SpecC Language Reference Manual

Version 1.0

Authors:

Rainer Dömer
Andreas Gerstlauer
Daniel Gajski

March 6, 2001

Contents

Abstract

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

The SpecC language is defined as extension of the ANSI-C programming language. This document describes 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. In the Appendix, the full SpecC grammar is included by use of an Extended Backus-Naur-Form (EBNF) notation.

Chapter 1

Introduction

The SpecC language is a formal notation intended for the specification and design of digital embedded systems including hardware and software. 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. 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 describes the foundation, the types, the classes, the statements, and other constructs of the SpecC language. In addition, the complete grammar of the SpecC language is included in Appendix A.

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) [2]. 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” [3].

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 Open Technology Consortium (STOC) in 1999 [4]. 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.

This document defines version 1.0 of the SpecC language standard approved by STOC.

1.2 Contributors

The contributors to this SpecC Language Reference Manual are, in alphabetical order:

Rainer Dömer

Daniel Gajski

Andreas Gerstlauer

Shuqing Zhao

Jianwen Zhu

Chapter 2

SpecC Language

2.1 ANSI-C Basis

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 intended 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.

Example:

```
1 int       a[10],
2           b[10];
3 double    c[3][3],
4           d[3][3];
5
6 void f(void)
7 {
8     a = b;          // array assignment
9     c = d;          // array assignment
10 }
```

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).

Example:

```
1 int       a = 0,   // explicitly initialized to 0
2           b;       // implicitly initialized to 0
3 char      c;       // implicitly initialized to '\000'
4 float     d;       // implicitly initialized to 0.0f
5 void      *e;      // implicitly initialized to 0 (NULL)
6 long      f[2];    // implicitly initialized to {0l,0l}
7
8 void f(void)
9 {
10   int             x;       // uninitialized
11   static int      y;       // initialized to 0
12
13   ...
14 }
```

2.2 SpecC Types

2.2.1 Boolean Type

Purpose: Explicit support of the Boolean data type

Synopsis:

```
basic_type_name =
        ...
        | bool

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

Semantics:

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

(b) It can be used to express the result of logical operations (e. g. $<$, $>$, $==$, etc.).

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

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. A boolean type cannot be **signed** or **unsigned**.

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

2.2.2 Long Long Type

Purpose: Explicit support of a high precision integer data type

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
        ...

basic_type_specifier =
        basic_type_name
        | basic_type_specifier basic_type_name
        ...

constant =
        ...
        | integer
```

Semantics:

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

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

(c) The **long long** type is an integer type with high precision. Its precision is equal to or higher than the precision of the **long int** type.

(d) The usual promotion and 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, 9223372036854775807]
21 unsigned long long ULongLong;    //  64 bit, unsigned
22                                   // [0, 18446744073709551615]
23 }
```

Notes:

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

2.2.3 Bitvector Type

Purpose: Explicit support for bitvectors of arbitrary length

Synopsis:

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

basic_type_name =
        ...
        | 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 bitvector **bit**$[l:r]$ represents an integral data type of arbitrary precision (length). The length of a bitvector is determined by the difference of its left and right bounds: $length(bv) = abs(left(bv) - right(bv) + 1)$.

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

(c) The left and right bounds, *l* and *r*, of a bitvector are specified at the time of declaration and must be constant expressions evaluatable at compile time. The same applies to *length*, for the short declaration.

(d) A bitvector is either **signed** (default) or **unsigned**.

(e) A bitvector can be used as any other integral type in expressions (for example, type **int** is equivalent to type **bit**$[sizeof(\textbf{int}) * 8 - 1 : 0]$).

(f) Implicit promotion from (**unsigned**) **int**, (**unsigned**) **long**, or (**unsigned**) **long long** to bitvector is performed when necessary.

(g) Automatic conversion (signed/unsigned extension or truncation) is supported as with any other integral type.

(h) Bitvector constants are noted as a sequence of zeros and ones directly followed by a suffix **u** or **ub** indicating the bitvector type (see the synopsis and example).

(i) In addition to all standard C operations, a concatenation operation, noted as @, and a slicing operation, noted as $[lb : rb]$, are supported (see lines 9 and 11 in the example). Both operations can be applied to bitvectors as well as to any other integral type (which will be treated as bitvector of suitable length).

(j) A bit-access operation, noted as $[b]$ (same as the array access operator), is provided as a short-hand for accessing a single bit ($[b : b]$) of a bitvector. The result type of this operation is **unsigned bit**$[0 : 0]$.

(k) The slicing operation requires the left and right bounds to be constant expressions which can be evaluated at compile time. This restriction does not apply to the single bit access.

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 bitvector can be thought of as a parameterized type whose bounds are defined with the name of the type.

ii. The length of any bitvector expression is always known at compile time. This is important for synthesis.

iii. Typically, no explicit type casting is necessary for operations on bitvectors.

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

2.2.4 Long Double Type

Purpose: Explicit support of a high precision floating point data type

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** type is a floating point type with high precision. Its precision is equal to or higher than the precision of the **double** 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
4 }
```

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: Events serve as a mechanism for synchronization and exception handling

Synopsis:

```
basic_type_name =
        ...
        | event

wait_statement =
        wait paren_event_list ';'

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

Semantics:

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

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

(c) Events can only be used with the **wait** and **notify** statements (see the example and Section 2.4.5), or with the **try-trap-interrupt** construct described in Section 2.4.6.

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:

2.2.6 Time Type

Purpose: Specification of simulation time

Synopsis:

```
waitfor_statement =
        waitfor time ';'

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

time_opt =
        <nothing>
        | time

time =
        constant_expression
```

Semantics:

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

(b) The time type is used only with the **waitfor** statement and with **range** statements in the **do-timing** construct (see Section 2.4.7).

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:

2.3 SpecC Classes

2.3.1 Behavior Class

Purpose: Active object for specification of behavior; 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) In SpecC, the functionality of a system is described by a hierarchical network of behaviors. A **behavior** declaration is a class declaration that consists of an optional set of ports and an optional set of implemented interfaces.

(b) A behavior definition contains a body that consists of an optional set of instantiations, an optional set of local variables and methods, and a mandatory `main` method.

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

(d) A behavior can implement a list of interfaces, as described with the channel construct in Section 2.3.2.

(e) A behavior that implements an interface, can refer back to itself by use of the **this** keyword. **this** can only be used inside a behavior body and 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 function of method parameter must be the interface type implemented by the behavior.

(f) All methods declared in a behavior are private, except the `main` method and methods implemented for interfaces.

(g) A behavior is called a composite behavior if it contains instantiations of other behaviors (as described in Section 2.3.5). Otherwise, it is called a leaf behavior.

(h) The `main` method of a behavior is called whenever an instantiated behavior is executed. Also, the completion of the `main` method determines the completion of the execution of the behavior.

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

(j) A behavior definition requires that all interfaces implemented by the behavior are defined (have a body).

(k) 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 to output port
15       }
16 };
```

Notes:

i. The example shows a simple leaf behavior. For typical composite behaviors, please refer to Sections 2.4.1 to 2.4.6.

ii. Local variables and methods, as *a*, *b*, and *f* in the example, can be used to conveniently program the functionality of a behavior.

iii. 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.

iv. Declarations of behaviors are sufficient to determine compatibility of the behaviors. Full definitions are not needed. This is important for reuse of IP and ”plug-and-play”.

v. The behavior `Main` usually is a composite behavior containing the testbench for the design as well as the instantiation of the actual design specification.

vi. Behaviors that implement interfaces are rarely used. Implemented interfaces for behaviors are only useful for communication schemes that involve call-backs. For call-back communication, the channel implementing the communication protocol, can call back methods provided by the behavior that called 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 function as an argument of interface type. The argument is set by the behavior implementing the call-back by use of the **this** keyword.

vii. Note that the type of **this** in SpecC is a behavior, not a pointer.

2.3.2 Channel Class

Purpose: Passive object for specification of protocols; 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) Communication between behaviors should be encapsulated in channels. A **channel** declaration is a class declaration that consists of an optional set of ports and an optional set of implemented interfaces.

(b) A channel definition contains the channels body which consists of an optional list of instantiations and an optional set of local variables and methods.

(c) A channel can include a list of ports through which it can communicate with other channels or behaviors. Ports are described in detail in Section 2.3.4.

(d) A channel is called a hierarchical channel if it contains instantiations of other channels (as described in Section 2.3.5).

(e) A channel is called a wrapper if it instantiates behaviors.

(f) Variables and methods defined in a channel cannot be accessed from the outside (in contrast to members of structures), unless through implemented interfaces.

(g) By use of implemented interfaces (see Section 2.3.3), a subset of the internal methods can be made accessable.

(h) The **implements** keyword declares the list of interfaces that are implemented by the channel.

(i) All the methods of the implemented interfaces must be defined inside the channel. A channel definition (a channel with body) requires that all implemented interfaces are definitions (have a body).

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

(k) Methods encapsulated in channels are executed in non-preemptive (atomic) manner. A thread executing a channel method is guaranteed not to be interrupted by other threads, unless it is waiting for an event (at a **wait** statement) or waiting for simulation time increase (at a **waitfor** statement). Also, atomic execution does not apply to functions or methods called from a channel method, unless a method in a channel (that is atomic itself) is called.

Example:

```
 1 interface I;
 2
 3 channel C (void) implements I
 4 {
 5    int  data;
 6
 7    void send(int x)
 8       {
 9          data = x;
10       }
11
12    int receive(void)
13       {
14          return(data);
```

```
15        }
16 };
```

Notes:

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

2.3.3 Interface Class

Purpose: Link between behaviors and channels; support for reuse of IP and ”plug-and-play”

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 consists of a set of method declarations. The method definitions for these declarations are supplied by a channel or behavior that **implements** the interface.

(b) A behavior or channel can have ports of interface type. Via such an interface port, the behavior or channel can call the communication methods declared in that interface. The actual channel or behavior performing these methods is determined by the mapping of the interface port.

(c) For each interface, multiple channels can provide an implementation of the declared communication methods. However, each channel must provide all methods declared in the interface.

(d) Any channel implementing a specific interface can be connected to a behavior or channel port of the interface type when the behavior of channel is instantiated (see Section 2.3.5).

Example:

```
 1 interface I
 2 {
 3    void send(int x);
 4    int receive(void);
 5 };
 6
 7 interface I4B
 8 {
 9    int get_word(void);
10    void put_word(int d);
11 };
12
13 interface I4C
14 {
15    void send_block(I4B b);
16    void receive_block(I4B b);
17 };
```

Notes:

i. Interfaces can be used to connect behaviors with channels in a way so that both, the behaviors and the channels, are easily interchangable with compatible components (”plug-and-play”).

ii. The example shows the **interface** *I* which declares the *send* and *receive* methods for the channel *C* in the example from Section 2.3.2.

iii. The interfaces *I4C* and *I4B* in the example can be used to define a communication scheme involving call-backs. Assuming, there is a behavior *B* implementing interface *I4B* and a channel *C* implementing interface *I4C*, the communication protocol is initiated by the behavior *B* with a call to *send_block* or *receive_block*, where the

behavior *B* supplies itself as an argument by use of the keyword **this**. These methods, implemented in the channel *C*, can then call-back the methods *get_word* and *put_word* in order to get or store the data word by word.

2.3.4 Ports

Purpose: Specification of connectors of behaviors and channels

Synopsis:

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

port_list =
        port_declaration
        | port_list ',' port_declaration

port_declaration =
        port_direction parameter_declaration
        | interface_parameter

port_direction =
        <nothing>
        | in
        | out
        | inout

interface_parameter =
        interface_name
        | interface_name identifier
```

Semantics:

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

(b) A port can be one of two types: standard or interface type. A standard type port can be of any SpecC type, but includes the ports direction as an additional type modifier.

(c) A port direction can be **in**, **out**, or **inout**, and is handled as an access restriction to that port.

(d) An **in** port allows only read access from inside the class (write access from the outside).

(e) An **out** port only allows write access from the inside (read access from the outside).

(f) An **inout** port can be accessed bi-directionally.

(g) For ports of event type, read access is to **wait** on the event, write access is to **notify** the event.

(h) As a shortcut, a port without any direction modifier is considered an **inout** port.

(i) On the other hand, an interface type port enables access to the methods of that interface class. Via such a port, a behavior or a channel can call any of the methods declared in the interface.

Example:

```
1 interface I;
2
3 behavior B1 (in int p1, out int p2, in event clk);
4
5 behavior B2 (I i, inout event clk);
6
7 channel C (inout bool f) implements I;
```

Notes:

2.3.5 Class Instantiation

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

Synopsis:

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

instance_declarator =
        identifier 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
        | identifier
        | identifier '[' constant_expression ':' constant_expression ']'
        | identifier '[' constant_expression ']'
```

Semantics:

(a) SpecC supports structural hierarchy by allowing child behaviors and child channels to be instantiated as components inside compound behaviors and channels. The instantiation of behaviors and channels also defines the connectivity of the instantiated components.

(b) A class instantiation defines connections to ports by use of a port mapping list. The port mapping list is left out if the class has no ports.

(c) In the port mapping list, each port of the instantiated class is mapped onto a corresponding constant, variable or port of compatible type, or is left open.

(d) A port mapping to a constant is only allowed for **in** ports.

(e) An open port mapping is only allowed for **out** ports.

(f) Ports of interface type must be mapped onto a channel or behavior that implements the interface, or another port of the same type.

(g) The port type must match the type of the mapped constant, variable or port, in the same way as the types of arguments to a function call must match the types of the function parameters.

(h) As a special case, a port of bitvector type can be connected to a list of concatenated bitslices. In this case the connection restrictions listed above apply accordingly for each single bit of the bitvector.

(i) 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, b, sum;
14
15    C       c(b);          // instantiate c as channel C
16    B1      b1(i,bus1,e), // instantiate b1 and b3 as behavior B1
17            b3(i,bus2[15:8],e);
18    B2      b2(c, e);      // instantiate b2 as behavior B2
19    Adder8  a8(a, b, 0b,   // constant mapping
20               sum,
21               /*open*/); // open mapping
22 };
```

Notes:

i. For bitvector ports, a port of type **bit**$[15:0]$ can be mapped onto two adjacent busses $a[7:0]@b[0:7]$, for example.

ii. The example above contains five class instantiations. In line 15, a channel *c* is instantiated as type channel *C*. Its only port of type **bool** is mapped to the Boolean variable *b*.

iii. Lines 16 and 17 instantiate two behaviors *b*1 and *b*3 (of type behavior *B*1) which are both connected to integer *i* and event *e*. The second port of *b*1 is connected to *bus*1 (the first port of *B*), whereas the second port of *b*3 is mapped to the higher bits of *bus*2.

iv. In line 18, *b*2 is instantiated as a *B*2 type behavior. Its ports are mapped to the channel *c* (instantiated in line 15) and event *e*.

v. In line 19, an adder *a*8 is instantiated, adding *a* and *b* to *sum*. Its carry input is connected to zero (hardwired to GND), its carry output is left open (unused).

2.4 SpecC Statements

2.4.1 Sequential Execution

Purpose: Specification of sequential control flow

Synopsis:

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

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

Semantics:

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

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 above shows the trivial case of sequential, unconditional execution of three behaviors, *b*1, *b*2 and *b*3.

2.4.2 Parallel Execution

Purpose: Specification of concurrency

Synopsis:

```
concurrent_statement =
        ...
        | par compound_statement

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

Semantics:

(a) Concurrent execution of behaviors can be specified with the **par** statement.

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

(c) When executed in the simulator, the order of execution for the parallel threads is undefined (i.e. any order is possible, including interleaved execution).

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

(e) The statements in the compound statement block are limited to function calls to `main` methods of behaviors. No other statement 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. Note that in simulation, concurrent threads of control are probably not really executed in parallel. Instead, the scheduler, which is part of the simulation run-time system, always executes one thread at a time and decides when to suspend and when to resume a thread depending on simulation time advance and synchronization points.

ii. The threads in a **par** statement may be executed by a thread implementation with preemption. In other words, the order in which the simulator executes the threads, or portions of the threads, is undefined and no assumption on the order should be made. In case a specific order is desired, this must be specified by use of explicit synchronization or communication.

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

2.4.3 Pipelined Execution

Purpose: Explicit support for specification 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 '}'
```

Semantics:

(a) Pipelined execution, specified by the **pipe** statement, is a special form of concurrent execution. As the threads in a **par** statement, all statements in the compound statement block after the **pipe** keyword form a new thread of control.

(b) The threads in a **pipe** statement represent pipeline stages and are executed in pipelined fashion. Each stage runs in parallel to the others, but works on different sets of data.

(c) In its first form, without the arguments, the **pipe** statement never finishes (except through abortion which is described in Section 2.4.6). In other words, the unspecified condition is assumed to be constant **false**, thus, the pipeline never reaches the flushing state.

(d) In its second form, the optional expressions specify the number of iterations the pipeline is executed. The expressions are used in the same fashion as with the **for**

statement. The first expression is is evaluated before the pipeline execution as an initializer. The second expression is the termination condition that is evaluated at the beginning of each iteration. If it evaluates to **true**, the pipeline execution is continued, otherwise the pipeline is flushed. The third expression is evaluated once in each iteration, typically used in order to increment the iteration counter.

(e) Each state in the pipeline is executed the same number of times.

(f) In a pipeline with n stages, the n-th behavior is executed for the first time after $n-1$ iterations of the first stage.

(g) When the termination condition becomes **false**, the n-th stage of the pipeline is executed for $n-1$ more iterations.

(h) In order to support buffered communication in pipelines, the **piped** storage class can be used for variables connecting pipeline stages. A variable with **piped** storage class can be thought of as a variable with two storages. Write access always writes to the first storage, read access reads from the second storage.

(i) For a **piped** variable, the contents of the first storage are shifted to the second storage whenever a new iteration starts in the pipeline.

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

(k) The statements in the compound statement block are limited to function calls to `main` methods of behaviors. No other statement is allowed.

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. In the example, the behaviors $b1$, $b2$ and $b3$ form a pipeline of behaviors. In the first iteration, only $b1$ is executed. When $b1$ finishes the second iteration starts and $b1$ and $b2$ are executed in parallel (as with the **par** statement). In the third iteration, after $b1$ and $b2$ have completed, $b3$ is executed in parallel with $b1$ and $b2$. Every following iteration is the same as the third iteration, until the termination condition $i < 10$ becomes false. Then, $b2$ and $b3$ are executed in parallel one more time, and finally only $b3$ is executed once.

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

2.4.4 Finite State Machine Execution

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

Synopsis:

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

transition_list =
        transition
        | transition_list transition

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

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 explicit specification of state transitions. Both Mealy and Moore type finite state machines can be modeled with the **fsm** construct.

(b) As shown in the synopsis section above, the **fsm** construct specifies a list of state transitions where the states are instantiated behaviors.

(c) A state transition is a triple $\langle current\_state, condition, next\_state \rangle$. The *current_state* and the *next_state* take the form of labels and denote behavior instances. The

condition is an expression which has to be evaluated to **true** for the transition to become valid.

(d) Each behavior instance must be listed exctly once in the list as a current behavior (every label may only appear once).

(e) The transition condition is optional. If it is left out, it defaults to **true**.

(f) As a special form of the next state, a **break** statement terminates the execution of the **fsm** construct.

(g) The execution of a **fsm** construct starts with the execution of the first behavior that is listed in the transition list. Once the behavior has finished, its state transitions determine the next behavior to be executed.

(h) The conditions of the transitions are evaluated in the order they are specified, and as soon as one condition becomes **true** the specified next behavior is started.

(i) If none of the conditions is true, the next behavior defaults to the following behavior instance listed (similar to a **case** statement without **break**). If there is no following instance, the **fsm** construct terminates (as if it had reached a **break** 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 the body of the **fsm** construct does not allow arbitrary statements. As specified in the synopsis section, the grammar limits the state transitions to well-defined triples.

2.4.5 Synchronization

Purpose: Support for synchronization of concurrent threads

Synopsis:

```
wait_statement =
        wait paren_event_list ';'

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

paren_event_list =
        event_list
        | '(' event_list ')'

event_list =
        identifier
        | event_list ',' identifier
```

Semantics:

(a) There are three statements to support synchronization between concurrent threads of execution: **wait**, **notify** and **notifyone**. Each of these statements takes a list of events (described in Section 2.2.5) as argument.

(b) Each thread is either active, or waiting.

(c) The **wait** statement suspends the current thread from execution (puts it into the waiting state), until one of the specified events is notified. Then, the thread becomes active again and resumes its execution.

(d) The **notify** statement triggers all specified events so that all threads, which are currently waiting on any of these events, can continue their execution.

(e) The same way as the **notify** statement, the **notifyone** statement triggers all specified events. However, only one thread out of the set of threads, that are currently waiting on the events triggered by this **notifyone**, is resumed.

(f) A notified event is guaranteed to wake up all threads (for **notifyone**, one of the threads) that are currently waiting for the event, including active threads that will be waiting for the event as their immediate next state.

(g) Otherwise, if there is no such thread waiting, the notified event is ignored.

Example:

```
 1 behavior A(out int x, out event e)
 2 {
 3    void main(void)
 4    {
 5       x = 42;
 6       notify e;
 7    }
 8 };
 9
10 behavior B(in int x, in event e)
11 {
12    void main(void)
13    {
14       wait(e);
15       printf("%d", x);
16    }
17 };
18
19 behavior Main
20 {
21    int   x;
22    event e;
23    A a(x, e);
24    B b(x, e);
25
26    void main(void)
27    { par { a.main();
28            b.main();
29          }
30    }
31 };
```

Notes:

i. 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.

ii. Notified events can be thought of as being collected until no active behavior is available for execution any more. Then, the collected events are applied to the waiting threads, activating those threads that are waiting on any of the collected events.

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

iv. 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 event semantics described above ensure that the event notified by A is not lost.

v. Note also that, in an extreme example, the event semantics allow for a thread to wake up itself, i.e. a thread will pass through a statement sequence of **notify** e; **wait** e;.

2.4.6 Exception Handling

Purpose: Support for abortion of execution and interrupt 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 =
        identifier
        | event_list ',' identifier
```

Semantics:

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

(b) With **try**, the listed behavior, including all its children, is made sensitive to the events listed with the **trap** and **interrupt** declarations. Sensitive means that, when any execution thread of the **try** behavior is waiting for events (at a **wait** statement) or waiting for simulation time increase (at a **waitfor** statement), it will implicitly wait also for the listed exception events, which take priority over the actual **wait** or **waitfor**. Thus, whenever an exception event occurs while executing the **try** behavior or any of its children, the execution is immediately suspended.

(c) For an **interrupt** event, the specified interrupt handler is executed. Immediately after its completion, the execution of the **try** behavior is resumed from the point it was stopped.

(d) For a **trap** event, the suspended execution is aborted and the trap handler takes over the execution.

(e) The **try-trap-interrupt** construct completes with the completion of the **try** behavior, or the completion of a **trap** behavior.

(f) During the execution of the exception handlers, any occurring exception events are ignored. In other words, the exception handlers are not sensitive to the events listed as exceptions, unless explicitly specified by another local **try** construct inside the handlers.

(g) In one **try-trap-interrupt** construct, the **interrupt** and **trap** declarations are prioritized in the order they are listed. Only the first specified exception, that matches any of the notified events, is executed.

(h) For hierarchically composed **try-trap-interrupt** constructs, the outer, higher-level exceptions take precedence over the inner, lower-level exceptions. Any exception event is serviced only at its highest possible level.

(i) The statements in the compound statement blocks are limited to zero or one function calls to `main` methods of behaviors. No other statement 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 the example, whenever event *e*1 is notified during execution of behavior *b*1, the execution of *b*1 is immediately suspended and behavior *b*2 is started. Then, when *b*2 finishes, the execution of behavior *b*1 is resumed right from the point where it was interrupted.

ii. During execution of *b*2, any events *e*1 are ignored. In other words, the interrupt does not interrupt itself.

iii. As soon as event *e*2 occurs, while executing behavior *b*1, the execution of *b*1 is aborted and *b*3 is started. Then, the termination of *b*3 will also terminate the execution of the **try-trap-interrupt** construct.

iv. A system reset can be modeled with a **try-trap** construct enclosed in an infinite reset loop.

v. The prioritiy rules described above essentially say that any simultaneously occurring events can only cause one single exception to be serviced.

vi. Also, exception events are not stored or queued.

2.4.7 Timing Specification

Purpose: Explicit specification of execution time and timing constraints

Synopsis:

```
waitfor_statement =
        waitfor time ';'

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
```

Semantics:

(a) There are two constructs that support the specification of timing (simulation time) in SpecC: **waitfor** and **do-timing**.

(b) The **waitfor** statement specifies execution time (or delay). Whenever the simulator reaches a **waitfor** statement, the execution of the current behavior is suspended for the specified amount of simulation time units.

(c) The argument of the **waitfor** statement must be of time type, or must be implicitly convertable to time type (see also Section 2.2.6).

(d) As soon as the simulation time has been increased by the number of time units specified with the **waitfor** statement, the execution of the current behavior will resume, unless it is still handling an interrupt exception. In this case, the execution will resume immediately after the interrupt has been handled.

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

(f) The **do**-**timing** construct is used to specify timing constraints in terms of minimum and maximum number of time units.

(g) The action block introduced with the **do** keyword is a compound statement block that includes labeled statements. Only the labels defined in this block can be used in the following **timing** block.

(h) The execution of the action block is the same as the execution of any other compound statement block. The enclosed statements are executed in the specified order.

(i) Timing constraints are specified with **range** statements in the **timing** block. Each constraint consists of two labels, which must have been defined in the action block, and a minimum and maximum time value.

(j) The minimum and maximum times are optional constant expressions of time type. These must be evaluatable at compile time.

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

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

(m) The **do**-**timing** construct specifies synthesis constraints. As such, it does not change the execution of the compound statement block, unless constraint validation is enabled in the simulator. The way, the simulator performs the constraint validation, is implementation dependent.

Example:

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

Notes:

i. Typically, constraint validation is performed as follows: during execution of the action block, the runtime system of the simulator collects time stamps at the execution of each label. After the execution of the action block is completed, these time stamps

are then used to validate the **range** constraints. Any violation of the specified constraints is reported to the user in form of a warning or error message.

ii. If a simulator allows constraint validation, this process should be parameterizable by the user. At least, there should be a mechanism to disable such checking.

iii. 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 action block, one valid instance of implementation (out of many possible implementations) is shown.

The **waitfor** statements in the action block can be validated by the range check performed during simulation. Without the **waitfor** statements, the specified timing constraints would not hold and the simulation would report errors or warnings, if the validation is turned on.

2.5 Other SpecC Constructs

2.5.1 Library Support

Purpose: Support of component libraries

Synopsis:

```
import_definition =
        import string_literal_list ';'

string_literal_list =
        string
        | string_literal_list string
```

Semantics:

(a) The SpecC language supports the inclusion of (possibly) precompiled design libraries into the current description by use of the **import** construct.

(b) The string argument of the **import** declaration denotes the file name of the library to be integrated.

(c) Multiple imports of the same library are automatically suppressed. Only one **import** declaration is effective for a library, repeated **import** declarations with the same library are ignored.

(d) The search for the library file in the file system is implementation dependent. (Usually, this involves applying a suffix and searching in a list of specified directories.)

(e) The contents of an imported library file are SpecC declarations and definitions. These are incorporated into the current description as if they were specified directly in the source code. The same rules apply for definitions in an imported file as for definitions in the source code. Multiple definitions of the same symbol are not allowed.

(f) The format of the imported file is implementation dependent. For example, source code and/or precompiled binary files can be supported for imported libraries.

Example:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 import "Interfaces/I1";
5 import "Interfaces/I2";
6 import "Channels/PCI_Bus";
7 import "Components/MPEG_II";
```

Notes:

i. In contrast to the `#include` construct inherited from the C language, the **import** construct 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** construct is seen by the SpecC compiler or synthesizer, i.e. it is not eliminated by the C preprocessor as the `#include` construct is. Thus, **import** can be used by the tools for code structuring purposes.

2.5.2 Persistent Annotation

Purpose: Support for persistent design annotation for synthesis purposes

Synopsis:

```
any_declaration =
        ...
        | note_definition

any_definition =
        ...
        | note_definition

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

annotation =
        constant_expression

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

Semantics:

(a) Using the **note** definition, a persistent annotation can be attached to any symbol, label, and user-defined type in the SpecC description.

(b) An annotation consists of a key and a value. The key is the name of the annotation. The value is any type of constant or constant expression (evaluated at compile time).

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

(d) Local annotation have their own local name space. There is no name conflict between annotations at different objects with the same key.

(e) In the first form, a note is attached to the current scope. Legal 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, the note is attached to the named object. The object name precedes the annotation key, separated by a dot.

Example:

```
 1 /* C style comment, not persistent */
 2 // C++ style comment, not persistent
 3
 4 note Author      = "Rainer Doemer";
 5 note Date        = "Tue Jan 23 08:20:37 PST 2001";
 6
 7 const int x = 42;
 8 struct S { int a, b; float f; };
 9
10 note x.Size      = sizeof(x);
11 note S.Bits      = sizeof(struct S) * 8;
12
13 behavior B(in int a, out int b)
14 {
15   note Version  = 1.1;
16
17   void main(void)
18   {
19     l1: b = 2 * a;
20         waitfor(10);
21     l2: b = 3 * a;
22
23     note NumOps  = 3;
24     note l1.OpID = 1;
25     note l2.OpID = 3;
26   }
27 };
28 note B.AreaCost = 12345;
```

Notes:

i. SpecC, as does any other programming language, allows comments in the source code to annotate the description. In particular, SpecC supports the same comment

styles as C++, which are comments enclosed in `/*` and `*/` delimiters as well as comments after `//` up to the end of the line (see lines 1 and 2 in the example above).

ii. These comments are not persistent, which means, they will be eliminated in the pre-processing step by the C preprocessor.

iii. A note can be attached to the current scope. This way, global notes (lines 4 and 5 in the example), notes at classes (line 15), notes at methods (line 23), and notes at user-defined types can be defined.

iv. Second, the object, a note will be attached to, can be named explicitly. In the example, this style is used to define the notes at variable *x* (line 10), structure *S* (line 11), and labels *l*1 and *l*2 (lines 24 and 25).

Appendix A

SpecC Grammar

In the following, the complete grammar of the SpecC language is listed in form 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 strings are concatenated, i.e. are treated the same way as one single string, if two strings directly follow each other, only separated by whitespace.

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**, **channel**, **event**, **false**, **fsm**, **implements**, **import**, **in**, **inout**, **interface**, **interrupt**, **note**, **notify**, **notifyone**, **out**, **par**, **pipe**, **piped**, **range**, **this**, **timing**, **trap**, **true**, **try**, **wait**, **waitfor**.

For future extensions, the following tokens are reserved. These keywords cannot be used in any SpecC program.

asm, **catch**, **class**, **const_cast**, **delete**, **dynamic_cast**, **explicit**, **export**, **friend**, **inline**, **mutable**, **namespace**, **new**, **operator**, **private**, **protected**, **public**, **reinterpret_cast**,

static_cast, **template**, **throw**, **typeid**, **typename**, **using**, **virtual**.

A.1.6 Tokens 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
```

```
declaring_list =
        declaration_specifier declarator initializer_opt
        | type_specifier declarator initializer_opt
        | declaring_list ',' declarator initializer_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 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
        | identifier
        | identifier '[' constant_expression ':' constant_expression ']'
        | identifier '[' constant_expression ']'
```

```
interface_declaration =
        interface_specifier ';'

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

interface_specifier =
        interface anyname

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

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
        | 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 =
        identifier ':'
        | identifier ':' cond_branch_list
        | identifier ':' '{' '}'
        | identifier ':' '{' cond_branch_list '}'

cond_branch_list =
        cond_branch
        | cond_branch_list cond_branch

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

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 ';'

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
```

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] 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.

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

[4] SpecC Open Technology Consortium, `www.specc.org`.

Index