

The SpecC language is a formal notation intended for the specification and design of digital embedded systems, including hardware and software portions. Built on top of the ANSI-C programming language, the SpecC language supports concepts essential for embedded systems design, including behavioral and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling, and timing.

This document defines the syntax and the semantics of the SpecC language, version 2.0. This version 2.0 is based on the SpecC LRM version 1.0 [7]. It incorporates the results of the language specification working group (LS-WG) established by the SpecC Technology Open Consortium (STOC).

Since the SpecC language is a true superset of the ANSI-C programming language, this document only covers the language constructs not found in ANSI-C. For detailed information about the syntax and semantics of ANSI-C, please refer to the ISO Standard ISO/IEC 9899 [1].

Chapter 2 defines the foundation, the types, the classes, the statements, and other constructs of the SpecC language. Chapter 3 then defines the execution semantics of the SpecC language by use of a formal notation called time interval formalism. In addition, an abstract simulation algorithm for SpecC program execution is given.

In Appendix A, the complete grammar of the SpecC language is included. Finally, Appendix B defines the contents of the SpecC standard library.

1.1 Brief history of the SpecC language

The first version of the SpecC language was developed in 1997 at the University of California, Irvine (UCI) [4]. While many concepts supported by the SpecC language were new at that time, some concepts were based on previous research, for example, the SpecCharts[2, 3] language.

In the following years, research on system design with the SpecC language was intensified at UCI and early tools including a SpecC compiler and a simulator were implemented. Highlights of this research have been published in the first book on SpecC, "SpecC: Specification Language and Methodology" [5], in 1999.

At the same time, the SpecC language gained world-wide acceptance in industry, reaching a major milestone in the SpecC history, the foundation of the SpecC Technology Open Consortium (STOC) in 1999 [10]. STOC was founded with the goal of promoting the SpecC idea by standardizing the SpecC language and establishing design guidelines, industry collaboration and interoperability among design tools, based on SpecC.

Since the foundation of STOC, a second book on SpecC, entitled "System Design: A Practical Guide with SpecC" [8], was published and the SpecC technology advanced further, driven by industrial and academic work in general, and by the formation of two STOC working groups in particular, namely the case study (CS-WG) and language specification working groups (LS-WG). As a result of the latter, the SpecC language was refined and extended, leading to its second generation, SpecC 2.0.

This document defines the version 2.0 of the SpecC language standard approved by STOC.

1.2 Contributors

This document is the result of the work of the language specification working group (LS-WG) of STOC. The authors wish to thank all active members of this working group for their fruitful discussions, their valuable contributions, and helpful suggestions.

At the time of approval of SpecC 2.0, the SpecC LS-WG consists of the following members, listed in alphabetical order:

Yamada Akihiko, Dai Araki, Przemyslaw Bakowski, Ken-ichi Chiboshi, Rainer Doe-

mer, Takashi Eda, Hans Eveking, Masahiro Fujita, Hiroshi Fukutomi, Daniel Gajski, Rajesh Gupta, Tedd Hadley, Roger Hale, Alan Hu, Masato Igarashi, Masaki Ito, Steven Johnson, Yamashiro Kenji, Tsuneo Kinoshita, Srivas Mandayam, George Milne, Hiroshi Nakamura, Mike Olivarez, Alex Orailoglu, Sreeranga Rajan, Yoshisato Sakai, Thanyapat Sakunkonchak, Komatsu Satoshi, Yamaguchi Suguru, Tanimoto Tadaaki, Ishii Tadatoshi, Hiroaki Takada, Iitsuka Takayoshi, Shinsuke Tamura, Shibashita Tetsu, Hiroyuki Tomiyama, Nakamura Toshihiko, Eugenio Villar, Wayne Wolf, Kodama Yuetsu.

Chapter 2

SpecC Language

2.1 Foundation

The SpecC language is based on the ANSI-C programming language as defined in ISO Standard ISO/IEC 9899 [1].

Unless specified otherwise in this document, the syntax and semantic rules specified for ANSI-C are also valid for SpecC. Also, the SpecC constructs described in this document are designed as straightforward extensions, to which the usual ANSI-C semantics are applied, whenever possible.

2.1.1 Array assignment

In contrast to ANSI-C, the SpecC language allows the assignment of variables of array type. Syntactically, such array assignment is specified in the same manner as basic variables are assigned.

The assignment of a whole array is equivalent to the assignment of every element in the source array to the element with the same index (or indices in case of multi-dimensional arrays) in the target array.

For array assignments, the target and source arrays must have the same type and the same dimensions. As the result of an array assignment, the target array will have the same contents as the source array.

The result type of an array assignment operation is **void**. This is in contrast to standard

assignment operations where the result type is given by the type of the left argument. As a consequence, an array assignment operation may not be used as a subexpression in another expression (but only as an expression statement).

Example:

```
1  int     a[10],
2          b[10];
3  double  c[3][3],
4          d[3][3],
5          e[3][3];
6
7  void f(void)
8  {
9     a = b;       // array assignment
10    c = d;       // array assignment
11    c[2] = d[1]; // sub-array assignment
12 // c = (d = e); // illegal!
13 }
```

2.1.2 Variable initialization

In contrast to ANSI-C, the SpecC language initializes every variable that is statically declared in the SpecC description. Unless a **static** variable has an explicit initializer specified by the user, the variable is implicitly initialized with zero (while it would be uninitialized in ANSI-C).

Variables defined in global scope without storage class specification are considered **static**, as are variables defined in class scope.

Example:

```
1  int     i = 0,  // explicitly initialized to 0
2          i2;     // implicitly initialized to 0
3  char    c;      // implicitly initialized to '\000'
4  float   f;      // implicitly initialized to 0.0f
5  void    *p;     // implicitly initialized to 0 (NULL)
6  long    l[2];   // implicitly initialized to {0l,0l}
7
8  void fct(void)
```

```
 9 {
10    int           x;      // uninitialized
11    static int    y;      // initialized to 0
12
13    // ...
14 }
15
16 behavior B
17 {
18    bool          b;      // initialized to false
19    double        d;      // initialized to 0.0
20
21    void main(void)
22    {
23       // ...
24    }
25 };
```

2.2 SpecC types

2.2.1 Boolean type

Purpose: Explicit representation of Boolean values

Synopsis:

```
basic_type_name =
        ...
        | bool
        ...

constant =
        ...
        | false
        | true
        ...
```

Semantics:

(a) A Boolean value, of type **bool**, has one of two values: **true** or **false**.

(b) A Boolean value can be used to hold the result of logical and relational operations (e. g. !, &&, <, >, ==, etc.).

(c) If converted (implicitly or explicitly) to an integer type, **true** becomes 1 and **false** becomes 0.

(d) A Boolean type cannot be **signed** or **unsigned**.

Example:

```
1 bool f(bool b1, int a)
2 {
3    bool b2;
4
5    if (b1 == true)
6       { b2 = b1 || (a > 0);
7       }
```

```
 8    else
 9       { b2 = !b1;
10       }
11    return(b2);
12 }
```

Notes:

i. The type **bool** in SpecC is equivalent to the type **bool** in C++.

2.2.2 Long long type

Purpose: Representation of very large integer values

Synopsis:

```
decinteger_ll   {decinteger}[lL][lL]
octinteger_ll   {octinteger}[lL][lL]
hexinteger_ll   {hexinteger}[lL][lL]
decinteger_ull  {decinteger}([uU][lL][lL]|[lL][lL][uU])
octinteger_ull  {octinteger}([uU][lL][lL]|[lL][lL][uU])
hexinteger_ull  {hexinteger}([uU][lL][lL]|[lL][lL][uU])

basic_type_name =
        int
        | long
        | ...
        | signed
        | unsigned
        ...

basic_type_specifier =
        basic_type_name
        | basic_type_specifier basic_type_name
        ...

constant =
        ...
        | integer
```

Semantics:

(a) An integer literal of type **signed long long int** is specified with a suffix *ll*, where the suffix is case-insensitive.

(b) An integer literal of type **unsigned long long int** is specified with a suffix *ull* or *llu*, where the suffix is case-insensitive.

(c) The **long long int** type is an integral data type for very large values. The number of bits of the representation is equal to or higher than the number of bits of the representation of the **long int** data type.

(d) The **long long int** type can be **signed** or **unsigned**.

(e) The usual type promotion and type conversion rules apply.

Example:

```
1  bool              Boolean;     //  1 bit
2                                 // [false=0, true=1]
3  char              Character;   //  8 bit, signed
4                                 // [-128, 127]
5  unsigned char     UCharacter;  //  8 bit, unsigned
6                                 // [0, 255]
7  short             Short;       // 16 bit, signed
8                                 // [-32768, 32767]
9  unsigned short    UShort;      // 16 bit, unsigned
10                                // [0, 65535]
11 int               Integer;     // 32 bit, signed
12                                // [-2147483648, 2147483647]
13 unsigned int      UInteger;    // 32 bit, unsigned
14                                // [0, 4294967295]
15 long              Long;        // 32 bit, signed
16                                // [-2147483648, 2147483647]
17 unsigned long     ULong;       // 32 bit, unsigned
18                                // [0, 4294967295]
19 long long         LongLong;    // 64 bit, signed
20                                // [-9223372036854775808,
21                                //   9223372036854775807]
22 unsigned long long ULongLong;// 64 bit, unsigned
23                                // [0, 18446744073709551615]
```

Notes:

i. The example shows the standard integral types of the SpecC language and their typical storage sizes and value ranges.

2.2.3 Bit vector type

Purpose: Representation of bit vectors of arbitrary length

Synopsis:

```
bindigit        [01]
binary          {bindigit}+
bitvector       {binary}[bB]
bitvector_u     {binary}([uU][bB]|[bB][uU])

basic_type_name =
        ...
        | signed
        | unsigned
        ...
        | bit '[' constant_expression ':' constant_expression ']'
        | bit '[' constant_expression ']'
        ...

constant =
        ...
        | bitvector
        | bitvector_u
        ...

postfix_expression =
        ...
        | postfix_expression '[' comma_expression ']'
        | postfix_expression '[' constant_expression ':'
              constant_expression ']'

concat_expression =
        cast_expression
        | concat_expression '@' cast_expression
```

Semantics:

(a) A bit vector **bit**$[l:r]$ represents an integral data type of arbitrary bit length. The length of a bit vector is determined by its left and right bounds, as follows: $length(bv) = abs(left(bv) - right(bv) + 1)$.

(b) In arithmetic operations, the bit indicated by the left bound represents the most-significant bit (MSB), whereas the bit indicated by the right bound represents the least-significant bit (LSB). Except for the bit slice and bit access operations (see below), a bit vector is always normalized to the bounds $[length-1:0]$ before any operation is performed with it. Also, the result type of a bit vector operation is always normalized to $[length-1:0]$, where *length* is determined by the operand with the greatest bit vector length.

(c) As a short-cut, the type **bit**$[length]$ is equivalent to **bit**$[l:r]$, where $l = length-1$ and $r = 0$.

(d) The left and right bounds, l and r, of a bit vector are specified at the time of declaration and must be constant expressions which can be evaluated to constants at compile time. The same applies to the *length* specifier for the short declaration.

(e) A bit vector is either **signed** or **unsigned**.

(f) A bit vector can be used as any other integral type in expressions and the usual conversion and promotion rules apply. (For example, type **int** is equivalent to type **bit**$[sizeof(\textbf{int})*8-1:0]$.)

(g) Implicit promotion from (**unsigned**) **int**, (**unsigned**) **long**, or (**unsigned**) **long long** to bit vector is performed when necessary. Hereby, the resulting bit vector length is determined from the number of bits of the source type (which is implementation dependent).

(h) Automatic conversion, such as **signed**/**unsigned** extension or truncation, is supported for bit vectors as with any other integral type.

(i) Bit vector constants are noted as a sequence of zeros and ones immediately followed by a suffix *b* or *ub* indicating the **signed** or **unsigned** bit vector type, respectively. The suffix is case-insensitive.

(j) In addition to the standard C operations, a concatenation operation, noted as @, and a slicing operation, noted as $[lb:rb]$, are available in SpecC (see lines 11 and 13 in the

example). Both operations can be applied to bit vectors as well as to any other integral type (which will then be implicitly converted to a bit vector of suitable length).

(k) The binary operator @, applied to two bit vectors *a* (left argument) and *b* (right argument), results in the concatenation of *a* and *b*. The result type is a bit vector with bounds $[length(a) + length(b) - 1 : 0]$.

(l) The unary postfix operator $[lb : rb]$, applied to a bit vector *a*, results in a bit slice of *a* where the left-most bit is specified by index *lb* and the right-most bit is specified by index *rb*. *lb* and *rb* must be constant expressions which can be evaluated to constants at compile time. The result type is a bit vector with bounds $[abs(lb - rb) + 1 : 0]$.

(m) The unary bit access operator $[b]$ (similar to the array access operator) is available as a short-hand for accessing a single bit $[b : b]$ of a bit vector. The result type of this operation is **unsigned bit**$[0 : 0]$.

Example:

```
 1 typedef bit[3:0]        nibble_type;
 2 nibble_type             a;
 3 unsigned bit[15:0]      c;
 4
 5 void f(nibble_type b, bit[16:1] d)
 6 {
 7    a = 1101B;                    // bitvector assignment
 8    c = 1110001111100011ub;
 9    c[7:4] = a;                   // bitslice assignment
10
11    b = c[2:5];                   // bitvector slicing
12    c[0] = c[16];                 // single bit access
13    d = a @ b @ c[0:15];          // bitvector concatenation
14    b += 42 + a * 12;             // arithmetic operations
15    d = ~(b | 10101010B);         // logic operations
16 }
```

Notes:

i. A bit vector can be thought of as a parameterized type whose bounds are defined with the name of the type.

ii. Bit vector bounds and bounds of bit vector slices are required to be constants at compile time. As a result, the length of any bitvector expression is always known at compile time. This enables an efficient implementation of bit vectors.

iii. Note that the index of the bit access operator is *not* required to be a constant. A single bit access always yields the bounds $[0:0]$, so there is no need for a constant index.

iv. Since bit vectors are fully integrated into the integral data types, there is typically no need for explicit type casting in any operations involving bitvectors. Thus, bit vectors may be used just as integers.

v. Note that, depending on the given bounds, bit slicing can reverse the order of the bits in the result.

vi. Special port mapping rules apply to ports of bitvector type, see Section 2.3.5.

2.2.4 Long double type

Purpose: Representation of high-precision floating point values

Synopsis:

```
digit           [0–9]
integer         {digit}+
exponent        [eE][+–]?{integer}
fraction        {integer}
float1          {integer}"."{fraction}?({exponent})?
float2          "."{fraction}({exponent})?
float3          {integer}{exponent}
floating        {float1}|{float2}|{float3}
float_f         {floating}[fF]
float_l         {floating}[lL]

basic_type_name =
        ...
        | long
        | double
        ...

basic_type_specifier =
        basic_type_name
        | basic_type_specifier basic_type_name
        ...

constant =
        ...
        | floating
        ...
```

Semantics:

(a) A floating point literal can be attached the suffix *l*, specifying it as type **long double**. The suffix is case-insensitive.

(b) The **long double** data type is a floating point type with a high-precision representation. Its precision is equal to or higher than the precision of the **double** data type.

(c) The usual promotion and conversion rules apply.

Example:

```
1 float              Float;       // 32 bit
2 double             Double;      // 64 bit
3 long double        LongDouble;  // 96 bit
```

Notes:

i. The example shows the standard floating point types of the SpecC language and their typical storage sizes.

ii. The type **long double** in SpecC is equivalent to the type **long double** in C++.

2.2.5 Event type

Purpose: Basic mechanism for synchronization and exception handling

Synopsis:

```
basic_type_name =
        ...
        | event

wait_statement =
        wait paren_event_list ';'
        | wait paren_and_event_list ';'

notify_statement =
        notify paren_event_list ';'
        | notifyone paren_event_list ';'

exception =
        trap paren_event_list compound_statement
        | interrupt paren_event_list compound_statement

paren_event_list =
        event_list
        | '(' event_list ')'

event_list =
        event_identifier
        | event_list ',' event_identifier
        | event_list '||' event_identifier

paren_and_event_list =
        and_event_list
        | '(' and_event_list ')'

and_event_list =
        event_identifier '&&' event_identifier
        | and_event_list '&&' event_identifier

clock_specifier =
        event_list
        | ...

sensitivity_list_opt =
```

```
<nothing>
| event_list
```

Semantics:

(a) The **event** type is a special type that enables SpecC to support exception handling and synchronization of concurrently executing behaviors.

(b) The **event** type must not be combined with any other type, type modifier or type qualifier. It must not be used as a member of any composite or aggregate type.

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

(d) Events can only be used with the **wait**, **notify** and **notifyone** statements (see the example and Section 2.4.6), with the **try-trap-interrupt** statement (see Section 2.4.7), with the **buffered** type (see Section 2.2.7), or with the **fsmd** statement (see Section 2.4.5).

Example:

```
 1 int     d;
 2 event   e;
 3
 4 void send(int x)
 5 {
 6    d = x;
 7    notify e;
 8 }
 9
10 int receive(void)
11 {
12    wait e;
13    return(d);
14 }
```

Notes:

i. The example shows a very primitive communication scheme synchronized by the use of an event *e*.

ii. Since an **event** does not have any value, it cannot transport any message. If a message needs to be sent together with an **event**, a **signal** (see Section 2.2.6) or **channel** (see Section 2.3.2) is probably the better choice for modeling the communication.

2.2.6 Signal type

Purpose: Representation of busses and wires

Synopsis:

```
default_declaring_list =
        ...
        | signal_class declaration_qualifier_list identifier_declarator
              initializer_opt
        | signal_class type_qualifier_list identifier_declarator
              initializer_opt

declaring_list =
        ...
        | signal_class declaration_specifier declarator initializer_opt
        | signal_class type_specifier declarator initializer_opt

port_declaration =
        port_direction signal_class_opt parameter_declaration
        | ...

signal_class_opt =
        <nothing>
        | signal_class

signal_class =
        signal
        | ...

event_identifier =
        identifier
        | edge_selector identifier
        | identifier edge_selector

edge_selector =
        rising
        | falling
```

Semantics:

(a) The **signal** type class defines an implicit composite type for the representation of wires or busses between concurrent behaviors.

(b) The **signal** type class can be specified for ordinary or composite types. It must not be specified for an **event** type or a **buffered** type class.

(c) The **signal** type class must not be used as a member of any composite or aggregate type.

(d) A variable of **signal** type is implicitly composed of an event, a current and a new value.

(e) A signal can be used wherever an event is expected. In this case, the event of the signal is accessed implicitly.

(f) An event carried by a signal can be filtered by use of either the **rising** or **falling** operator. A **rising** or **falling** operator must not be applied to a pure event.

(g) The **rising** operator filters out all events that do not represent a rising edge of the signal. A rising edge of the signal is defined as the signal value changing from a zero (current) value to a non-zero (new) value.

(h) The **falling** operator filters out all events that do not represent a falling edge of the signal. A falling edge of the signal is defined as the signal value changing from a non-zero (current) value to a zero (new) value.

(i) The **rising** or **falling** operator must only be used with read accesses to the event of a signal. These operators must not be used with the **notify** or **notifyone** statements.

(j) A signal can be used in an expression. For every read access, implicitly the current value of the signal is read. For every write access, implicitly the new value of the signal is written.

(k) The event of a signal is implicitly notified with every write access to the signal.

(l) At the time the event of a signal is delivered, the current value of the signal is updated with the new value.

Example:

```
 1 behavior B1(in  signal bit[16] a,
 2             in  signal bit[16] b,
 3             out signal bit[16] c)
 4 {
 5    void main(void)
 6    {
 7       while(true)
 8       {
 9          wait a, b;
10          c = a + b;
11       }
12    }
13 };
14
15 behavior B2(in  signal bit[1] CLK,
16             in  signal bit[16] a,
17             in  signal bit[16] b,
18             out signal bit[16] c)
19 {
20    void main(void)
21    {
22       while(true)
23       {
24          wait CLK rising;
25          c = a + b;
26       }
27    }
28 };
```

Notes:

i. A variable of type class **signal** can be viewed as a composite variable which contains an event, a current value and a new value. Whenever the signal is used in place of an event, such as with the **wait** or **notify** statements, the event of the signal is accessed. On the other hand, if the signal is accessed in any expression, the current value is used for read access, and the new value is used for write access.

ii. Any assignment to a signal will write to the new value and also notify the event of the signal. Then, the new value will be copied to the current value at the time the notified event is delivered. Thus, the signal will be updated (with a slight delay) and

any behaviors waiting for the update of the signal will be notified about the arrival of a new value.

iii. A **signal** can also be viewed as a **buffered** variable with a built-in clock event (and no reset condition).

iv. Since a signal can be used whenever an event is expected, a signal can also be used with the **wait** and **notify** statements (see the example and Section 2.4.6), or with the **try-trap-interrupt** statement (see Section 2.4.7). In these cases, the **rising** or **falling** operators can be used conveniently to react only to the rising or falling edges of the signal.

v. The example shows two behaviors *B1* and *B2* which add the values given at their input ports *a* and *b* and write the result to their output port *c*. *B1* is modeled as a combinatorial component. It reacts immediately to any change of the input values. *B2*, on the other hand, is modeled as a sequential component. It recomputes its output only at the rising edge of the *CLK* input port.

2.2.7 Buffered type

Purpose: Representation of clocked storage components

Synopsis:

```
default_declaring_list =
        ...
        | signal_class declaration_qualifier_list identifier_declarator
              initializer_opt
        | signal_class type_qualifier_list identifier_declarator
              initializer_opt

declaring_list =
        ...
        | signal_class declaration_specifier declarator initializer_opt
        | signal_class type_specifier declarator initializer_opt

port_declaration =
        port_direction signal_class_opt parameter_declaration
        | ...

signal_class_opt =
        <nothing>
        | signal_class

signal_class =
        ...
        | buffered
        | buffered '[' clock_specifier ']'
        | buffered '[' clock_specifier ';' reset_signal_opt ']'

clock_specifier =
        event_list
        | constant
        | '(' time ')'

event_list =
        event_identifier
        | event_list ',' event_identifier
        | event_list '||' event_identifier

event_identifier =
        identifier
```

```
        | edge_selector identifier
        | identifier edge_selector

edge_selector =
        rising
        | falling

time =
        constant_expression

reset_signal_opt =
        <nothing>
        | identifier
        | '!' identifier
```

Semantics:

(a) The **buffered** type class defines an implicit composite type for the representation of clocked storage components such as registers, register files, or memories.

(b) The **buffered** type class can be specified for ordinary or composite types. It must not be specified for an **event** type or a **signal** type class.

(c) The **buffered** type class must not be used as a member of any composite or aggregate type.

(d) A variable of **buffered** type is implicitly composed of a current and a new value.

(e) A buffered variable can be used in an expression. For every read access, implicitly the current value of the buffered variable is read. For every write access, implicitly the new value of the buffered variable is written.

(f) The clock specifier of a buffered variable determines the internal or external clock that updates the variable. An external clock is specified by an explicit event list. An internal clock is specified by a time period given as a constant or constant expression. The internal clock is an implicit periodic task that notifies update events in a periodic fashion, where the periodic delay is given by the specified time period.

(g) At the time a clock event (internal or external) is received at the buffered variable, its current value is updated with the new value, unless a reset signal is asserted.

(h) If specified, the reset signal of the buffered variable defines an asynchronous reset of the variable. At any time it is asserted, the asynchronous reset signal resets the buffered variable to its initial value. The initial value is determined by the specified initializer for the variable. If no initializer is specified, the initial value defaults to zero.

(i) The asynchronous reset signal must be specified as a variable of **signal** type class. Optionally, the reset signal may be negated, which is specified by the !-operator.

(j) An asynchronous reset is asserted whenever the value of the specified reset signal becomes non-zero. For a negated reset signal, an asynchronous reset is asserted whenever the value of the specified reset signal becomes zero.

(k) As long as an asynchronous reset signal is asserted for a buffered variable, the variable will contain its initial value. No value update will take place.

Example:

```
 1 signal unsigned bit[1]             CLK = 0,
 2                                    RST = 1;
 3 buffered[CLK rising; !RST] bit[16] Reg1 = 10,
 4                                    Reg2 = 20;
 5
 6 void Swap(void)
 7 {
 8    Reg1 = Reg2;
 9    Reg2 = Reg1;
10    wait CLK rising;
11 }
12
13 void Reset(void)
14 {
15    RST = 0;
16    waitfor 10;
17    RST = 1;
18 }
```

Notes:

i. A variable of type class **buffered** can be viewed as a composite variable which contains a current value and a new value. Whenever the buffered variable is accessed in any expression, the current value is used for read access, and the new value is used for write access.

ii. Any assignment to a buffered variable will write to the new value. The new value will be copied to the current value at the time a clock event is notified. In other words, the buffered variable will be updated with a delay of one clock cycle.

iii. An asynchronous reset signal can be specified to reset the buffered variable to its initial value. As soon and as long as the specified reset signal is asserted, the buffered variable will have its initial value.

iv. The example shows two buffered variables *Reg1* and *Reg2* which represent registers driven by the rising edge of a clock. Also, the registers have an asynchronous reset condition which is active low.

v. The function *Swap* in the example demonstrates how the contents of the registers can be exchanged at a clock event.

vi. The function *Reset* asserts the reset signal for the registers for a period of 10 time units, resetting the registers to their initial values of 10 and 20.

2.2.8 Time type

Purpose: Representation of time

Synopsis:

```
waitfor_statement =
        waitfor time ';'

constraint =
        range '(' any_name ';' any_name ';' time_opt ';' time_opt ')' ';'

clock_specifier =
        ...
        | constant
        | '(' time ')'

time_opt =
        <nothing>
        | time

time =
        constant_expression
```

Semantics:

(a) The time type represents the type of time. Time is not an explicit type. It is an implementation dependent integral type (for example, **unsigned long long int**).

(b) The time type is used with the **waitfor** statement (see Section 2.4.8), with the **range** statement in the **do-timing** statement (see Section 2.4.9), and as a possible clock specifier in the **buffered** type (see Section 2.2.7) and **fsmd** statement (see Section 2.4.5).

Example:

```
1 event           SystemClock;
2 const long long CycleTime = 10; // 10ns = 100MHz
3
4 void ClockDriver(void)
5 {
6    while(true)
```

```
 7       { notify SystemClock;
 8         waitfor(CycleTime);
 9       }
10 }
```

Notes:

i. Note that the physical unit of the time type is not defined by the SpecC language. Instead, users should follow a general convention to use the same time unit (such as nano seconds) in all designs and tools in order to simplify integration and interoperability.

2.3 SpecC classes

2.3.1 Behavior class

Purpose: Representation of active objects; container for computation

Synopsis:

```
behavior_declaration =
        behavior_specifier port_list_opt implements_interface_opt ';'

behavior_definition =
        behavior_specifier port_list_opt implements_interface_opt
                '{' internal_definition_list_opt '}' ';'

behavior_specifier =
        behavior identifier

implements_interface_opt =
        <nothing>
        | implements interface_list

interface_list =
        interface_name
        | interface_list ',' interface_name

primary_expression =
        ...
        | this
```

Semantics:

(a) A **behavior** is a class for encapsulation of computation.

(b) A behavior declaration is a class declaration that consists of an optional set of ports and an optional set of implemented interfaces.

(c) A behavior is compatible with another behavior if and only if the number and the types of the behavior ports and the lists of implemented interfaces match.

(d) A behavior definition contains a behavior body which consists of an optional set of local variable declarations and/or definitions, an optional set of behavior and/or channel instantiations, an optional set of method declarations and/or definitions, and a mandatory *main* method declaration or definition.

(e) Through its ports, a behavior can communicate with other behaviors or channels. This is described in detail in Section 2.3.4.

(f) If specified, the **implements** keyword declares the list of interfaces (see Section 2.3.3) that are implemented by the behavior. All the methods of all the listed interfaces must be implemented as methods in the behavior body. Only these methods, and the mandatory *main* method, can be called from outside the behavior (via suitable interfaces). All other methods are private to the behavior.

(g) A behavior definition (a behavior with a body) requires that all listed implemented interfaces are previously fully defined (not only declared).

(h) A behavior that implements an interface can refer back to itself by use of the **this** keyword. **this** can only be used within the scope of the behavior body. The type of **this** is the behavior type. **this** can be passed as an argument to a function or method. In this case, the type of the argument which **this** is assigned to, must be an interface type implemented by the behavior.

(i) A behavior can instantiate other behaviors or channels. This is described in detail in Section 2.3.5.

(j) The *main* method of a behavior is called whenever an instantiated behavior is executed. The completion of the *main* method determines the completion of the execution of the behavior.

(k) As a short cut, the *main* method of a behavior can be called by a statement that consists of only a behavior instance name. For a behavior instance *b*, the statement *b;* is equivalent to the statement *b.main();*.

(l) A SpecC program starts with the execution of the *main* method of the *Main* behavior.

Example:

```
1  behavior B (in int p1, out int p2)
2  {
3     int  a, b;
4
5     int f(int x)
6        {
7          return(x * x);
8        }
9
10    void main(void)
11       {
12         a = p1;          // read data from input port
13         b = f(a);        // compute
14         p2 = b;          // output result to output port
15       }
16 };
```

Notes:

i. The example shows a simple leaf behavior *B*. For typical composite behaviors, please refer to Sections 2.4.1 to 2.4.7.

ii. Local variables and methods, such as *a*, *b*, and *f* in the example, can be used to describe the functionality of a behavior. The actual functionality of the behavior is determined by the execution of its *main* method.

iii. Declarations of behaviors are sufficient to determine compatibility of the behaviors. The behavior body is not needed for this. This is important for reuse of IP and ”plug-and-play”.

iv. In contrast to members of **struct** or **union** definitions, the members of a behavior cannot be accessed from the outside, unless through the implemented interfaces. Note also that only the behavior methods can be made accessable through interfaces, not the variables.

v. A behavior is called a composite behavior if it contains instantiations of other behaviors. Otherwise, it is called a leaf behavior.

vi. Please note that, although *main* and *Main* are recognized by the SpecC compiler as names denoting the start of the program and start of a behavior, these names are not keywords of the SpecC language.

vii. The behavior *Main* usually is a composite behavior containing the test bench of the design as well as the instantiation of the actual design under test.

viii. Implemented interfaces are rarely used with behaviors. However, they are useful for communication schemes that involve call-backs. For example, in a call-back communication, a connected channel implementing a communication protocol can call-back methods provided by the behavior that is calling the channel. In order to enable the channel to call-back the behavior, the channel needs to have a ”pointer” to the behavior. This pointer is passed to the communication method in the channel as an argument of interface type. This argument is supplied by the behavior implementing the call-back by use of the **this** keyword.

ix. Note that the type of **this** is a class in SpecC, not a pointer to a class as in C++.

2.3.2 Channel class

Purpose: Representation of passive objects; container for communication

Synopsis:

```
channel_declaration =
        channel_specifier port_list_opt implements_interface_opt ';'

channel_definition =
        channel_specifier port_list_opt implements_interface_opt
                '{' internal_definition_list_opt '}' ';'

channel_specifier =
        channel identifier

implements_interface_opt =
        <nothing>
        | implements interface_list

interface_list =
        interface_name
        | interface_list ',' interface_name
```

Semantics:

(a) A **channel** is a class for encapsulation of communication.

(b) A channel declaration is a class declaration that consists of an optional set of ports and an optional set of implemented interfaces.

(c) A channel is compatible with another channel if and only if the number and the types of the channel ports and the lists of implemented interfaces match.

(d) A channel definition contains a channel body which consists of an optional set of local variable declarations and/or definitions, an optional set of behavior and/or channel instantiations, and an optional set of method declarations and/or definitions.

(e) Through its ports, a channel can communicate with other behaviors or channels. This is described in detail in Section 2.3.4.

(f) If specified, the **implements** keyword declares the list of interfaces (see Section 2.3.3) that are implemented by the channel. All the methods of all the listed interfaces must be implemented as methods in the channel body. Only these methods can be called from outside the channel (via suitable interfaces). All other methods are private to the channel.

(g) A channel definition (a channel with a body) requires that all listed implemented interfaces are previously fully defined (not only declared).

(h) A channel that implements an interface can refer back to itself by use of the **this** keyword. **this** can only be used within the scope of the channel body. The type of **this** is the channel type. **this** can be passed as an argument to a function or method. In this case, the type of the argument which **this** is assigned to, must be an interface type implemented by the channel.

(i) A channel can instantiate other behaviors or channels. This is described in detail in Section 2.3.5.

(j) For each instance of a channel, the channel methods are mutually exclusive in their execution. Implicitly, each channel instance has a mutex associated with it that the calling thread acquires before and releases after the execution of any method of the channel instance. Further, all mutexes a thread has acquired from any channel instances are implicitly released before and re-acquired after the execution of any **wait** and **waitfor** statements.

Example:

```
 1 interface I
 2 {
 3    void send(int);
 4    int receive(void);
 5 };
 6
 7 channel C implements I
 8 {
 9    int  d;
10
```

```
11    void send(int x)
12       {
13         d = x;
14       }
15
16    int receive(void)
17       {
18         return(d);
19       }
20 };
```

Notes:

i. The example shows the definition of an interface *I* that specifies *send* and *receive* methods. A channel *C* implements the interface *I* by providing a (very simple) implementation of the *send* and *receive* methods by use of an encapsulated integer variable *d*.

ii. In terms of communication, the methods of a channel specify the communication protocol, whereas the variables of a channel resemble the communication media.

iii. A channel is called a hierarchical channel if it contains instantiations of other channels. Otherwise, it is called a leaf channel. A channel is called a wrapper if it instantiates one or more behaviors.

iv. Declarations of channels are sufficient to determine compatibility of the channels. The channel body is not needed. This is important for reuse of IP and ”plug-and-play”.

v. In contrast to members of **struct** or **union** definitions, the members of a channel cannot be accessed from the outside, unless through the implemented interfaces. Note also that only the channel methods can be made accessable through interfaces, not the variables.

vi. For hierarchical communication schemes that involve call-backs, a lower level channel can call-back methods provided by a higher-level channel that called the lower-level channel. Here, the higher-level channel passes a handle to himself as an argument to the lower-level channel by use of the **this** keyword. Then, the lower-level

channel can in turn call-back methods implemented by the higher-level channel. Note that **this** can only be passed as an argument if the argument is of an interface type that is implemented by **this** channel.

vii. Note that the type of **this** is a class in SpecC, not a pointer to a class as in C++.

viii. Note that for safe communication among concurrent threads, the access to the communication variables encapsulated in channels must typically be protected such that no two methods access the same variables at the same time. This protection is guaranteed by the mutex that is implicitly associated with each channel instance. In other words, the required protection of the shared resources in channels is automatically built-in with each channel instance.

ix. Whenever a thread is about to execute a method provided by a channel instance, the thread has to acquire the mutex of the channel instance first, in order to ensure that no other thread is executing any code from the same channel instance. After acquiring the mutex, the thread can execute the channel method. When done, the thread must release the mutex again in order to allow other threads to use the same channel.

x. Furthermore, releasing the channel mutex before a **wait** or **waitfor** statement is important in order to ensure a deadlock-free execution. Otherwise, a thread may be waiting for notification by another thread which is blocked because it cannot acquire the mutex owned by the waiting thread.

xi. Note that acquiring and releasing of channel mutexes is implicit. That is, it is handled automatically by the simulator or refinement tools. There is no need for the user to worry about this.

xii. Note that the implicit mutex in a channel instance is not necessarily required to be present in an implementation. For example, in a non-preemptive simulation algorithm that always executes only one thread at a time, no conflict in running two threads in the methods of a channel instance is possible. Thus, no explicit mutex is required in this case.

xiii. Note that the mutual exclusion in executing methods of channel instances does not imply that the methods are executed in non-preemptive (atomic) manner. That is, a

thread executing a channel method is not guaranteed to be not interrupted by other threads. However, it will not be interrupted by any thread that is executing any method of the same channel instance, unless, of course, it is waiting for an event (at a **wait** statement) or waiting for simulation time increase (at a **waitfor** statement).

2.3.3 Interface class

Purpose: Representation of interfaces between behaviors and channels; container for interface method declarations

Synopsis:

```
interface_declaration =
        interface_specifier ';'

interface_definition =
        interface_specifier '{' internal_declaration_list_opt '}' ';'

interface_specifier =
        interface identifier

internal_declaration_list_opt =
        <nothing>
        | internal_declaration
        | internal_declaration_list internal_declaration

internal_declaration =
        declaration
        | note_definition
```

Semantics:

(a) An **interface** is a class that contains declarations of methods which are implemented in channels or behaviors. Via an interface, a class can call methods provided by another class that **implements** the interface.

(b) An interface declaration consists of the keyword **interface** followed by its name.

(c) An interface definition contains an interface body which consists of an optional set of method declarations. No variable declarations or definitions, no behavior or channel instantiations, and no method definitions must be contained in the scope of an interface body.

(d) An interface is a type that may be used for behavior or channel ports, or for function or method arguments. An interface cannot be instantiated.

(e) A port of interface type must be mapped onto a behavior or channel instance (see Section 2.3.5) that **implements** the interface.

(f) An argument of interface type must be initialized by **this** within the scope of a behavior (see Section 2.3.1) or channel (see Section 2.3.2) that **implements** the interface.

(g) A port or argument of interface type can be used to call a method declared by the interface. The actual method definition called by such an interface call is determined by the mapping of the interface port or argument.

Example:

```
 1 interface I
 2 {
 3    void send(int x);
 4    int receive(void);
 5 };
 6
 7 channel C implements I;
 8
 9 behavior B1(I i)
10 {
11    void main(void)
12    {
13       i.send(42);
14    }
15 };
16
17 behavior B2
18 {
19    C   c;
20    B1  b(c);
21
22    void main(void)
23    {
24       b.main();
25    }
26 };
```

Notes:

i. Interfaces provide a flexible way of communication between behaviors and channels. By use of interfaces, both behaviors and channels become easily interchangeable with compatible components (”plug-and-play”).

ii. Typically, there are many channels (or behaviors) which implement an interface. Because each channel (or behavior) is required to provide an implementation for all methods declared in the interface, any one of these channels (or behaviors) may be used in a mapping of a port of the interface type. Thus, the implementation of the interface methods (typically a communication protocol) can be easily exchanged, just by mapping the interface port to a compatible channel (or behavior) instance.

iii. The example shows an **interface** *I* which declares a *send* and a *receive* method as a simple communication protocol. The channel *C* implements the interface *I* (even though the actual implementation in the channel body is not shown), so it may be used as a mapping for the interface.

iv. Behavior *B1* in the example has a port *i* of the interface type *I*. Via this port, *B1* can call the methods provided by *I*, as is shown in line 13 where the *send* method is called.

v. Behavior *B2* shows an instance *c* of the channel *C* in line 19. Then in line 20, a behavior *b* of type *B1* is instantiated and its port is mapped onto the channel *c*. Note that the port is of interface type *I*, which is implemented by the channel *C*.

2.3.4 Ports

Purpose: Representation of communication ports

Synopsis:

```
port_list_opt =
        <nothing>
        | '(' ')'
        | '(' port_list ')'

port_list =
        port_declaration
        | port_list ',' port_declaration

port_declaration =
        port_direction signal_class_opt parameter_declaration
        | interface_parameter

port_direction =
        <nothing>
        | in
        | out
        | inout

interface_parameter =
        interface_name
        | interface_name identifier
```

Semantics:

(a) Behavior and channel classes have a list of ports through which they communicate. These ports are defined with the declaration of the behavior or channel they are attached to (similar as function parameters are defined with a function declaration).

(b) A port can be one of two types: standard or interface type.

(c) A standard type port is of any SpecC type. In addition, a port direction may be specified as a port type modifier which restricts the way the port can be accessed and connected.

(d) The port direction can be **in**, **out**, or **inout**. If unspecified, the port direction defaults to **inout**.

(e) An **in** port allows only read access from within the class scope, and only write access from the outside.

(f) An **out** port allows only write access from within the class scope, and only read access from the outside.

(g) An **inout** port may be accessed bidirectionally.

(h) For a port of event or signal type, read access is performed by a **wait**, **trap**, or **interrupt** statement on the event, or when used as clock specifier, sensitivity list or reset condition with the **fsmd** statement or **buffered** data type. Write access is performed by a **notify** or **notifyone** statement on the port.

(i) An interface type port allows to call the methods provided by the interface class. An interface type port must not have any port direction.

Example:

```
1 behavior B(in int p1, out int p2, in event clk);
2
3 interface I;
4
5 channel  C(inout bool f) implements I;
```

Notes:

i. The example shows a behavior *B* with an input port *p1*, an output port *p2*, and a clock input port *clk*.

ii. The channel *C* has a bidirectional port *f*.

2.3.5 Class instantiation and port mapping

Purpose: Structural hierarchy and connectivity of behaviors and channels

Synopsis:

```
instance_declaring_list =
        behavior_or_channel instance_declarator
        | instance_declaring_list ',' instance_declarator

behavior_or_channel =
        behavior_name
        | channel_name

instance_declarator =
        identifier port_mapping_list_opt

port_mapping_list_opt =
        <nothing>
        | '(' port_mapping_list ')'

port_mapping_list =
        port_mapping_opt
        | port_mapping_list ',' port_mapping_opt

port_mapping_opt =
        <nothing>
        | port_mapping

port_mapping =
        bit_slice
        | port_mapping '@' bit_slice

bit_slice =
        constant
        | '(' constant_expression ')'
        | identifier
        | identifier '[' constant_expression ':' constant_expression ']'
        | identifier '[' constant_expression ']'
```

Semantics:

(a) Structural hierarchy is described by child behaviors and/or child channels instantiated as components inside compound behaviors and channels.

(b) Connectivity is described by the mapping of the ports of child behaviors and/or child channels.

(c) At class instantiation, a port mapping list defines the mapping for each port of the class. The number of ports must match the number of mappings in the port mapping list. If there are no ports, then no port mapping list must be specified.

(d) A port mapping maps a port of the instantiated class onto a constant, variable, port or instance of suitable type, or is left open.

(e) A constant port mapping is only allowed for ports with port direction **in**. Unless a constant literal is specified directly, a constant expression in parenthesis is evaluated to a constant at compile time. The type of the constant must be convertable to the port type.

(f) An open port mapping is only allowed for ports with port direction **out**.

(g) For a port mapping to a variable, the variable type must match the type of the port. The port direction is not considered part of the port type in this matching.

(h) For a port mapping to a port of the parent class, the types (without port direction) of both ports must match. For the port direction, an instance port with direction **in** can only be mapped onto a class port with direction **in** or **inout**. An instance port with direction **out** can only be mapped onto a class port with direction **out** or **inout**. An instance port with direction **inout** can only be mapped onto a class port with direction **inout**.

(i) For a port mapping to an instance, the port must be of interface type and the class of the instance must implement the interface.

(j) A port of bit vector type can be mapped onto a list of concatenated bit slices. In this case, the mapping rules listed above apply accordingly for each single bit of the bitvector.

(k) Concatenation and bit slicing must not be used for port mappings of non-bitvector type.

Example:

```
 1 interface I { };
 2 channel   C     (inout bool f) implements I;
 3 behavior  B1    (in int p1, out bit[7:0] p2, in event clk);
 4 behavior  B2    (I i, out event clk);
 5 behavior  Adder8(in bit[8] a, in bit[8] b, in bit[1] carry_in,
 6                  out bit[8] sum, out bit[1] carry_out);
 7
 8 behavior B (bit[7:0] bus1, bit[15:0] bus2)
 9 {
10    bool         b;
11    int          i;
12    event        e;
13    bit[8]       a;
14
15    C      c (b);
16    B1     b1(i, bus1, e);
17    B2     b2(c, e);
18    Adder8 a8(a,         // mapping onto variable
19              bus1,      // mapping onto port
20              0b,        // mapping onto constant
21              bus2[7:0], // mapping onto bit slice
22                  );     // open mapping
23
24    void main(void)
25    {
26       b1.main();
27       b2.main();
28       a8.main();
29    }
30 };
```

Notes:

i. The example shows four class instantiations. In line 15, an instance *c* of channel *C* is instantiated. Its only port of type **bool** is mapped onto the Boolean variable *b*.

ii. In line 16, a behavior *b1* of type *B1* is instantiated. Its input port *p1* is mapped onto the variable *i*, whereas the output port *p2* is mapped onto the class port *bus1*. Finally, the clock port *clk* is mapped onto the event *e*.

iii. In line 17, an instance *b2* is defined as a *B2* type behavior. Its ports are mapped onto the channel *c* and event *e*.

iv. In line 18, an adder *a8* is instantiated. The left input is mapped onto variable *a*, whereas the right input is mapped onto port *bus1* (line 19). In line 20, the carry input is connected to zero (hardwired to GND). The output is mapped onto the lower bits of *bus2* in line 21. Finally in line 22, the carry output is left open (i.e. it is unused).

v. Note that the rules for mapping instance ports onto class ports ensure that a class port can only be accessed in its specified direction.

2.4 SpecC statements

2.4.1 Sequential execution

Purpose: Representation of sequential control flow

Synopsis:

```
compound_statement =
        '{' '}'
        | '{' declaration_list '}'
        | '{' statement_list '}'
        | '{' declaration_list statement_list '}'

statement_list =
        statement
        | statement_list statement
        | statement_list note_definition

statement =
        labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        | spec_c_statement

spec_c_statement =
        concurrent_statement
        | fsm_statement
        | fsmd_statement
        | exception_statement
        | timing_statement
        | wait_statement
        | waitfor_statement
        | notify_statement
```

Semantics:

(a) Sequential execution and control flow is represented by the same statements as in

ANSI-C. In particular, the following statements can be used: **if-then-else**, **switch-case-default**, **for**, **while**, **do-while**, **goto**, **break**, **continue**, **return**.

(b) Sequential execution can be organized hierarchically by use of function calls, and by calls to non-private methods of behavior and channel instances.

Example:

```
 1 behavior B;
 2
 3 behavior B_seq(void)
 4 {
 5    B    b1, b2, b3;
 6
 7    void main(void)
 8    {
 9        b1.main();
10        b2.main();
11        b3.main();
12    }
13 };
```

Notes:

i. The example shows the trivial case of sequential, unconditional execution of three child behaviors, *b1*, *b2* and *b3*.

ii. Note that a short cut notation for calling behavior instances exists (see Section 2.3.1). With the short notation, the three statements in lines 9 through 11 of the example can be reduced to *b1; b2; b3*.

2.4.2 Concurrent execution

Purpose: Representation of concurrency

Synopsis:

```
concurrent_statement =
        par compound_statement
        ...

compound_statement =
        '{' '}'
        | '{' declaration_list '}'
        | '{' statement_list '}'
        | '{' declaration_list statement_list '}'

statement_list =
        statement
        | statement_list statement
        ...
```

Semantics:

(a) The **par** statement specifies concurrent execution.

(b) Every statement in the compound statement block following the **par** keyword forms a new thread of control that is executed concurrently.

(c) The execution of the concurrent threads is defined by the time interval formalism described in Section 3.3.

(d) An abstract simulation algorithm for the semantics of concurrency is specified in Section 3.6. This algorithm represents one valid implementation of concurrency. Other valid implementations may exist.

(e) The execution of the **par** statement completes when each thread of control has finished its execution.

(f) The statements in the compound statement block after the **par** keyword are restricted to calls to *main* methods of behaviors. No other statement type is allowed.

Example:

```
 1 behavior B;
 2
 3 behavior B_par(void)
 4 {
 5    B    b1, b2, b3;
 6
 7    void main(void)
 8    {
 9       par{ b1.main();
10            b2.main();
11            b3.main();
12          }
13    }
14 };
```

Notes:

i. Concurrent threads may be executed truly in parallel, or portion-wise sequentially, where the order and size of the portions is undefined. No assumptions about the order of execution, the use of preemptive or non-preemptive execution, or any atomic execution must be made. These are undefined for the SpecC language.

ii. A sequential simulator may choose any order of execution for the concurrent threads, including interleaved (preemptive) execution.

iii. For simulation, typically a dynamic scheduler decides the order and interleaving of the execution of the concurrent threads. That is, the scheduler always executes only one thread at a time and decides when to suspend and when to resume a thread depending on simulation time advance, synchronization points, and/or time outs.

iv. Note that, because the execution of concurrent theads is essentially undefined, explicit synchronization (see Section 2.4.6) and explicit mutual exclusion (see Section 2.3.2) are necessary in order to make concurrent threads cooperate safely.

v. The example shows the concurrent execution of three child behaviors *b1*, *b2* and *b3*. The compound behavior *B_par* finishes when *b1*, *b2* and *b3* have completed their execution.

vi. Note that a short cut notation for calling behavior instances exists (see Section 2.3.1). With the short notation, the **par** statement in the example can be reduced to **par**{*b1; b2; b3*}.

2.4.3 Pipelined execution

Purpose: Explicit representation of pipelining

Synopsis:

```
storage_class =
        ...
        | piped
        | storage_class piped

concurrent_statement =
        ...
        | pipe compound_statement
        | pipe '(' comma_expression_opt ';' comma_expression_opt
               ';' comma_expression_opt ')' compound_statement

compound_statement =
        '{' '}'
        | '{' declaration_list '}'
        | '{' statement_list '}'
        | '{' declaration_list statement_list '}'

statement_list =
        statement
        | statement_list statement
        ...
```

Semantics:

(a) The **pipe** statement specifies execution in pipelined manner, a special form of concurrent execution.

(b) Every statement in the compound statement block following the **pipe** keyword represents a pipeline stage. Each pipeline stage forms a new thread of control that is executed concurrently.

(c) The execution of the concurrent threads is defined by the time interval formalism described in Section 3.3.

(d) An abstract simulation algorithm for the semantics of concurrency is specified in Section 3.6. This algorithm represents one valid implementation of concurrency. Other valid implementations may exist.

(e) The optional set of arguments to the **pipe** statement specifies the number of pipeline iterations. The first argument expression serves as an initializer and is evaluated once at the beginning of the pipeline execution. The second expression represents the iteration condition which is evaluated at the beginning of each pipeline iteration. While the iteration condition evaluates to **true**, the pipeline executes the pipeline stages, otherwise the pipeline is flushed and the **pipe** statement terminates. The third expression is evaluated once after each iteration.

(f) If no arguments are specified for the **pipe** statement, the first and third arguments default to empty expressions, and the iteration condition defaults to **true**.

(g) The **pipe** statement executes the N pipeline stages in three phases. In the first phase, the pipeline is filled and, in the n-th iteration, only the first n pipeline stages are executed concurrently, where $n \leq N$. Then, in the second phase, all N pipeline stages are executed concurrently in each iteration. As soon as the iteration condition evaluates to **false** during the first or second phase, the third phase flushes the pipeline by executing $n-1$ more iterations. In the m-th last iteration, only the m last pipeline stages are executed concurrently. After the pipeline has been flushed, the **pipe** statement terminates.

(h) Unless aborted through an exception (see Section 2.4.7), the **pipe** statement executes each pipeline stage for the same number of times.

(i) The statements in the compound statement block after the **pipe** keyword are restricted to calls to *main* methods of behaviors. No other statement type is allowed.

(j) Variables used by pipeline stages, which are declared in a non-global scope visible from a **pipe** statement, can be declared of storage class **piped**. A **piped** variable represents a buffer between pipeline stages that operates in first-in-first-out (FIFO) order.

(k) Write accesses to **piped** variables store data in the first stage of the FIFO buffer. Read accesses read data from the last stage of the FIFO buffer. If a port of a behavior instance that represents a pipeline stage is mapped onto a **piped** variable, the port must be of direction **in** or **out**.

(l) The **piped** storage class can be specified multiple times with a variable declaration. The number n of **piped** keywords specifies the number $n+1$ of stages in the FIFO buffer.

(m) Variables with **piped** storage class, that are in visible scope from a **pipe** statement, are synchronized with the **pipe** statement. After each iteration of the pipeline, the data stored in **piped** variables is shifted by one stage in the FIFO.

Example:

```
 1 behavior B(in int p1, out int p2);
 2
 3 behavior B_pipe(in int a, out int b)
 4 {
 5    int          x;
 6    piped int    y;
 7    B            b1(a, x),
 8                 b2(x, y),
 9                 b3(y, b);
10
11    void main(void)
12    { int i;
13      pipe(i=0;i<10;i++)
14         { b1.main();
15           b2.main();
16           b3.main();
17          }
18    }
19 };
```

Notes:

i. The arguments of the **pipe** statement are basically the same as the arguments of the **for** statement. Also, the execution of a **pipe** statement basically resembles the execution of a **for** loop, except that the loop body is organized concurrently in pipeline

fashion. Note also that, if the arguments of the **pipe** statement are unspecified, the **pipe** statement acts as an endless loop (it does not terminate), the same way as the **for** statement without arguments.

ii. The threads in a **pipe** statement represent pipeline stages and are executed in pipelined fashion. Basically, each pipeline stage runs concurrently to the others, but works on different sets of data. Here, **piped** variables can be used as buffers between the pipeline stages that are automatically updated with every iteration.

iii. The example shows a pipeline behavior *B_pipe* consisting of three stages represented by the behavior instances *b1*, *b2* and *b3*. In the first iteration, only *b1* is executed. When *b1* finishes, the second iteration starts and *b1* and *b2* are executed concurrently. In the third iteration, after *b1* and *b2* have completed, *b3* is executed in parallel with *b1* and *b2*. Every following iteration executes the same way as the third iteration, until the iteration condition $i<10$ becomes false. Then, *b2* and *b3* are executed concurrently one more time, and finally only *b3* is executed once.

iv. In the example, *x* is a standard variable connecting *b1* (pipeline stage 1) with *b2* (stage 2). This variable is not **piped**, in other words, every access from stage 1 is immediately visible in stage 2. On the other hand, variable *y* connecting *b2* and *b3* is **piped**. Data computed by behavior *b2* and stored in *y* is available for processing by *b3* in the next pipeline iteration when *b2* already produces new data.

v. Note that a short cut notation for calling behavior instances exists (see Section 2.3.1). With the short notation, the **pipe** statement in the example can be reduced to **pipe***(*$i = 0; i < 10; i++$*){b1; b2; b3}*.

2.4.4 Abstract finite state machine execution

Purpose: Explicit representation of abstract finite state machines

Synopsis:

```
fsm_statement =
        fsm '{' '}'
        | fsm '{' transition_list '}'

transition_list =
        transition
        | transition_list transition

transition =
        state ':'
        | state ':' cond_branch_list
        | state ':' '{' '}'
        | state ':' '{' cond_branch_list '}'

state =
        identifier
        | identifier compound_statement

cond_branch_list =
        cond_branch
        | cond_branch_list cond_branch

cond_branch =
        if '(' comma_expression ')' goto identifier ';'
        | goto identifier ';'
        | if '(' comma_expression ')' break ';'
        | break ';'
```

Semantics:

(a) Finite state machine (FSM) execution is a special form of sequential execution which allows the explicit specification of states, state transitions and hierarchy.

(b) The **fsm** statement consists of a list of states and a list of state transitions from each state. The states are represented by either local compound statements or behavior instances.

(c) A local state in the FSM is specified by a compound statement block placed between the state label and the colon that separates the state from its transitions to the next state.

(d) If a state is specified locally by a compound statement block, then no behavior must be instantiated within the visible scope of the **fsm** statement where the instance name matches the name of the state.

(e) A non-local state in the FSM is specified by a behavior that is instantiated within the visible scope of the **fsm** statement where the instance name matches the name of the state.

(f) If a state is specified non-locally by a behavior instantiated within the visible scope of the **fsm** statement where the instance name matches the name of the state, then no compound statement must be specified between the state label and the colon that separates the state from its transitions to the next state.

(g) A state transition is a triple $\langle current\_state, condition, next\_state \rangle$. The *current_state* and the *next_state* are specified in the form of labels and denote local states or behavior instances of the same name. At the time of a transition, the *condition* is evaluated to determine whether the transition is taken, or not.

(h) Each state must be listed exactly once in the transition list as a *current_state*.

(i) The transition *condition* is optional. If unspecified, the *condition* defaults to **true**.

(j) The *next_state* is specified as a label denoting a state of the **fsm**, or as a **break** statement. The **break** statement terminates the execution of the **fsm** statement.

(k) The execution of a **fsm** statement starts with the execution of the state that is listed first in the transition list.

(l) A non-local state in the **fsm** is executed by an implicit call of the *main* method of the behavior instance denoted by the *current_state* label. The execution of the state terminates with the completion of the *main* method.

(m) A local state in the **fsm** is executed by the execution of the statements in the local compound statement block. The execution of the state terminates with the completion of the compound statement.

(n) After a state is executed, the transitions listed with the state determine the next state to be executed. For this, the *conditions* of the transitions are evaluated in the specified order and the first *condition* that evaluates to **true** determines the next state, which is then immediately executed.

(o) If none of the conditions evaluates to **true**, the next state defaults to the following state listed in the **fsm**. After the last state of the **fsm**, the next state defaults to the termination of the **fsm** statement.

Example:

```
 1 behavior B;
 2
 3 behavior B_fsm(in int a, in int b)
 4 {
 5    B    b1, b2, b3;
 6
 7    void main(void)
 8    {
 9      fsm{ b1: { if (b < 0)  break;
10                 if (b >= 0) goto b2;
11               }
12           b2: { if (a > 0)  goto b1;
13                 goto b3;
14               }
15           b3: { break;
16               }
17         }
18    }
19 };
```

Notes:

i. Note that both, abstract Mealy-type (sensitive to the current state and the current input) and abstract Moore-type (sensitive only to the current state) finite state machines, can be modeled with the **fsm** statement.

ii. Note that hierarchical FSMs can be specified by use of local **fsm** statements in the states of the parent FSM.

iii. Note that in contrast to the **fsmd** statement, the **fsm** statement represents an abstract FSM whose states can be hierarchical. Also, the **fsm** statement transitions to the next state immediately on the completion of the current state, whereas the **fsmd** statement transitions to the next state on the event of a specified clock.

iv. Note that the transition section of the **fsm** statement does not allow arbitrary statements. The SpecC grammar limits the state transitions to well-defined triples.

v. The default transitions of the **fsm** statement are similar to the default control flow within a **switch** statement where **case** statements are not terminated by a **break** statement.

vi. The example shows a behavior *B_fsm* that models a finite state machine with three states *b1*, *b2* and *b3*. All the states are defined non-locally by behavior instances.

2.4.5 Finite state machine with datapath

Purpose: Explicit representation of a finite state machine with datapath (FSMD)

Synopsis:

```
fsmd_statement =
        fsmd '(' fsmd_head ')' fsmd_body

fsmd_head =
        clock_specifier
        | clock_specifier ';' sensitivity_list_opt
        | clock_specifier ';' sensitivity_list_opt ';' reset_signal_opt

clock_specifier =
        event_list
        | constant
        | '(' time ')'

sensitivity_list_opt =
        <nothing>
        | event_list

reset_signal_opt =
        <nothing>
        | identifier
        | '!' identifier

event_list =
        event_identifier
        | event_list ',' event_identifier
        | event_list '||' event_identifier

event_identifier =
        identifier
        | edge_selector identifier
        | identifier edge_selector

edge_selector =
        rising
        | falling

time =
        constant_expression
```

```
fsmd_body =
        '{' '}'
        | '{' declaration_list '}'
        | '{' reset_state '}'
        | '{' declaration_list reset_state '}'
        | '{' default_action '}'
        | '{' declaration_list default_action '}'
        | '{' reset_state default_action '}'
        | '{' declaration_list reset_state default_action '}'
        | '{' fsmd_state_list '}'
        | '{' declaration_list fsmd_state_list '}'
        | '{' reset_state fsmd_state_list '}'
        | '{' declaration_list reset_state fsmd_state_list '}'
        | '{' default_action fsmd_state_list '}'
        | '{' declaration_list default_action fsmd_state_list '}'
        | '{' reset_state default_action fsmd_state_list '}'
        | '{' declaration_list reset_state default_action fsmd_state_list '}'

reset_state =
        if '(' comma_expression ')' action

default_action =
        action

fsmd_state_list =
        fsmd_state
        | fsmd_state_list fsmd_state

fsmd_state =
        identifier_or_typedef_name ':' action

action =
        '{' '}'
        | '{' declaration_list '}'
        | '{' rtl_statement_list '}'
        | '{' declaration_list rtl_statement_list '}'

rtl_statement_list =
        rtl_statement
        | rtl_statement_list rtl_statement
        | rtl_statement_list note_definition

rtl_statement =
        rtl_labeled_statement
```

```
        | rtl_compound_statement
        | expression_statement
        | rtl_selection_statement
        | rtl_jump_statement

rtl_labeled_statement =
        case constant_expression ':' rtl_statement
        | default ':' rtl_statement

rtl_compound_statement =
        '{' '}'
        | '{' declaration_list '}'
        | '{' rtl_statement_list '}'
        | '{' declaration_list rtl_statement_list '}'

rtl_selection_statement =
        if '(' comma_expression ')' rtl_statement
        | if '(' comma_expression ')' rtl_statement else rtl_statement
        | switch '(' comma_expression ')' rtl_statement

rtl_jump_statement =
        goto identifier_or_typedef_name ';'
        | break ';'
```

Semantics:

(a) The **fsmd** statement explicitly represents a finite state machine with datapath (FSMD).

(b) The **fsmd** statement consists of a header and a body. The **fsmd** header defines the clock, sensitivity and asynchronous reset signal of the FSMD. The **fsmd** body defines the states and state transitions of the FSMD.

(c) The clock of the FSMD is defined with the clock specifier. The clock specifier determines the internal or external clock that triggers the state transitions of the FSMD. An external clock is specified by an explicit event list. An internal clock is specified by a time period given as a constant or constant expression. The internal clock is an implicit periodic task that notifies clock events in a periodic fashion, where the periodic delay is given by the specified time period.

(d) At the time a clock event (internal or external) is received by the FSMD, a state transition takes place and the current state is updated to the next state. Also, the actions associated with the new current state are executed, unless a synchronous or asynchronous reset is asserted.

(e) If specified, the sensitivity list of the **fsmd** statement defines additional events upon which the actions associated with the current state are executed again, unless a synchronous or asynchronous reset is asserted. No state transition takes place for events in the sensitivity list, unless clock events occur at the same time.

(f) If specified, the reset signal of the **fsmd** statement defines an asynchronous reset of the FSMD. At any time it is asserted, the asynchronous reset signal resets the current state of the FSMD to the initial state. The initial state of the FSMD is the first regular state listed in the **fsmd** body.

(g) The asynchronous reset signal must be specified as a variable of **signal** type class. Optionally, the reset signal may be negated, which is specified by the !-operator.

(h) An asynchronous reset is asserted whenever the value of the specified reset signal becomes non-zero. For a negated reset signal, an asynchronous reset is asserted whenever the value of the specified reset signal becomes zero.

(i) As long as an asynchronous reset signal is asserted, the FSMD will stay in its initial state. No state transition will take place.

(j) The **fsmd** body consists of an optional declaration list, an optional reset state, an optional default action block, and an optional list of FSMD states.

(k) If specified, the declaration list in the **fsmd** body defines variables representing the wires, busses, registers, etc. of the FSMD.

(l) If specified, the reset state in the **fsmd** body defines a synchronous reset of the FSMD. A synchronous reset of the FSMD is asserted if the reset condition specified at the **if** statement evaluates to **true** at the occurence of a clock event. In this case, only the action block of the reset state is executed. No default actions and no regular state are executed.

(m) If specified, the default action block in the **fsmd** body defines default actions that are executed before every regular state.

(n) The list of regular states in the **fsmd** body defines the states of the FSMD. Each FSMD state must be labeled with a unique name and must be listed exactly once in the state list.

(o) The actions associated with each FSMD state are limited to valid register transfers or state transitions. Register transfers are specified by conditional or unconditional assignment expressions.

(p) State transitions are specified by conditional or unconditional **goto** or **break** statements. A **goto** statement specifies the transition to the next state in the FSMD, where the specified label must match a state label defined in the **fsmd** statement. A **break** statement terminates the execution of the **fsmd** statement.

(q) If no state transition is executed in a state, the next state defaults to the current state.

Example:

```
 1 behavior B(in  signal bit[ 1] CLK,
 2            in  signal bit[ 1] RST,
 3            in  signal bit[32] a,
 4            in  signal bit[32] b,
 5            out signal bit[32] s)
 6 {
 7    void main(void)
 8    {
 9       fsmd(CLK falling)
10       {
11          buffered[CLK falling] bit[32] sum, tmp;
12
13          if (RST) { sum = 0;
14                     goto S1;
15                   }
16                   { s = sum;
17                   }
18          S1 :     { tmp = a * b;
19                     goto S2;
20                   }
```

```
21          S2 :     { sum += tmp;
22                     goto S1;
23                   }
24       }
25    }
26 };
```

Notes:

i. The specification of the **fsmd** statement is based on the RTL semantics draft standard as defined by Accellera [6].

ii. Note that in contrast to the **fsm** statement, the **fsmd** statement represents a controller at the register transfer level (RTL) which is driving a data path. The FSMD states contain the register transfers performed by the datapath. Also, the **fsmd** statement transitions to the next state on the event of a specified clock, whereas the **fsm** statement transitions immediately upon the completion of the current state.

iii. The **fsmd** statement can be understood as a loop where in each iteration a specified set of statements, called a state, is executed. Each loop iteration is called a clock cycle. Each cycle basically starts with an implicit **wait** statement on an external clock event or with an implicit **waitfor** statement representing an internal clock, as indicated by the clock specifier.

iv. Note that an external FSMD clock may be specified as a simple event (i.e. *event CLK;*) or as a signal (i.e. *signal bit[1] CLK;*). The former reflects an abstract clock, whereas the latter models a very specific clock with explicit hi (1) and lo (0) phases. Moreover, by specifying a list of events and/or signals for the clock specifier, the FSMD may be driven by multiple clocks.

v. Note that both Mealy-type (sensitive to the current state and the current input) and Moore-type (sensitive only to the current state) finite state machines can be modeled with the **fsmd** statement. Both types are easily identified by the existence or non-existence of the sensitivity list. If a sensitivity list is specified, the **fsmd** represents a Mealy machine, otherwise a Moore machine.

vi. Note that the clock specifier and the reset signal of the **fsmd** statement are specified the same way as the clock specifier and the asynchronous reset signal of a **buffered** variable (see Section 2.2.7). In fact, the implicit state register of the FSMD can be seen as a buffered variable for which the clock specifier and reset signal are taken from the **fsmd** header.

vii. The **fsmd** body starts with a list of declarations of local variables. These variables represent registers and wires inside the FSMD, or simply uninterpreted variables, as defined by the Accellera RTL semantics (see [6]).

viii. The optional reset state in the **fsmd** body represents a synchronous reset of the FSMD. The specified reset condition is checked at the beginning of every clock cycle. If it evaluates to **true**, a reset cycle is executed.

ix. The optional set of default actions will be executed first in every state, but take effect only if the assignments are not overwritten within the same clock cycle. This can be used to assign default values to registers and ports so that these assignments don't have to be repeated in every state when the registers and ports are not used. On the other hand, in those states that actually do use the registers and ports, the default assignments can be easily overwritten by specific assignments.

x. As with the **fsm** statement, the states in the **fsmd** statement are identified by state labels and transitions among these states are specified as conditional or unconditional **goto** statements. Also, for exiting the FSMD, a **break** statement is used. However, if no state transition is executed, the next state defaults to the current state in the **fsmd** statement (whereas it defaults to the following state in the **fsm** statement).

xi. Note that the body of the **fsmd** statement does not allow arbitrary statements. The SpecC grammar limits the actions and state transitions to well-defined register transfers. In particular, the state actions may include expression statements, such as assignments and function calls, selection statements, such as **if** and **switch** statements, and state transitions by use of **goto** and **break**. However, loop statements, such as **for**, **while** and **do** loops, behavioral hierarchy, such as **par**, **pipe** and **fsm** statements, exceptions, such as **try**, **interrupt** and **trap**, timing statements, such as **do-timing**

and **waitfor**, and synchronization, such as **wait**, **notify** and **notifyone**, are *not* allowed in any state because these do not represent valid register transfers according to the semantics of a FSMD.

xii. Note that a large FSMD can be specified by use of multiple **fsmd** statements which are executed sequentially, possibly under control of a top-level **fsm** statement.

xiii. The example shows a behavior *B* modeling a simple finite state machine with data path by use of the **fsmd** statement. The component has a clock input port *CLK*, a reset input port *RST*, two data input ports *a* and *b*, and a data output port *s*.

xiv. The FSMD in the example is a synchronous component, executing a new state with every falling edge of the clock signal *CLK*. It also has a synchronous reset which is activated if the reset port *RST* is set to hi.

xv. Internally, the example FSMD uses two registers *sum* and *tmp* which are clocked the same way as the FSMD.

xvi. In state *S1*, the FSMD computes the product of its input ports *a* and *b* and stores the result in register *tmp*. Next in state *S2*, the value of *tmp* is accumulated in the register *sum*.

xvii. Note that the default assignment in line 16 in every state makes the value stored in register *sum* available at the output port *s*. Thus, the component *B* acts as a multiply-accumulator (MAC) unit which takes two clock cycles for each computation.

2.4.6 Synchronization

Purpose: Representation of synchronization

Synopsis:

```
wait_statement =
        wait paren_event_list ';'
        | wait paren_and_event_list ';'

notify_statement =
        notify paren_event_list ';'
        | notifyone paren_event_list ';'

paren_event_list =
        event_list
        | '(' event_list ')'

event_list =
        event_identifier
        | event_list ',' event_identifier
        | event_list '||' event_identifier

paren_and_event_list =
        and_event_list
        | '(' and_event_list ')'

and_event_list =
        event_identifier '&&' event_identifier
        | and_event_list '&&' event_identifier

event_identifier =
        identifier
        | edge_selector identifier
        | identifier edge_selector

edge_selector =
        rising
        | falling
```

Semantics:

(a) Synchronization of concurrent threads of execution is specified by the **wait**, **notify**

and **notifyone** statements which operate by the use of events (see Section 2.2.5) or signals (see Section 2.2.6).

(b) The semantics of synchronization of concurrent threads are defined by use of the time interval formalism described in Section 3.5.

(c) An abstract simulation algorithm for the synchronization semantics is specified in Section 3.6. This algorithm represents one valid implementation of the synchronization semantics. Other valid implementations may exist.

(d) The **wait** statement, where a single event is specified as an argument or a list of events separated by comma or logical-or (||) is specified as argument, suspends the current thread from execution until at least one of the events specified as arguments is notified. Then, the thread becomes active again and resumes its execution.

(e) The **wait** statement, where a list of events separated by logical-and (&&) is specified as argument, suspends the current thread from execution until all of the events specified as arguments are notified, regardless of the order of the notification. Then, the thread becomes active again and resumes its execution.

(f) The **notify** statement triggers the events specified as arguments so that all threads, which are currently waiting on any of these events, are notified. If there is no thread waiting or sensitive to the notified events at the time of the execution of the **notify** statement, then the statement has no effect.

(g) The **notifyone** statement triggers the events specified as arguments so that at most one thread, which is currently waiting on any of these events, is notified. The thread is non-deterministically chosen. If there is no thread waiting or sensitive to the notified events at the time of the execution of the **notifyone** statement, then the statement has no effect.

Example:

```
1 #include <stdio.h>
2
3 behavior A(out int x, out event e)
```

```
 4 {
 5    void main(void)
 6    {
 7       x = 42;
 8       notify e;
 9    }
10 };
11
12 behavior B(in int x, in event e)
13 {
14    void main(void)
15    {
16       wait(e);
17       printf("%d", x);
18    }
19 };
20
21 behavior Main
22 {
23    int   x;
24    event e;
25    A a(x, e);
26    B b(x, e);
27
28    int main(void)
29    { par { a.main();
30            b.main();
31          }
32      return(0);
33    }
34 };
```

Notes:

i. The **wait** statement operates with either *or* or *and* semantics. If the events specified for the **wait** statement are separated by comma or logical-or (||), then *or* semantics are used and the notification of only one of the events is sufficient to resume the execution. On the other hand, if the events specified for the **wait** statement are separated by logical-and (&&), then *and* semantics are used and all of the events must be notified in order to resume the execution. In the latter case, it does not matter in which order the events are received.

ii. Note that the **wait** statement with *and* semantics is equivalent to a set of parallel **wait** statements where each **wait** statement is waiting for a single event. The execution resumes only if all events have been notified.

iii. The SpecC standard channel library described in Appendix B.2 contains many standard synchronization channels that can be used conveniently and safely. It is recommended to use these standard channels whenever possible instead of the primitives **wait**, **notify**, and **notifyone**.

iv. Note that, when resuming execution from a **wait** statement due to a notified event, the **wait** statement provides no information to determine which of the specified events was actually notified. If such information is required, it must be supplied explicitly by the event generator, for example, by setting a specific value in an additional variable, or by using a **signal** (which includes a value) instead of the event (that does not carry any value).

v. Notified events can be thought of as being collected until no active behavior is available any more for execution. Then, the set of notified events is delivered to the waiting threads, activating those threads that are waiting on any of them. As a result, the **notify** and **notifyone** statements are guaranteed to reach all threads that are currently waiting for the event, including active threads that will be waiting for the event as their immediate next state.

vi. The example shows two parallel executing behaviors *A* and *B*, where *A* sends data via *x* to *B*. To make sure that *B* reads the value of *x* only after *A* has produced it, *B* is waiting for the event *e* to be notified by *A*.

vii. Note that, regardless of the execution order of the **par** statement, the example will correctly transfer the data from *A* to *B* and then terminate. The synchronization semantics ensure that the event notified by *A* is not lost.

viii. Note also that the synchronization semantics even allow for a thread to wake up itself. For example, a thread executing the statement sequence **notify** *e*; **wait** *e*; will receive the notified event himself and can continue its execution after the **wait** statement.

2.4.7 Exception handling

Purpose: Representation of exception handling

Synopsis:

```
exception_statement =
        try compound_statement exception_list_opt

exception_list_opt =
        <nothing>
        | exception_list

exception_list =
        exception
        | exception_list exception

exception =
        trap paren_event_list compound_statement
        | interrupt paren_event_list compound_statement

paren_event_list =
        event_list
        | '(' event_list ')'

event_list =
        event_identifier
        | event_list ',' event_identifier
        | event_list '||' event_identifier

event_identifier =
        identifier
        | edge_selector identifier
        | identifier edge_selector

edge_selector =
        rising
        | falling
```

Semantics:

(a) The **try-trap-interrupt** statement represents two types of exception handling in the regular control flow (specified by **try**), namely abortion (specified by **trap**) and in-

terrupt (specified by **interrupt**). Exception handling operates by use of events (see Section 2.2.5) or signals (see Section 2.2.6).

(b) The execution of the **try-trap-interrupt** statement starts with the execution of the compound statement block specified after the **try** keyword. It terminates with the completion of the execution of the **try** block, or with the completion of the execution of a **trap** handler.

(c) The **try** keyword enables exception handling for the execution of the compound statement block following the **try** keyword. Within a **try** block, a thread (and all its children) is sensitive to all events specified with the **trap** and **interrupt** handlers.

(d) When one or more events to which a thread is sensitive is notified, the execution of the thread (and all its children) is immediately suspended and a corresponding **trap** or **interrupt** handler is executed. The point of execution where a thread is suspended due to an exception is chosen non-deterministically.

(e) Within a **try-trap-interrupt** statement, the **interrupt** and **trap** handlers are prioritized in the order they are specified. Only the first specified exception, that matches any of the notified events, is executed.

(f) For hierarchically composed **try-trap-interrupt** statements, the outer (higher level) exception handlers take priority over the inner (lower level) exception handlers.

(g) An **interrupt** handler is executed upon notification of one or more events specified as arguments to the **interrupt** keyword. After the execution of the compound statement block corresponding to the **interrupt** handler, the suspended thread (and all its children) of the **try** block resumes its execution.

(h) A **trap** handler is executed upon notification of one or more events specified as arguments to the **trap** keyword. After the execution of the compound statement block corresponding to the **trap** handler, the execution of the **try-trap-interrupt** statement completes. The execution of the suspended thread (and all its children) of the **try** block is aborted.

(i) The statements in the compound statement blocks after the **try**, **trap** and **interrupt** keywords are restricted to at most one call to a *main* method of a behavior. No other statement type is allowed.

Example:

```
 1 behavior B;
 2
 3 behavior B_except(in event e1, in event e2)
 4 {
 5    B    b1, b2, b3;
 6
 7    void main(void)
 8       {
 9         try            { b1.main(); }
10         interrupt (e1) { b2.main(); }
11         trap      (e2) { b3.main(); }
12       }
13 };
```

Notes:

i. In a behavior sensitive to exceptions, interrupts and/or traps can occur at any time and at any place in the code. The point where the exception handler is called is chosen in non-deterministic manner. Note that this non-determinism in exception handling goes well along with the non-determinism in the concurrent execution semantics.

ii. Note that events are never stored or queued. Thus, an event targeted at a **wait** statement of a thread that is currently interrupted or trapped, will not reach the suspended **wait** statement. Also, an event triggering an exception handler with high priority will not reach any exception handler with lower priority. In other words, in a hierarchy of exception handlers, any set of simultaneously notified events will cause at most one exception (the one with the highest priority) to be serviced.

iii. Note that sensitivity to exception events only applies to the **try** block, not to any of the exception handlers. In other words, exception handlers cannot be interrupted or aborted by themselves or by other exception handlers specified with the same **try-trap-interrupt** statement.

iv. The example shows a behavior *B_except* demonstrating exception handling. Whenever event *e1* is notified during the execution of behavior *b1*, the execution of *b1* will be suspended and behavior *b2* is started. Then, when *b2* finishes, the execution of behavior *b1* is resumed right from the point where it was interrupted.

v. When an event *e2* occurs in the example during the execution of behavior *b1*, the execution of *b1* is aborted and the abortion handler *b3* is started. Then, when *b3* is completed, the execution of *B_except* completes as well.

vi. Note that a short cut notation for calling behavior instances exists (see Section 2.3.1). With the short notation, the **try-trap-interrupt** statement in the example can be reduced to **try**{*b1;*} **interrupt***(e1)*{*b2;*} **trap***(e2)*{*b3;*}.

vii. An abstract system reset can be modeled by a **try-trap** statement enclosed in an infinite loop.

2.4.8 Execution time

Purpose: Representation of execution time

Synopsis:

```
waitfor_statement =
        waitfor time ';'

time =
        constant_expression
```

Semantics:

(a) The **waitfor** statement represents the concept of execution time in a SpecC program.

(b) The execution of the **waitfor** statement with respect to other concurrent threads is defined by the time interval formalism described in Section 3.4.

(c) An abstract simulation algorithm for the semantics of **waitfor** is described in Section 3.6. This algorithm represents one valid implementation of execution time. Other valid implementations may exist.

(d) The **waitfor** statement suspends the current thread from execution for the specified amount of time. After the specified amount of time has passed, the thread can resume its execution.

(e) The argument specified for the **waitfor** statement must be of type time, or must be implicitly convertable to type time (see Section 2.2.8).

(f) The expression specified as argument for the **waitfor** statement is evaluated at the time the **waitfor** statement is reached.

(g) The evaluation of the argument of the **waitfor** statement must result in a non-negative value.

(h) If, during the execution of a **waitfor** statement, a thread is interrupted (see Section 2.4.7), then the total amount of execution time spent for the execution of the

interrupt handler is added to the amount of time specified as argument to the **wait-for** statement.

Example:

```
1  behavior B (out int i, out event e)
2  {
3     void main(void)
4     {
5        i = 0;
6        waitfor 10;
7
8        i = 1;
9        notify e;
10       waitfor 10;
11
12       // ...
13    }
14 };
```

Notes:

i. Execution time is logical time, in contrast to real time.

ii. Sometimes execution time is also referred to as simulation time or execution delay.

iii. The **waitfor** statement is the only statement in SpecC whose execution results in an increase of (simulation) time.

iv. The **waitfor** statement and the **wait** statement are the only non-composite statements in SpecC whose execution time can be greate than zero. All other statements execute in zero time.

v. Since time spent for the execution of interrupts is added to the amount specified in the argument, the execution of a **waitfor** statement takes at least the amount of time specified in the argument, or longer.

2.4.9 Timing constraints

Purpose: Representation of timing constraints

Synopsis:

```
timing_statement =
        do compound_statement timing '{' constraint_list_opt '}'

compound_statement =
        '{' '}'
        | '{' declaration_list '}'
        | '{' statement_list '}'
        | '{' declaration_list statement_list '}'

statement_list =
        statement
        | statement_list statement
        ...

statement =
        labeled_statement
        ...

labeled_statement =
        identifier_or_typedef_name ':' statement
        ...

constraint_list_opt =
        <nothing>
        | constraint_list

constraint_list =
        constraint
        | constraint_list constraint

constraint =
        range '(' any_name ';' any_name ';' time_opt ';' time_opt ')' ';'

time_opt =
        <nothing>
        | time

time =
```

```
constant_expression
```

Semantics:

(a) The **do-timing** statement specifies timing constraints on the execution of statements in a compound statement block.

(b) The **do-timing** statement consists of a compound statement block containing a set of labeled statements, and a set of **range** statements.

(c) A **range** statement specifies a timing constraint between a pair of labeled statements denoted by the two labels specified as the first and second argument. The labels used for a **range** statements must be defined within the compound statement block of the **do-timing** statement where the **range** statement is specified.

(d) The **range** statement specifies the timing constraint by a minimum (third argument) and maximum (forth argument) amount of time to be spent from the start of the execution of the statement denoted by the first label to the start of the execution of the statement denoted by the second label.

(e) The minimum and maximum times are specified as optional constant expressions of type time. If specified, these values must be evaluatable to constants at compile time.

(f) If left unspecified, the minimum time value defaults to negative infinity ($-\infty$), the maximum time value defaults to positive infinity ($+\infty$).

(g) The execution semantics of the compound statement block within a **do-timing** statement are the same as for any other compound statement block.

(h) For simulation, the **range** statements specified with a **do-timing** statement can be used for timing validation at runtime. The way, a simulator performs this constraint validation, is implementation dependent.

Example:

```
 1 channel C (
 2    inout bit[16] ABus,
 3    inout bit[ 8] DBus,
 4    out   bit[ 1] RMode,
 5    out   bit[ 1] WMode)
 6 {
 7    bit[8] ReadByte(bit[16] Address)
 8    {
 9       bit[7:0]  MyData;
10
11       do { t1:{ ABus = Address;
12                 waitfor(2);
13                }
14            t2:{ RMode = 1; WMode = 0;
15                 waitfor(12);
16                }
17            t3:{ waitfor(5);
18                }
19            t4:{ MyData = DBus;
20                 waitfor(5);
21                }
22            t5:{ ABus = 0;
23                 waitfor(2);
24                }
25            t6:{ RMode = 0; WMode = 0;
26                 waitfor(10);
27                }
28            t7:{
29                }
30          }
31       timing
32          { range(t1; t2;  0;   );
33            range(t1; t3; 10; 20);
34            range(t2; t3; 10; 20);
35            range(t3; t4;  0;   );
36            range(t4; t5;  0;   );
37            range(t5; t7; 10; 20);
38            range(t6; t7;  5; 10);
39           }
40       return(MyData);
41    }
42 };
```

Notes:

i. The semantics of a statement **range***(l1, l2, min, max)* is that the statement labeled *l1* is to be executed at least *min* time units before, but not more than *max* time units later than the statement labeled *l2*.

ii. The **do-timing** statement specifies constraints for the implementation (e.g. synthesis) of the design model. These constraints can also be used for timing validation during simulation of the model.

iii. For example, timing constraint validation can be performed as follows. During the execution of the compound statement block, the simulation runtime system collects time stamps at the execution of each timing label. The time stamps are then validated by comparison with the specified **range** constraints and any violation of the specified timings is reported to the user in form of a warning or error message.

iv. Typically, it is best if the way a simulator implements timing validation can be controlled by the user. The SpecC reference compiler and simulator, for example, implement **range** statements by calling a function *_scc_range_check* for each **range** statement. By default, the function *_scc_range_check* is provided automatically by the simulator and will, if the constraints are not met, abort the simulation with a suitable error message. However, the function *_scc_range_check* can also be defined by the user, in which case he is fully in control of timing validation.

v. The example shows the specification of a read protocol for a static RAM. The timing constraints specified with the protocol are listed in form of **range** statements. In the compound statement block, one valid instance of implementation of the protocol is shown by the **waitfor** statements which specify the execution time of each action in the protocol.

2.5 Other SpecC constructs

2.5.1 Libraries

Purpose: Representation and handling of design libraries

Synopsis:

```
import_definition =
        import string_literal_list ';'

string_literal_list =
        string
        | string_literal_list string
```

Semantics:

(a) The **import** declaration specifies the inclusion of an external design file into the current design.

(b) The string argument of the **import** declaration denotes the name of the design to be included.

(c) The search for the denoted design file in the file system is implementation dependent.

(d) An imported design must be a valid SpecC program in itself. In particular, an imported design cannot rely on declarations specified in the importing design.

(e) The format of an imported design file is implementation dependent.

(f) The declarations and definitions contained in an imported design are incorporated into the current design as if they were specified in the current design itself. The usual rules for redeclaration and redefinition of symbols apply.

(g) The **import** declaration can be hierarchical. An imported design can in turn contain other imported designs.

(h) The **import** declaration can be used multiple times for the same design. In this case, only the first **import** declaration is effective, all following ones are ignored.

Example:

```
1 #include <sim.sh>
2 #include <stdio.h>
3
4 import "i_send";
5 import "i_receive";
6 import "c_handshake";
7
8 import "c_semaphore";
```

Notes:

i. In contrast to the *#include* construct inherited from the C language, the **import** declaration automatically avoids multiple inclusions of the same file. There is no need to use *#ifdef*'s around a library file to avoid unwanted redefinitions.

ii. The **import** declaration is visible to the SpecC compiler and any tool. In particular, it is not eliminated by the C preprocessor as the *#include* construct is. Thus, **import** can be used by tools for code structuring purposes.

iii. The search for an import file typically involves appending a file suffix and searching along a defined import path for the file name.

iv. The file format of import files typically includes plain SpecC source code and precompiled binary files.

2.5.2 Persistent annotation

Purpose: Persistent annotation at specific objects

Synopsis:

```
any_declaration =
        ...
        | note_definition

any_definition =
        ...
        | note_definition

note_definition =
        note any_name '=' annotation ';'
        | note any_name '.' any_name '=' annotation ';'

annotation =
        constant_expression
        | '{' '}'
        | '{' annotation_list '}'

annotation_list =
        annotation
        | annotation_list ',' annotation

any_name =
        identifier
        | typedef_name
        | behavior_name
        | channel_name
        | interface_name
```

Semantics:

(a) The **note** definition attaches a persistent annotation globally to the design, or locally to a specified symbol, label, or user-defined type.

(b) The annotation consists of a key and a value. The key is the name of the annotation and identifies the annotation at its object. It is an error to define multiple annotations with the same key at the same object.

(c) The annotation value is either a single or composite constant. Each constant can be of any constant type or be a constant expression. For the latter, the constant expression is evaluated to a constant at compile time.

(d) Annotation keys have their own name space. There is no name conflict possible with the name spaces of symbols, labels or named user-defined types.

(e) In the first form, without an object specifier, the annotation is attached to the current scope. Valid scopes are the global scope, the class scope, the function or method scope, or the scope of a user-defined type.

(f) In the second form, with an object specifier, the annotation is attached to the named object. The object specifier preceeds the annotation key, separated by a dot.

(g) The annotated object is searched by its name in the following order. First, if the annotation is defined in function or method scope, the name is searched in the list of defined labels. If not found, the name is searched among the symbols defined in the current local scope, then among the named user-defined types in the current scope, and finally among any symbols in visible scope. The annotation is attached to the first match. It is an error if no match is found.

Example:

```
 1 /* C style comment, not persistent */
 2 // C++ style comment, not persistent
 3
 4 note Author     = "Rainer Doemer";
 5 note Date       = {{ 2002, 05, 15 }, { 10, 47, 49 }};
 6 note DateString = "Wed May 15 10:47:49 PDT 2002";
 7
 8 const int x = 42;
 9 struct S { int a, b; float f; };
10
11 note x.Size     = sizeof(x);
12 note S.Bits     = sizeof(struct S) * 8;
13
14 behavior B(in int a, out int b)
15 {
16   note Version  = 1.1;
```

```
17
18    void main(void)
19    {
20       l1: b = 2 * a;
21           waitfor(10);
22       l2: b = 3 * a;
23
24       note NumOps  = 3;
25       note l1.OpID = 1;
26       note l2.OpID = 3;
27    }
28 };
29 note B.Area = { 123.45, 67.89 };
```

Notes:

i. Derived from the C language, the SpecC language allows comments in the source code to annotate the design description. In particular, SpecC supports the same comment styles as C++, namely 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).

ii. Code comments are not persistent. This means, they will be eliminated in the preprocessing step by the C preprocessor. Thus, comments are not visible to the compiler or any tools and therefore cannot be used to store information beyond the language specification.

Persistent annotations specified by the **note** definition do not have this problem. They are visible to the compiler and tools and therefore can be conveniently use for storing additional information that is not included in the SpecC code.

iii. As described above, persistent annotations can be attached to the current scope. This way, global annotations (lines 4, 5 and 6 in the example), annotations at classes (line 16), annotations at methods (line 24), and annotations at user-defined types can be defined.

iv. Alternatively, the object to be annotated can be named explicitly. In the example, this style is used to define the annotations at variable *x* (line 11), structure *S* (line 12), and labels *l1* and *l2* (lines 25 and 26).

v. Annotation values may be plain or composite. This is similar to variable initializers which also can be plain or composite. In the example, the annotation *DateString* is a plain string constant, whereas the annotation *Date* consists of a pair of lists denoting the date (year, month, day) and time (hour, minute, second) separately.

Chapter 3

SpecC Execution Semantics

The execution semantics of the SpecC language are defined by use of a time interval formalism [9] which is described in Section 3.1 through Section 3.5. For completeness, an abstract simulation algorithm for the SpecC execution semantics is given in Section 3.6.

3.1 Time interval formalism

For each statement s in a SpecC program, a time interval $\langle T_{start}(s), T_{end}(s) \rangle$ is defined, where $T_{start}(s)$ and $T_{end}(s)$ denote the start and end times, respectively, of the execution of the statement s. For any time interval, the condition $T_{start}(s) < T_{end}(s)$ holds.

The execution time T_{exec} of a statement s is given by the length of the time interval, $T_{exec}(s) = T_{end}(s) - T_{start}(s)$. The execution time of any statement is always positive.

With the exception of the **wait** and **waitfor** statements, the execution time of any statement is an infinitesimal (very close to zero) number in terms of simulation time. Only the **wait** and **waitfor** statements, and composite statements that include **wait** and/or **waitfor** statements as sub-statements, can have an execution time greater than one simulation time unit.

3.2 Sequential execution

Sequential execution of statements is defined by ordered time intervals that do not overlap. Formally, for any sequence of statements $\langle s_1, s_2, \ldots s_n \rangle$, the following condition holds:

$\forall i \in \{1, 2, \ldots, n-1\} : T_{end}(s_i) \leq T_{start}(s_{i+1})$

For a sequentially composed statement, such as a function call or a call to a method of a behavior or channel, the composite time interval includes all time intervals of the sub-statements. In other words, the time intervals of all sub-statements lie within the time interval of the composite statement.

Formally, for a composite statement f consisting of sub-statements $s_1, s_2, \ldots s_n$, the following conditions hold: $\forall i \in \{1, 2, \ldots, n\} : T_{start}(f) \leq T_{start}(s_i) \wedge T_{end}(s_i) \leq T_{end}(f)$

Note that sequential statements are not necessarily executed continuously. In particular, gaps may exist between the end of one statement and the start of the following statement, as well as between the start (end) of a composite statement and the start (end) of its sub-statements. The presence and length of such gaps are non-deterministic.

Example: Figure 3.1 shows an example of a composite behavior *B* that is composed of the sequential execution of three child behaviors *a*, *b* and *c*.

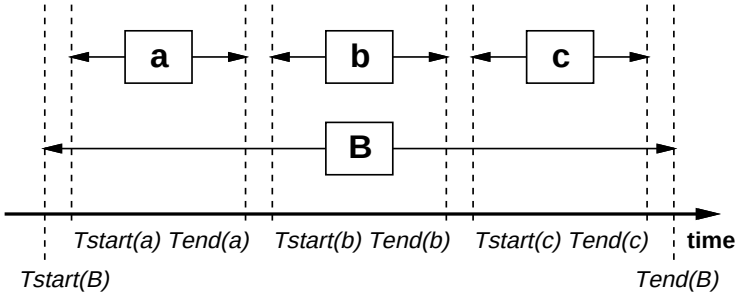

Figure 3.1: Time interval example for sequential execution.

The time interval formalism for this example derives the following equations:

$T_{start}(B) \leq T_{start}(a) < T_{end}(a) \leq T_{start}(b) < T_{end}(b) \leq T_{start}(c) < T_{end}(c) \leq T_{end}(B)$

3.3 Concurrent execution

Concurrent execution of statements, specified by either the **par** or **pipe** statements, is defined by time intervals that have the same start and end times as the **par** or **pipe** statement, respectively. In particular, the child behaviors invoked by a **par** or **pipe** statement begin and terminate their execution at the same time.

Formally, for a **par** or **pipe** statement p that executes the child behaviors $p_1, p_2, \ldots p_n$ concurrently, the following equations hold:

$\forall i \in \{1, 2, \ldots, n\} : T_{start}(p) = T_{start}(p_i) \wedge T_{end}(p_i) = T_{end}(p)$

Note that again a non-deterministic gap may exist between the start (end) time of a concurrent child behavior and the start (end) time of its statements. Therefore, it is possible but not necessary that the statements of concurrent child behaviors are actually executed in parallel.

As a result, concurrent execution may be implemented truly in parallel, or by portion-wise sequential execution where the order and size of the portions is undefined. This, in particular, includes the possibility of using preemptive execution. No atomicity is guaranteed for the execution of any portion of concurrent code. In other words, except for the time interval equations defined above, concurrent execution is non-deterministic.

Example: Figure 3.2 shows an example of a composite behavior *B* that is composed of the concurrent execution of two child behaviors *a* and *b*. The child behaviors *a* and *b* in turn consist of the statements *a*1 and *a*2, and *b*1 and *b*2, respectively.

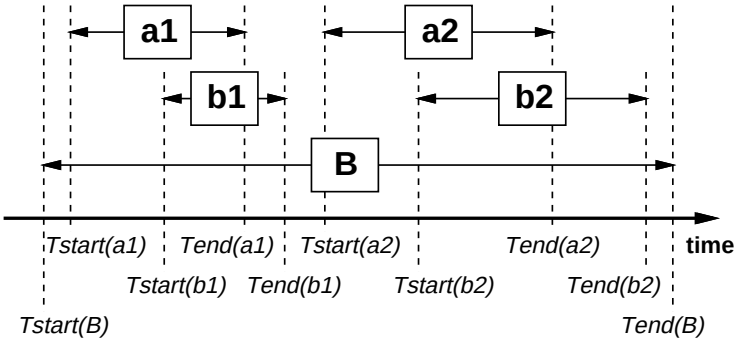

Figure 3.2: Time interval example for concurrent execution.

The time interval formalism for this example derives the following equations:

$T_{start}(B) = T_{start}(a) = T_{start}(b)$

$T_{end}(B) = T_{end}(a) = T_{end}(b)$

$T_{start}(B) \leq T_{start}(a1) < T_{end}(a1) \leq T_{start}(a2) < T_{end}(a2) \leq T_{end}(B)$

$T_{start}(B) \leq T_{start}(b1) < T_{end}(b1) \leq T_{start}(b2) < T_{end}(b2) \leq T_{end}(B)$

3.4 Simulation time

The concept of simulation time in SpecC is supported by the use of the **waitfor** statement. The execution of a **waitfor** statement suspends the current thread from further execution for the amount of simulation time specified as an argument to the **waitfor** statement.

For the time interval of a **waitfor** statement w with argument d, the end time $T_{end}(w)$ is given by adding the specified delay d to the start time $T_{start}(w)$. Formally,

$T_{start}(w) + d \leq T_{end}(w)$

In other words, the execution time of a **waitfor** statement is specified explicitly as its argument which, as defined in Section 2.4.8, must be a non-negative, integral value of type time. As such, the execution time of a **waitfor** statement is not restricted to an infinitesimal amount of time as for ordinary statements. Instead, it extends to an integral amount of simulation time.

In case the execution of a **waitfor** statement is interrupted due to activation of any **interrupt** handlers (see Section 2.4.7), then the execution time of the **waitfor** statement is prolonged by the amount of time serving the interrupts. More specifically, the total execution time $d_{interrupt}$ spent in any interrupt handlers is added to the delay $d_{argument}$ specified with the **waitfor** statement. Formally, $d = d_{argument} + d_{interrupt}$.

In summary, for a **waitfor** statement w with argument d, whose execution starts at simulation time t, the following equations hold:

$t \leq T_{start}(w) < t + 1$

$t + d \leq T_{end}(w) < t + d + 1$ where $d = d_{argument} + d_{interrupt}$

Example: Figure 3.3 illustrates the relation between execution time and simulation time. The example shows three sequential statements a, w and b, where a and b are ordinary

statements and *w* is a **waitfor** statement with argument 10. Further, the example assumes that statement *a* is executed at simulation time 0.

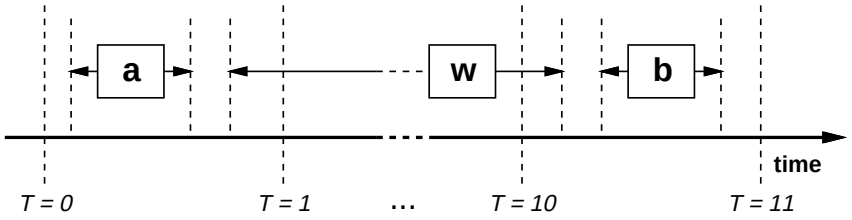

Figure 3.3: Time interval example for simulation time.

The time interval formalism for this example derives the following equations:

$0 \leq T_{start}(a) < T_{end}(a) \leq T_{start}(w) < 1$

$10 \leq T_{end}(w) \leq T_{start}(b) < T_{end}(b) < 11$

3.5 Synchronization

In order for concurrent threads to be cooperative, the threads need to be synchronized at points of communication. Synchronization of concurrent threads can be specified by use of the **notify** (or **notifyone**) and **wait** statements. As defined in Section 2.4.6, a **wait** statement suspends the current thread from further execution until a specified event is triggered by the execution of a **notify** (or **notifyone**) statement.

In order to define this synchronization mechanism in terms of the time interval formalism, a notify-wait pair $\langle n, w \rangle$ is defined as a **notify** (or **notifyone**) statement *n* and a corresponding **wait** statement *w*, where an event triggered by the **notify** (or **notifyone**) statement *n* reaches the **wait** statement *w*.

Note that, whether or not an event triggered by a **notify** (or **notifyone**) statement actually reaches a **wait** statement, is not defined within the time interval formalism. However, in Section 3.6 one valid simulation algorithm is defined that determines how a valid notify-wait pair is found.

Formally, the following equation holds for a notify-wait pair $\langle n, w \rangle$:

$T_{end}(n) \leq T_{end}(w)$

Example: Figure 3.4 illustrates the time interval formalism for synchronization. The example shows two concurrent threads *a* and *b*, where *a* consists of the sequential statements *a*1, *w* and *a*2, and *b* consists of the sequential statements *b*1, *n* and *b*2. Further, the example assumes that *a*1, *a*2, *b*1 and *b*2 are ordinary statements, whereas *n* and *w* are **notify** and **wait** statements, respectively, that form a notify-wait pair.

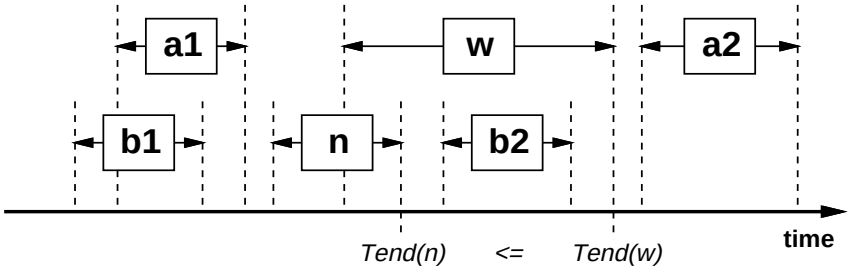

Figure 3.4: Time interval example for synchronization.

The time interval formalism for this example derives the following equations:
$T_{start}(a1) < T_{end}(a1) \leq T_{start}(w) < T_{end}(w) \leq T_{start}(a2) < T_{end}(a2)$
$T_{start}(b1) < T_{end}(b1) \leq T_{start}(n) < T_{end}(n) \leq T_{start}(b2) < T_{end}(b2)$
$T_{end}(n) \leq T_{end}(w)$

3.6 Abstract simulation algorithm

In order to summarize the execution semantics of the SpecC language, this section describes an abstract simulation algorithm for SpecC programs.

The algorithm defined in the following is one valid implementation of the SpecC execution semantics. Other valid implementations may exist.

Definitions:

(a) The set T represents the set of threads that are active during the program execution. In the beginning, T contains only the root thread t_{root}. When t_{root} is completed, the execution of the program terminates.

(b) The number of threads in T changes during the program execution due to creation of new threads and termination of completed threads. At a **par** (or **pipe**) statement,

the specified concurrent child behaviors are added as threads to T. When the child behaviors have completed their execution, they are taken out of T.

(c) At any time, each thread $t \in T$ belongs to exactly one subset of T. More specifically, T is composed of the three subsets T_{ready}, T_{wait} and $T_{wait\,for}$ which do not overlap.

Formally, $T = T_{ready} \cup T_{wait} \cup T_{wait\,for}$, where

$T_{ready} \cap T_{wait} = \emptyset \quad \wedge \quad T_{wait} \cap T_{wait\,for} = \emptyset \quad \wedge \quad T_{ready} \cap T_{wait\,for} = \emptyset$

In the beginning, $T_{ready} = \{t_{root}\}$, $T_{wait} = \emptyset$, $T_{wait\,for} = \emptyset$

(d) The set N represents the set of notified events during the program execution. Events notified by **notify** statements are added to N. Delivered or expired events are taken out of N. In the beginning, $N = \emptyset$.

(e) The variable t_{now} holds the current simulation time. In the beginning, $t_{now} = 0$.

(f) For each thread $t \in T_{wait\,for}$, a function $t_{wakeup}(t)$ determines the wakeup time of the thread. The wakeup time is computed at the time a **waitfor** statement is reached. For a **waitfor** statement with argument d, $t_{wakeup}(t) = t_{now} + d$.

Algorithm:

Step 1 : Start.

Step 2 : Select one thread $t_{run} \in T_{ready}$, execute t_{run}.

Step 3 : At **notify** statement, add notified events to N, go to step 6.

Step 4 : At **wait** statement, move t_{run} from T_{ready} to T_{wait}, go to step 6.

Step 5 : At **waitfor** statement, move t_{run} from T_{ready} to $T_{wait\,for}$, go to step 6.

Step 6 : If $T_{ready} \neq \emptyset$ then go to step 2.

Step 7 : Move all $t \in T_{wait}$ waiting for events $e \in N$ to T_{ready}.

Step 8 : Set $N = \emptyset$.

Step 9 : If $T_{ready} \neq \emptyset$ then go to step 2.

Step 10 : Set $t_{now} = \min\{t_{wakeup}(t) \mid t \in T_{wait\,for}\}$.

Step 11 : Move all $t \in T_{wait\,for}$ where $t_{wakeup}(t) = t_{now}$ to T_{ready}.

Step 12 : If $T_{ready} \neq \emptyset$ then go to step 2.

Step 13 : Stop.

Notes:

i. The abstract simulation algorithm starts with Step 1. As defined above, all variables are set to their initial values: $T = T_{ready} = \{t_{root}\}$, $T_{wait} = \emptyset$, $T_{wait\,for} = \emptyset$, $N = \emptyset$, $t_{now} = 0$.

ii. In Step 2, one thread is chosen for execution out of all threads in the ready queue T_{ready}. Note that, at any time, this algorithm runs only one thread. This is a valid choice for implementing the potential concurrency, as defined in Section 3.3.

A different implementation, that also would be valid according to Section 3.3, could actually choose to execute some or even all threads out of T_{ready} in parallel. However, in this case care must be taken when accessing the variables of the algorithm because those are shared among all the threads.

iii. In Step 3, events triggered by **notify** statements are collected in N. Although not shown in the algorithm, events triggered by **notifyone** statements would be handled in a very similar manner.

iv. In Step 4, threads executing a **wait** statement are suspended from further execution by putting them into the wait queue T_{wait}.

v. In Step 5, **waitfor** statements are handled in the same fashion as the **wait** statements in Step 4. Threads executing a **waitfor** statement are suspended from further execution by putting them into the waitfor queue $T_{wait\,for}$. Note that in this case a wakeup time for the thread is computed so that it can be resumed after the specified time period.

vi. Step 6 defines the innermost loop of the simulation algorithm, which is called the synchronization cycle. This synchronization cycle ensures that all threads are exe-

cuted until they hit a **wait** or **waitfor** statement. Only if no thread is available any more for execution, Step 7 is reached.

vii. In Step 7, all notified events are delivered to the waiting threads. Threads from the wait queue, whose events were notified, are resumed. Other threads stay in the wait queue.

Note that in this step also updating of **signal** and **buffered** variables would take place, as well as exception handling. These tasks, however, are left out in the algorithm for the reason of simplicity.

viii. In Step 8, the set of notified events is cleared. Events that could not be delivered in Step 7, expire at this point.

ix. Step 9 defines the second loop of the simulation algorithm. All the threads, that have received events they were waiting for, can resume their execution.

x. In Step 10 the simulation time t_{now} is updated. It is increased by the minimum amount of time that any threads in the waitfor queue still have to wait for.

xi. In Step 11, the threads whose wakeup time has been reached, are enabled for execution again.

xii. Step 12 defines the outermost loop of the simulation algorithm. If any threads could be awakened in Step 11, they can now resume their execution.

xiii. In Step 13, the simulation algorithm terminates. Note that at this point no thread is available for further execution any more because both the ready queue T_{ready} and the waitfor queue $T_{wait\,for}$ are empty. The wait queue T_{wait}, however, still contains one or more threads that are waiting for events, but since no other thread is available any more to notify any events, the simulation is stuck in a deadlock situation and must terminate.

Appendix A

SpecC Grammar

In the following, the complete grammar of the SpecC language version 2.0 is listed in the format of an extended Backus-Naur form (EBNF).

A.1 Lexical elements

A.1.1 Lexical rules

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

```
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}([uU][lL]|[lL][uU])
octinteger_ul     {octinteger}([uU][lL]|[lL][uU])
hexinteger_ul     {hexinteger}([uU][lL]|[lL][uU])
decinteger_ll     {decinteger}[lL][lL]
octinteger_ll     {octinteger}[lL][lL]
hexinteger_ll     {hexinteger}[lL][lL]
decinteger_ull    {decinteger}([uU][lL][lL]|[lL][lL][uU])
octinteger_ull    {octinteger}([uU][lL][lL]|[lL][lL][uU])
hexinteger_ull    {hexinteger}([uU][lL][lL]|[lL][lL][uU])
octchar           "\\"{octdigit}{1,3}
hexchar           "\\x"{hexdigit}+
exponent          [eE][+-]?{integer}
fraction          {integer}
float1            {integer}"."{fraction}?({exponent})?
float2            "."{fraction}({exponent})?
float3            {integer}{exponent}
floating          {float1}|{float2}|{float3}
float_f           {floating}[fF]
float_l           {floating}[lL]
bitvector         {binary}[bB]
bitvector_u       {binary}([uU][bB]|[bB][uU])
```

A.1.2 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.1.3 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 */
```

Strings are character sequences surrounded by quotation marks. Two subsequent strings, only separated by whitespace, are concatenated, in other words, they are treated the same way as one single string.

A.1.4 White space and preprocessor directives

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 actual SpecC parser.

```
{newline}       /* skip */
{whitespace}    /* skip */
```

A.1.5 Keywords

The SpecC language recognizes the following ANSI-C keywords:

auto, **break**, **case**, **char**, **const**, **continue**, **default**, **do**, **double**, **else**, **enum**, **extern**, **float**, **for**, **goto**, **if**, **int**, **long**, **register**, **return**, **short**, **signed**, **sizeof**, **static**, **struct**, **switch**, **typedef**, **union**, **unsigned**, **void**, **volatile**, **while**.

In addition, the following SpecC keywords are recognized:

behavior, **bit**, **bool**, **buffered**, **channel**, **event**, **falling**, **false**, **fsm**, **fsmd**, **implements**, **import**, **in**, **inout**, **interface**, **interrupt**, **note**, **notify**, **notifyone**, **out**, **par**, **pipe**, **piped**, **range**, **rising**, **signal**, **this**, **timing**, **trap**, **true**, **try**, **wait**, **waitfor**.

For future extensions, the following tokens are reserved. These keywords must not be used as identifiers in any SpecC program.

and, **and_eq**, **asm**, **bitand**, **bitor**, **catch**, **class**, **compl**, **const_cast**, **delete**, **dynamic_cast**, **explicit**, **export**, **friend**, **inline**, **mutable**, **namespace**, **new**, **not**, **not_eq**,

operator, **or**, **or_eq**, **private**, **protected**, **public**, **reinterpret_cast**, **static_cast**, **template**, **throw**, **typeid**, **typename**, **using**, **virtual**, **wchar_t**, **xor**, **xor_eq**, **fix**.

A.1.6 Token with values

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

```
identifier =
        {identifier}

typedef_name =
        {identifier}

behavior_name =
        {identifier}

channel_name =
        {identifier}

interface_name =
        {identifier}

integer =
        {decinteger}
        | {octinteger}
        | {hexinteger}
        | {decinteger_u}
        | {octinteger_u}
        | {hexinteger_u}
        | {decinteger_l}
        | {octinteger_l}
        | {hexinteger_l}
        | {decinteger_ul}
        | {octinteger_ul}
        | {hexinteger_ul}
        | {decinteger_ll}
        | {octinteger_ll}
        | {hexinteger_ll}
        | {decinteger_ull}
        | {octinteger_ull}
        | {hexinteger_ull}

floating =
```

```
        {floating}
        | { float_f }
        | { float_l }

character =
        {character}

string =
        {string}

bitvector =
        {bitvector}
        | { bitvector_u }
```

A.2 Constants

```
constant =
        integer
        | floating
        | character
        | false
        | true
        | bitvector
        | string_literal_list

string_literal_list =
        string
        | string_literal_list string
```

A.3 Expressions

```
primary_expression =
        identifier
        | constant
        | '(' comma_expression ')'
        | this

postfix_expression =
        primary_expression
```

```
        | postfix_expression '[' comma_expression ']'
        | postfix_expression '(' ')'
        | postfix_expression '(' argument_expression_list ')'
        | postfix_expression '.' member_name
        | postfix_expression '->' member_name
        | postfix_expression '++'
        | postfix_expression '--'
        | postfix_expression '[' constant_expression ':'
              constant_expression ']'

member_name =
        identifier
        | typedef_or_class_name

argument_expression_list =
        assignment_expression
        | argument_expression_list ',' assignment_expression

unary_expression =
        postfix_expression
        | '++' unary_expression
        | '--' unary_expression
        | unary_operator cast_expression
        | sizeof unary_expression
        | sizeof '(' type_name ')'

unary_operator =
        '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'

cast_expression =
        unary_expression
        | '(' type_name ')' cast_expression

concat_expression =
        cast_expression
        | concat_expression '@' cast_expression

multiplicative_expression =
        concat_expression
        | multiplicative_expression '*' concat_expression
```

```
        | multiplicative_expression '/' concat_expression
        | multiplicative_expression '%' concat_expression

additive_expression =
        multiplicative_expression
        | additive_expression '+' multiplicative_expression
        | additive_expression '-' multiplicative_expression

shift_expression =
        additive_expression
        | shift_expression '<<' additive_expression
        | shift_expression '>>' additive_expression

relational_expression =
        shift_expression
        | relational_expression '<' shift_expression
        | relational_expression '>' shift_expression
        | relational_expression '<=' shift_expression
        | relational_expression '>=' shift_expression

equality_expression =
        relational_expression
        | equality_expression '==' relational_expression
        | equality_expression '!=' relational_expression

and_expression =
        equality_expression
        | and_expression '&' equality_expression

exclusive_or_expression =
        and_expression
        | exclusive_or_expression '^' and_expression

inclusive_or_expression =
        exclusive_or_expression
        | inclusive_or_expression '|' exclusive_or_expression

logical_and_expression =
        inclusive_or_expression
        | logical_and_expression '&&' inclusive_or_expression

logical_or_expression =
        logical_and_expression
        | logical_or_expression '||' logical_and_expression
```

```
conditional_expression =
        logical_or_expression
        | logical_or_expression '?' comma_expression ':'
                conditional_expression

assignment_expression =
        conditional_expression
        | unary_expression assignment_operator assignment_expression

assignment_operator =
        '='
        | '*='
        | '/='
        | '%='
        | '+='
        | '-='
        | '<<='
        | '>>='
        | '&='
        | '^='
        | '|='

comma_expression =
        assignment_expression
        | comma_expression ',' assignment_expression

constant_expression =
        conditional_expression

comma_expression_opt =
        <nothing>
        | comma_expression
```

A.4 Declarations

```
declaration =
        sue_declaration_specifier ';'
        | sue_type_specifier ';'
        | declaring_list ';'
        | default_declaring_list ';'

default_declaring_list =
        declaration_qualifier_list identifier_declarator initializer_opt
```

```
        | type_qualifier_list identifier_declarator initializer_opt
        | default_declaring_list ',' identifier_declarator initializer_opt
        | signal_class declaration_qualifier_list identifier_declarator
              initializer_opt
        | signal_class type_qualifier_list identifier_declarator
              initializer_opt

declaring_list =
        declaration_specifier declarator initializer_opt
        | type_specifier declarator initializer_opt
        | declaring_list ',' declarator initializer_opt
        | signal_class declaration_specifier declarator initializer_opt
        | signal_class type_specifier declarator initializer_opt

signal_class_opt =
        <nothing>
        | signal_class

signal_class =
        signal
        | buffered
        | buffered '[' clock_specifier ']'
        | buffered '[' clock_specifier ';' reset_signal_opt ']'

declaration_specifier =
        basic_declaration_specifier
        | sue_declaration_specifier
        | typedef_declaration_specifier

type_specifier =
        basic_type_specifier
        | sue_type_specifier
        | typedef_type_specifier

declaration_qualifier_list =
        storage_class
        | type_qualifier_list storage_class
        | declaration_qualifier_list declaration_qualifier

type_qualifier_list =
        type_qualifier
        | type_qualifier_list type_qualifier

declaration_qualifier =
        storage_class
```

```
        | type_qualifier

type_qualifier =
        const
        | volatile

basic_declaration_specifier =
        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 =
        basic_type_name
        | type_qualifier_list basic_type_name
        | basic_type_specifier type_qualifier
        | basic_type_specifier basic_type_name

sue_declaration_specifier =
        declaration_qualifier_list elaborated_type_name
        | sue_type_specifier storage_class
        | sue_declaration_specifier declaration_qualifier

sue_type_specifier =
        elaborated_type_name
        | type_qualifier_list elaborated_type_name
        | sue_type_specifier type_qualifier

typedef_declaration_specifier =
        typedef_type_specifier storage_class
        | declaration_qualifier_list typedef_name
        | typedef_declaration_specifier declaration_qualifier

typedef_type_specifier =
        typedef_name
        | type_qualifier_list typedef_name
        | typedef_type_specifier type_qualifier

storage_class =
        typedef
        | extern
        | static
        | auto
        | register
        | piped
```

```
basic_type_name =
        int
        | char
        | short
        | long
        | float
        | double
        | signed
        | unsigned
        | void
        | bool
        | bit '[' constant_expression ':' constant_expression ']'
        | bit '[' constant_expression ']'
        | event

elaborated_type_name =
        aggregate_name
        | enum_name

aggregate_name =
        aggregate_key '{' member_declaration_list '}'
        | aggregate_key identifier_or_typedef_name '{'
              member_declaration_list '}'
        | aggregate_key identifier_or_typedef_name

aggregate_key =
        struct
        | union

member_declaration_list =
        member_declaration
        | member_declaration_list member_declaration

member_declaration =
        member_declaring_list ';'
        | member_default_declaring_list ';'
        | note_definition

member_default_declaring_list =
        type_qualifier_list member_identifier_declarator
        | member_default_declaring_list ',' member_identifier_declarator

member_declaring_list =
        type_specifier member_declarator
```

```
        | member_declaring_list ',' member_declarator

member_declarator =
        declarator bit_field_size_opt
        | bit_field_size

member_identifier_declarator =
        identifier_declarator bit_field_size_opt
        | bit_field_size

bit_field_size_opt =
        <nothing>
        | bit_field_size

bit_field_size =
        ':' constant_expression

enum_name =
        enum '{' enumerator_list '}'
        | enum identifier_or_typedef_name '{' enumerator_list '}'
        | enum identifier_or_typedef_name

enumerator_list =
        identifier_or_typedef_name enumerator_value_opt
        | enumerator_list ',' identifier_or_typedef_name
                enumerator_value_opt

enumerator_value_opt =
        <nothing>
        | '=' constant_expression

parameter_type_list =
        parameter_list
        | parameter_list ',' '...'

parameter_list =
        parameter_declaration
        | parameter_list ',' parameter_declaration
        | interface_parameter
        | parameter_list ',' interface_parameter

parameter_declaration =
        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 =
        identifier
        | typedef_or_class_name

type_name =
        type_specifier
        | type_specifier abstract_declarator
        | type_qualifier_list
        | type_qualifier_list abstract_declarator

initializer_opt =
        <nothing>
        | '=' initializer

initializer =
        '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        | constant_expression

initializer_list =
        initializer
        | initializer_list ',' initializer
```

A.5 Classes

```
spec_c_definition =
        import_definition
        | behavior_declaration
        | behavior_definition
        | channel_declaration
        | channel_definition
```

```
        | interface_declaration
        | interface_definition
        | note_definition

import_definition =
        import string_literal_list ';'

behavior_declaration =
        behavior_specifier port_list_opt implements_interface_opt ';'

behavior_definition =
        behavior_specifier port_list_opt implements_interface_opt
                '{' internal_definition_list_opt '}' ';'

behavior_specifier =
        behavior identifier

channel_declaration =
        channel_specifier port_list_opt implements_interface_opt ';'

channel_definition =
        channel_specifier port_list_opt implements_interface_opt
                '{' internal_definition_list_opt '}' ';'

channel_specifier =
        channel identifier

port_list_opt =
        <nothing>
        | '(' ')'
        | '(' port_list ')'

port_list =
        port_declaration
        | port_list ',' port_declaration

port_declaration =
        port_direction signal_class_opt parameter_declaration
        | interface_parameter

port_direction =
        <nothing>
        | in
        | out
        | inout
```

```
interface_parameter =
        interface_name
        | interface_name identifier

implements_interface_opt =
        <nothing>
        | implements interface_list

interface_list =
        interface_name
        | interface_list ',' interface_name

internal_definition_list_opt =
        <nothing>
        | internal_definition_list

internal_definition_list =
        internal_definition
        | internal_definition_list internal_definition

internal_definition =
        function_definition
        | declaration
        | instantiation
        | note_definition

instantiation =
        instance_declaring_list ';'

instance_declaring_list =
        behavior_or_channel instance_declarator
        | instance_declaring_list ',' instance_declarator

instance_declarator =
        identifier port_mapping_list_opt
        | typedef_or_class_name port_mapping_list_opt

behavior_or_channel =
        behavior_name
        | channel_name

port_mapping_list_opt =
        <nothing>
        | '(' port_mapping_list ')'
```

```
port_mapping_list =
        port_mapping_opt
        | port_mapping_list ',' port_mapping_opt

port_mapping_opt =
        <nothing>
        | port_mapping

port_mapping =
        bit_slice
        | port_mapping '@' bit_slice

bit_slice =
        constant
        | '(' constant_expression ')'
        | identifier
        | identifier '[' constant_expression ':' constant_expression ']'
        | identifier '[' constant_expression ']'

interface_declaration =
        interface_specifier ';'

interface_definition =
        interface_specifier '{' internal_declaration_list_opt '}' ';'

interface_specifier =
        interface identifier

internal_declaration_list_opt =
        <nothing>
        | internal_declaration_list

internal_declaration_list =
        internal_declaration
        | internal_declaration_list internal_declaration

internal_declaration =
        declaration
        | note_definition

note_definition =
        note any_name '=' annotation ';'
        | note any_name '.' any_name '=' annotation ';'
```

```
annotation =
        constant_expression
        | '{' '}'
        | '{' annotation_list '}'

annotation_list =
        annotation
        | annotation_list ',' annotation

typedef_or_class_name =
        typedef_name
        | behavior_name
        | channel_name
        | interface_name

any_name =
        identifier
        | typedef_name
        | behavior_name
        | channel_name
        | interface_name
```

A.6 Statements

```
statement =
        labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        | spec_c_statement

labeled_statement =
        identifier_or_typedef_name ':' statement
        | case constant_expression ':' statement
        | default ':' statement

compound_statement =
        '{' '}'
        | '{' declaration_list '}'
        | '{' statement_list '}'
        | '{' declaration_list statement_list '}'
```

```
declaration_list =
        declaration
        | declaration_list declaration
        | note_definition
        | declaration_list note_definition

statement_list =
        statement
        | statement_list statement
        | statement_list note_definition

expression_statement =
        comma_expression_opt ';'

selection_statement =
        if '(' comma_expression ')' statement
        | if '(' comma_expression ')' statement else statement
        | switch '(' comma_expression ')' statement

iteration_statement =
        while '(' comma_expression_opt ')' statement
        | do statement while '(' comma_expression ')' ';'
        | for '(' comma_expression_opt ';' comma_expression_opt ';'
              comma_expression_opt ')' statement

jump_statement =
        goto identifier_or_typedef_name ';'
        | continue ';'
        | break ';'
        | return comma_expression_opt ';'

spec_c_statement =
        concurrent_statement
        | fsm_statement
        | fsmd_statement
        | exception_statement
        | timing_statement
        | wait_statement
        | waitfor_statement
        | notify_statement

concurrent_statement =
        par compound_statement
        | pipe compound_statement
```

```
        | pipe '(' comma_expression_opt ';' comma_expression_opt
               ';' comma_expression_opt ')' compound_statement

fsm_statement =
        fsm '{' '}'
        | fsm '{' transition_list '}'

transition_list =
        transition
        | transition_list transition

transition =
        state ':'
        | state ':' cond_branch_list
        | state ':' '{' '}'
        | state ':' '{' cond_branch_list '}'

state =
        identifier
        | identifier compound_statement

cond_branch_list =
        cond_branch
        | cond_branch_list cond_branch

cond_branch =
        if '(' comma_expression ')' goto identifier ';'
        | goto identifier ';'
        | if '(' comma_expression ')' break ';'
        | break ';'

fsmd_statement =
        fsmd '(' fsmd_head ')' fsmd_body

fsmd_head =
        clock_specifier
        | clock_specifier ';' sensitivity_list_opt
        | clock_specifier ';' sensitivity_list_opt ';' reset_signal_opt

clock_specifier =
        event_list
        | constant
        | '(' time ')'

sensitivity_list_opt =
```

```
        <nothing>
        | event_list

reset_signal_opt =
        <nothing>
        | identifier
        | '!' identifier

fsmd_body =
        '{' '}'
        | '{' declaration_list '}'
        | '{' reset_state '}'
        | '{' declaration_list reset_state '}'
        | '{' default_action '}'
        | '{' declaration_list default_action '}'
        | '{' reset_state default_action '}'
        | '{' declaration_list reset_state default_action '}'
        | '{' fsmd_state_list '}'
        | '{' declaration_list fsmd_state_list '}'
        | '{' reset_state fsmd_state_list '}'
        | '{' declaration_list reset_state fsmd_state_list '}'
        | '{' default_action fsmd_state_list '}'
        | '{' declaration_list default_action fsmd_state_list '}'
        | '{' reset_state default_action fsmd_state_list '}'
        | '{' declaration_list reset_state default_action fsmd_state_list '}'

reset_state =
        if '(' comma_expression ')' action

default_action =
        action

fsmd_state_list =
        fsmd_state
        | fsmd_state_list fsmd_state

fsmd_state =
        identifier_or_typedef_name ':' action

action =
        '{' '}'
        | '{' declaration_list '}'
        | '{' rtl_statement_list '}'
        | '{' declaration_list rtl_statement_list '}'
```

```
rtl_statement_list =
        rtl_statement
        | rtl_statement_list  rtl_statement
        | rtl_statement_list  note_definition

rtl_statement =
        rtl_labeled_statement
        | rtl_compound_statement
        | expression_statement
        | rtl_selection_statement
        | rtl_jump_statement

rtl_labeled_statement =
        case constant_expression ':' rtl_statement
        | default ':' rtl_statement

rtl_compound_statement =
        '{' '}'
        | '{' declaration_list '}'
        | '{' rtl_statement_list '}'
        | '{' declaration_list  rtl_statement_list '}'

rtl_selection_statement =
        if '(' comma_expression ')' rtl_statement
        | if '(' comma_expression ')' rtl_statement  else  rtl_statement
        | switch '(' comma_expression ')' rtl_statement

rtl_jump_statement =
        goto identifier_or_typedef_name ';'
        | break ';'

exception_statement =
        try compound_statement  exception_list_opt

exception_list_opt =
        <nothing>
        | exception_list

exception_list =
        exception
        | exception_list  exception

exception =
        trap  paren_event_list  compound_statement
        | interrupt  paren_event_list  compound_statement
```

```
paren_event_list =
        event_list
        | '(' event_list ')'

event_list =
        event_identifier
        | event_list ',' event_identifier
        | event_list '||' event_identifier

paren_and_event_list =
        and_event_list
        | '(' and_event_list ')'

and_event_list =
        event_identifier '&&' event_identifier
        | and_event_list '&&' event_identifier

event_identifier =
        identifier
        | edge_selector identifier
        | identifier edge_selector

edge_selector =
        rising
        | falling

timing_statement =
        do compound_statement timing '{' constraint_list_opt '}'

constraint_list_opt =
        <nothing>
        | constraint_list

constraint_list =
        constraint
        | constraint_list constraint

constraint =
        range '(' any_name ';' any_name ';' time_opt ';' time_opt ')' ';'

time_opt =
        <nothing>
        | time
```

```
time =
        constant_expression

wait_statement =
        wait paren_event_list ';'
        | wait paren_and_event_list ';'

waitfor_statement =
        waitfor time ';'

notify_statement =
        notify paren_event_list ';'
        | notifyone paren_event_list ';'
```

A.7 External definitions

```
translation_unit =
        <nothing>
        | external_definition_list

external_definition_list =
        external_definition
        | external_definition_list external_definition

external_definition =
        function_definition
        | declaration
        | spec_c_definition

function_definition =
        identifier_declarator compound_statement
        | declaration_specifier declarator compound_statement
        | type_specifier declarator compound_statement
        | declaration_qualifier_list identifier_declarator
              compound_statement
        | type_qualifier_list identifier_declarator
              compound_statement

declarator =
        identifier_declarator
        | typedef_declarator

typedef_declarator =
```

```
        paren_typedef_declarator
        | parameter_typedef_declarator

parameter_typedef_declarator =
        typedef_or_class_name
        | typedef_or_class_name postfixing_abstract_declarator
        | clean_typedef_declarator

clean_typedef_declarator =
        clean_postfix_typedef_declarator
        | '*' parameter_typedef_declarator
        | '*' type_qualifier_list parameter_typedef_declarator

clean_postfix_typedef_declarator =
        '(' clean_typedef_declarator ')'
        | '(' clean_typedef_declarator ')'
                postfixing_abstract_declarator

paren_typedef_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 =
        '(' paren_typedef_declarator ')'
        | '(' simple_paren_typedef_declarator
                postfixing_abstract_declarator ')'
        | '(' paren_typedef_declarator ')'
                postfixing_abstract_declarator

simple_paren_typedef_declarator =
        typedef_or_class_name
        | '(' simple_paren_typedef_declarator ')'

identifier_declarator =
        unary_identifier_declarator
        | paren_identifier_declarator

unary_identifier_declarator =
        postfix_identifier_declarator
        | '*' identifier_declarator
        | '*' type_qualifier_list identifier_declarator
```

```
postfix_identifier_declarator =
        paren_identifier_declarator postfixing_abstract_declarator
        | '(' unary_identifier_declarator ')'
        | '(' unary_identifier_declarator ')'
                postfixing_abstract_declarator

paren_identifier_declarator =
        identifier
        | '(' paren_identifier_declarator ')'

abstract_declarator =
        unary_abstract_declarator
        | postfix_abstract_declarator
        | postfixing_abstract_declarator

postfixing_abstract_declarator =
        array_abstract_declarator
        | '(' ')'
        | '(' parameter_type_list ')'

array_abstract_declarator =
        '[' ']'
        | '[' constant_expression ']'
        | array_abstract_declarator '[' constant_expression ']'

unary_abstract_declarator =
        '*'
        | '*' type_qualifier_list
        | '*' abstract_declarator
        | '*' type_qualifier_list abstract_declarator

postfix_abstract_declarator =
        '(' unary_abstract_declarator ')'
        | '(' postfix_abstract_declarator ')'
        | '(' postfixing_abstract_declarator ')'
        | '(' unary_abstract_declarator ')'
                postfixing_abstract_declarator
```

Appendix B

SpecC Standard Library

In addition to the standard library inherited from the ANSI-C language, the SpecC Reference Compiler (SCRC) includes a standard library that supports the handling of SpecC data types. In addition, a set of standard channels is provided, covering popular synchronization, resource management and communication methods.

At this time, the SpecC standard library described in this chapter is not yet approved by the SpecC Technology Open Consortium (STOC) as part of the SpecC language. Therefore, it is subject to change.

B.1 SpecC standard type and simulation library

In particular for simulation and test bench specification, the SpecC language supports a simulation library whose application programming interface (API) is declared in the SpecC header file *sim.sh*. Via the API defined in *sim.sh*, the current simulation time in the simulator can be accessed. Also, *sim.sh* offers APIs for the handling and conversion of the special SpecC data types that are not supported by the standard library of the ANSI-C language.

The actual contents of *sim.sh* are implementation dependent.

The following example of *sim.sh* shows the declarations that can be expected to be part of any implementation of the SpecC simulation library.

Example of *sim.sh*:

```
1  /*************************************************************/
2  /* sim.sh: API for SpecC run-time simulation library         */
3  /*************************************************************/
4
5  #ifndef __SIM_SH
6  #define __SIM_SH
7
8
9  /*** type definitions ************************************/
10
11
12 typedef unsigned long long sim_time;    // simulation time
13
14
15 /*** exported functions **********************************/
16
17
18 extern sim_time now(            // current simulation time
19         void);
20
21 extern const char *time2str(    // convert time to string
22         sim_time        Time);
23
24 extern sim_time str2time(       // convert string to time
25         const char      *str);
26
27 extern char *ll2str(            // conv. longlong to string
28         unsigned int    base,           // 2 <= base <= 36
29         char            *endptr,        // last char in buff.
30         long long       ll);            // long long value
31
32 extern char *ull2str(           // conv. ulonglong to string
33         unsigned int    base,           // 2 <= base <= 36
34         char            *endptr,        // last char in buff.
35         unsigned long long ull);        // u. long long value
36
37 extern long long str2ll(        // conv. string to longlong
38         unsigned int    base,           // 2 <= base <= 36
39         const char      *str);          // string value
40
41 extern unsigned long long str2ull( // string to ulonglong
42         unsigned int    base,           // 2 <= base <= 36
43         const char      *str);          // string value
44
45 extern char *bit2str(           // convert bit to string
```

```
46         unsigned int    base,           // 2 <= base <= 36
47         char            *endptr,        // last char in buff.
48         ... /* bit[l:r] b */);          // bit vector value
49
50 extern char *ubit2str(          // convert ubit to string
51         unsigned int    base,           // 2 <= base <= 36
52         char            *endptr,        // last char in buff.
53         ... /* unsigned bit[l:r] b */); // bit vector value
54
55 extern void str2bit(            // convert string to bit
56         unsigned int    base,           // 2 <= base <= 36
57         const char      *str,           // string value
58         ... /* bit[l:r] *bptr */);      // pointer to result
59
60 extern void str2ubit(           // convert string to ubit
61         unsigned int    base,           // 2 <= base <= 36
62         const char      *str,           // string value
63         ... /* unsigned bit[l:r] *bptr */);
64
65 #endif /* __SIM_SH */
66
67 /*** EOF sim.sh ***************************************/
```

Notes:

i. The SpecC simulation library is available through inclusion of the SpecC standard header file *sim.sh*.

ii. The type *sim_time* defines the type of the simulation time. Since the actual representation of simulation time is implementation dependent, the type *sim_time* should be used for declaration of any variables of type time.

iii. The function *now* returns the current simulation time.

iv. The functions *time2str* and *str2time* convert from simulation time to a text string, and vice versa.

v. The functions *ll2str* and *str2ll* convert from **signed long long int** to a text string, and vice versa.

vi. The functions *ull2str* and *str2ull* convert from **unsigned long long int** to a text string, and vice versa.

vii. The functions *bit2str* and *str2bit* convert from bit vector to a text string, and vice versa.

viii. The functions *ubit2str* and *str2ubit* convert from unsigned bit vector to a text string, and vice versa.

B.2 SpecC standard channel library

The SpecC standard channel library provides well-known mechanisms for synchronization, resource management and communication.

The interfaces and channels listed in the following sections can be expected to be part of any SpecC language implementation. Furthermore, each of these interfaces and channels can be expected to be available for **import** into any design, by using the interface or channel name as argument to the **import** declaration.

B.2.1 Semaphore channel

Purpose: Protected access to shared resources

Synopsis:

```
interface i_semaphore
{
    void release(void);
    void acquire(void);
    bool attempt(void);
};

channel c_semaphore(in const unsigned long c)
    implements i_semaphore;
```

Semantics:

(a) Each thread must call *acquire()* before using a resource and call *release()* when the resource is not used any more. Calling *release()* without prior call to *acquire()* is illegal.

(b) Calling *acquire()* multiple times in order to reserve multiple resources at the same time is legal. However, a deadlock situation may occur if an insufficient number of resources is available.

(c) *release()* may only be called as many times as *acquire()* has been called.

(d) Calling *acquire()* may suspend the calling thread indefinitely.

(e) Method *attempt()* tries to aquire a resource without any waiting and returns immediately with or without success. *attempt()* returns **false** if the resource could not be acquired, and **true** if the resource has successfully been acquired.

(f) A successful *attempt()* must be followed by a call to *release()*. An unsuccessful *attempt()* must not be followed by a call to *release()*.

(g) If a thread needs to obtain multiple resources at the same time, a global partial order of obtaining the resources should be used, otherwise deadlock situations may occur.

(h) One channel instance is required for each set of shared resources.

(i) The number of available resources is given as an external count which must be specified at the time of the channel instantiation.

B.2.2 Mutex channel

Purpose: Mutually exclusive access to a shared resource; binary semaphore

Synopsis:

```
interface i_semaphore
{
    void release(void);
    void acquire(void);
    bool attempt(void);
};

channel c_mutex implements i_semaphore;
```

Semantics:

(a) Each thread must call *acquire()* before using a resource and call *release()* when the resource is not used any more. Calling *release()* without prior call to *acquire()* is illegal.

(b) Calling *acquire()* multiple times in order to reserve multiple resources at the same time is legal. However, a deadlock situation may occur if an insufficient number of resources is available.

(c) *release()* may only be called as many times as *acquire()* has been called.

(d) Calling *acquire()* may suspend the calling thread indefinitely.

(e) Method *attempt()* tries to aquire a resource without any waiting and returns immediately with or without success. *attempt()* returns **false** if the resource could not be acquired, and **true** if the resource has successfully been acquired.

(f) A successful *attempt()* must be followed by a call to *release()*. An unsuccessful *attempt()* must not be followed by a call to *release()*.

(g) If a thread needs to obtain multiple resources at the same time, a global partial order of obtaining the resources should be used, otherwise deadlock situations may occur.

(h) One channel instance is required for each mutex.

B.2.3 Critical section channel

Purpose: Protected access to a critical section

Synopsis:

```
interface i_critical_section
{
    void enter(void);
    void leave(void);
};

channel c_critical_section implements i_critical_section;
```

Semantics:

(a) Each thread must call *enter()* before entering the critical section and call *leave()* after leaving the critical section.

(b) Calling *leave()* without prior call to *enter()* is illegal.

(c) Calling *enter()* twice without *leave()* in between is illegal.

(d) Calling *leave()* twice without *enter()* in between is illegal.

(e) Calling *enter()* may suspend the calling thread indefinitely.

(f) If a thread needs to enter multiple critical sections at the same time, a global partial order of entering the sections should be used, otherwise deadlock situations may occur.

(g) One channel instance is required for each critical section.

B.2.4 Barrier channel

Purpose: Barrier synchronization, rendezvous

Synopsis:

```
interface i_barrier
{
    void barrier(void);
};

channel c_barrier(in unsigned long N)
    implements i_barrier;
```

Semantics:

(a) Each participating thread calls *barrier()* to synchronize with the other participating threads at the barrier.

(b) A call to *barrier()* will suspend the calling thread until all other participating threads have arrived at the barrier. Then, all participating treads resume their execution.

(c) Calling *barrier()* may suspend the calling thread indefinitely.

(d) One channel instance is required for each barrier.

(e) At the time of barrier instantiation, the number *N* of participating threads that use the barrier is fixed.

(f) Exactly *N* threads must use the barrier. If less than *N* or more than *N* threads use the barrier, the behavior is undefined.

B.2.5 Token channel

Purpose: Modeling Petri nets with consumers and producers

Synopsis:

```
interface i_consumer
{
    void consume(unsigned long n);
};

interface i_producer
{
    void produce(unsigned long n);
};

interface i_token
{
    void produce(unsigned long n);
    void consume(unsigned long n);
};

channel c_token implements i_producer, i_consumer, i_token;
```

Semantics:

(a) *i_consumer* represents a consumer interface to a token channel as known from Petri nets. Each connected thread acts as a consumer.

(b) *i_producer* represents a producer interface to a token channel as known from Petri nets. Each connected thread acts as a producer.

(c) *i_token* represents a general interface to a token channel as known from Petri nets. Each connected thread may act as both, a consumer and/or producer.

(d) A consumer calls *consume(n)* to consume *n* tokens.

(e) A call to *consume()* will return immediately if the requested number of tokens is already present, consuming those tokens.

(f) A call to *consume()* will block the caller if not enough tokens are present, until a sufficient number of tokens has been produced.

(g) Calling *consume()* may suspend the calling thread indefinitely.

(h) A producer calls *produce(n)* to produce *n* tokens.

(i) A call to *produce()* will produce the given number of tokens and immediately return.

(j) One token channel instance may be used multiple times and with multiple consumers and/or producers.

(k) If used for production and consumption by the same thread, the thread may consume its own tokens.

B.2.6 Queue channel

Purpose: Type-less, fixed-size queue for use with any number of senders and receivers

Synopsis:

```
interface i_sender
{
    void send(void *d, unsigned long l);
};

interface i_receiver
{
    void receive(void *d, unsigned long l);
};

interface i_tranceiver
{
    void send(void *d, unsigned long l);
    void receive(void *d, unsigned long l);
};

channel c_queue(in const unsigned long size)
    implements i_sender, i_receiver, i_tranceiver;
```

Semantics:

(a) A thread connected to the interface *i_sender* acts as a sender.

(b) A thread connected to the interface *i_receiver* acts as a receiver.

(c) A thread connected to the interface *i_tranceiver* acts as a tranceiver. In other words, it may act as a sender, receiver, or both.

(d) A call to *send()* sends out a packet of data to a connected channel.

(e) Calling *send()* may suspend the calling thread indefinitely.

(f) A call to *receive()* receives a packet of data from a connected channel.

(g) Calling *receive()* may suspend the calling thread indefinitely.

(h) Data packets are typeless (represented by an array of bytes) and may vary in size for separate calls to *send()* and *receive()*.

(i) The queue channel *i_queue* operates in first-in-first-out (FIFO) mode.

(j) One channel instance is required for each queue.

(k) Multiple threads may use the same channel instance.

(l) The size (number of bytes) of the queue must be specified at the time of the channel instantiation. The data packet size must not be larger than the specified size of the queue.

(m) If different packet sizes are used with the same queue, a receiver may receive only partial or multiple packets depending on the requested packet size.

(n) If insufficient space is available in the queue, *send()* will suspend the calling thread until sufficient space becomes available.

(o) If insufficient data is available in the queue, *receive()* will suspend the calling thread until sufficient data becomes available.

B.2.7 Handshake channel

Purpose: Safe one-way synchronization between a sender and a receiver

Synopsis:

```
interface i_receive
{
    void receive(void);
};

interface i_send
{
    void send(void);
};

channel c_handshake implements i_send, i_receive;
```

Semantics:

(a) A thread connected to the interface *i_send* acts as a sender.

(b) A thread connected to the interface *i_receive* acts as a receiver.

(c) A call to *receive()* lets the receiver wait for a handshake from the sender.

(d) If a handshake is present at the time of *receive()*, the call to *receive()* will immediately return.

(e) If no handshake is present at the time of *receive()*, the calling thread is suspended until the sender sends the handshake. Then, the receiver will resume its execution.

(f) Calling *receive()* may suspend the calling thread indefinitely.

(g) A call to *send()* sends a handshake to the receiver. If the receiver is waiting at the time of the *send()*, it will wake up and resume its execution. Otherwise, the handshake is stored until the receiver calls *receive()*.

(h) Calling *send()* will not suspend the calling thread.

(i) The behavior is undefined if *send()* is called successively without any calls to *receive()*.

(j) Only one sender and one receiver may use the channel at any time. Otherwise, the behavior is undefined.

B.2.8 Double handshake channel

Purpose: 2-way handshake channel for type-less data transfer from a sender to a receiver

Synopsis:

```
interface i_sender
{
    void send(void *d, unsigned long l);
};

interface i_receiver
{
    void receive(void *d, unsigned long l);
};

channel c_double_handshake implements i_sender, i_receiver;
```

Semantics:

(a) A thread connected to the interface *i_sender* acts as a sender.

(b) A thread connected to the interface *i_receiver* acts as a receiver.

(c) A call to *send()* sends out a packet of data to a connected channel.

(d) Calling *send()* may suspend the calling thread indefinitely.

(e) A call to *receive()* receives a packet of data from a connected channel.

(f) Calling *receive()* may suspend the calling thread indefinitely.

(g) Data packets are typeless (represented by an array of bytes) and may vary in size for separate calls to *send()* and *receive()*.

(h) The channel *c_double_handshake* operates in rendezvous fashion. A call to *send()* will suspend the sender until the receiver calls *receive()*, and vice versa. When both communicating parties are ready, data is transferred from the sender to the receiver and both can resume their execution.

(i) Exactly one receiver and one sender thread may use the same channel instance. If used by more than one sender or receiver, the behavior of the channel is undefined.

(j) The same channel instance may be used multiple times in order to transfer multiple data packets from the sender to the receiver.

(k) If different packet sizes are used with the same channel, the user has to ensure that the data size of the sender always matches the data size expected by the receiver. It is an error if the sizes in a transaction don’t match.

Bibliography

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

[2] D. Gajski, F. Vahid, S. Narayan, J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[3] F. Vahid, S. Narayan, D. Gajski. *SpecCharts: A VHDL Front-End for Embedded Systems*. IEEE Transactions on CAD, Vol. 14, No. 6, pp. 694-706, 1995.

[4] 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, 1997.

[5] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 1999.

[6] Accellera C/C++ Working Group of the Architectural Language Committee. *RTL Semantics, Draft Specification.* Accellera, February, 2001.
`http://www.eda.org/alc-cwg/cwg-open.pdf`.

[7] R. Dömer, A. Gerstlauer, D. Gajski. *SpecC Language Reference Manual, Version 1.0.* SpecC Technology Open Consortium, Japan, March 2001.

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

[9] M. Fujita, H. Nakamura. *The Standard SpecC Language*. Proceedings of the International Symposium on System Synthesis, Montreal, October 2001.

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

Index