SpecC Reference Compiler
(SCRC V1.1)
Software Architecture and Implementation

Rainer Dömer

Technical Report
September 17, 2001
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
doemer@cecs.uci.edu

Abstract

This report describes the software architecture and implementation of the SpecC Reference Compiler
(SCRC). The SCRC is an Open Source implementation of a compiler and simulator for the SpecC lan-
guage. As such, the SCRC is provided with full source code, ready to be used on most Unix-like platforms.

This report describes the organization of the SCRC software. In particular, the data structure and the
application procedural interface (API) of the SpecC Internal Representation (SIR) is addressed in detail
in this document. The SIR data structure is used by the SpecC Reference Compiler to build, store, and
output designs specified in the SpecC language.

The report documents in detail the SIR API which is implemented by use of C++ classes. For each
object class, its data fields and the public methods are listed and explained. In addition, this report contains
an example program that demonstrates the use of the API by generating a simple SpecC design example.
## Contents

1 Introduction .................................................. 1

2 The program flow of the SpecC reference compiler ........ 2

3 The internal design representation ............................ 2
   3.1 The design tree ........................................... 4
   3.2 The base classes .......................................... 4
   3.3 The API layers ............................................ 6

4 The SIR classes ................................................. 9
   4.1 Class SIR BitSlice ........................................ 9
   4.2 Class SIR BitSlices ....................................... 11
   4.3 Class SIR Constant ....................................... 13
   4.4 Class SIR Constraint ..................................... 17
   4.5 Class SIR Constraints ................................... 19
   4.6 Class SIR Design ......................................... 21
   4.7 Class SIR Event .......................................... 25
   4.8 Class SIR Events ........................................ 27
   4.9 Class SIR Exception ..................................... 29
   4.10 Class SIR Exceptions ................................... 33
   4.11 Class SIR Expression ................................... 35
   4.12 Class SIR Expressions .................................. 41
   4.13 Class SIR FileInfo ...................................... 43
   4.14 Class SIR FileList ...................................... 45
   4.15 Class SIR Import ....................................... 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.16 Class SIR ImportList</td>
<td>49</td>
</tr>
<tr>
<td>4.17 Class SIR Initializer</td>
<td>51</td>
</tr>
<tr>
<td>4.18 Class SIR Initials</td>
<td>53</td>
</tr>
<tr>
<td>4.19 Class SIR Label</td>
<td>55</td>
</tr>
<tr>
<td>4.20 Class SIR Labels</td>
<td>57</td>
</tr>
<tr>
<td>4.21 Class SIR LineInfo</td>
<td>59</td>
</tr>
<tr>
<td>4.22 Class SIR List</td>
<td>61</td>
</tr>
<tr>
<td>4.23 Class SIR ListElem</td>
<td>65</td>
</tr>
<tr>
<td>4.24 Class SIR Member</td>
<td>67</td>
</tr>
<tr>
<td>4.25 Class SIR Members</td>
<td>69</td>
</tr>
<tr>
<td>4.26 Class SIR Node</td>
<td>71</td>
</tr>
<tr>
<td>4.27 Class SIR Note</td>
<td>73</td>
</tr>
<tr>
<td>4.28 Class SIR Notes</td>
<td>77</td>
</tr>
<tr>
<td>4.29 Class SIR Parameter</td>
<td>79</td>
</tr>
<tr>
<td>4.30 Class SIR Parameters</td>
<td>81</td>
</tr>
<tr>
<td>4.31 Class SIR PortMap</td>
<td>83</td>
</tr>
<tr>
<td>4.32 Class SIR PortMaps</td>
<td>85</td>
</tr>
<tr>
<td>4.33 Class SIR Statement</td>
<td>87</td>
</tr>
<tr>
<td>4.34 Class SIR Statements</td>
<td>93</td>
</tr>
<tr>
<td>4.35 Class SIR Symbol</td>
<td>97</td>
</tr>
<tr>
<td>4.36 Class SIR Symbols</td>
<td>107</td>
</tr>
<tr>
<td>4.37 Class SIR SymbolPtr</td>
<td>113</td>
</tr>
<tr>
<td>4.38 Class SIR SymbolPtrs</td>
<td>115</td>
</tr>
<tr>
<td>4.39 Class SIR Transition</td>
<td>117</td>
</tr>
<tr>
<td>4.40 Class SIR Transitions</td>
<td>119</td>
</tr>
</tbody>
</table>
List of Figures

1. Program flow of the SpecC reference compiler. ........................................... 3
2. Generic SIR design tree in the SCRC .............................................................. 5
3. SIR API layers .................................................................................................. 6
4. SIR header files and their dependencies ............................................................ 146
5. SIR of the HelloWorld example: root tree ....................................................... 151
6. SIR of the HelloWorld example: behavior 'Main' ........................................ 152
7. SIR of the HelloWorld example: method 'main' ............................................ 153
List of Listings

class SIR BitSlice .................................................. 9

class SIR BitSlices ................................................. 11

class SIR Constant ................................................ 13

class SIR Constraint ............................................... 17

class SIR Constraints ............................................. 19

class SIR Design .................................................. 21

class SIR Event ................................................... 25

class SIR Events .................................................. 27

class SIR Exception .............................................. 29

class SIR Exceptions ............................................. 33

class SIR Expression ............................................. 35

class SIR Expressions ............................................ 41

class SIR FileInfo ................................................. 43

class SIR FileList ................................................. 45

class SIR Import .................................................. 47

class SIR ImportList ............................................. 49

class SIR Initializer ............................................... 51

class SIR Initials ................................................. 53

class SIR Label ................................................... 55

class SIR Labels .................................................. 57

class SIR LineInfo ............................................... 59

class SIR List ..................................................... 61

class SIR ListElem ............................................... 65
class SIR Member ......................................................... 67
class SIR Members ...................................................... 69
class SIR Node .......................................................... 71
class SIR Note ........................................................... 73
class SIR Notes .......................................................... 77
class SIR Parameter .................................................... 79
class SIR Parameters ................................................... 81
class SIR PortMap ....................................................... 83
class SIR PortMaps ..................................................... 85
class SIR Statement ..................................................... 87
class SIR Statements ................................................... 93
class SIR Symbol ....................................................... 97
class SIR Symbols ....................................................... 107
class SIR SymbolPtr .................................................... 113
class SIR SymbolPtrs .................................................. 115
class SIR Transition .................................................... 117
class SIR Transitions ................................................... 119
class SIR Type ............................................................ 121
class SIR Types ......................................................... 125
class SIR TypePtr ....................................................... 129
class SIR TypePtrs ...................................................... 131
class SIR Unit ............................................................ 133
class SIR UserType ..................................................... 135
class SIR UserTypes .................................................... 141
Test for Error ........................................................... 147

viii
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld example in SpecC</td>
<td>149</td>
</tr>
<tr>
<td>CreateHelloWorld.cc</td>
<td>154</td>
</tr>
<tr>
<td>HelloWorld.sc</td>
<td>160</td>
</tr>
</tbody>
</table>
1 Introduction

The SpecC Reference Compiler (SCRC) [SCRC] is an Open Source implementation of a compiler and simulator for the SpecC language [BOOK00, BOOK01].

The goal of the SpecC Reference Compiler is to:

- promote SpecC standardization,
- provide a reference implementation, that is
  - compliant with the SpecC Language Reference Manual (LRM) [LRM01], and
  - freely available,
- provide a basis for SpecC tool development.

The SCRC distribution consists of:

- a SpecC compiler (parser, internal representation, code generator)
- a SpecC simulator (run-time libraries for SpecC execution)
- a test suite (SpecC LRM compliance test cases)

Full source code is provided, ready to be used on most Unix-like environments. For convenience, pre-compiled binary distributions are also provided for popular platforms, including Linux and Solaris (see the download page at [SCRC] for details).

The first official version, version 1.0, of the SCRC was released on June 1, 2001. This was followed by a bug fix release, version 1.1, released on August 6, 2001. This report reflects the latest status, that is SCRC version 1.1.

The SpecC Reference Compiler is licensed under the BSD license, making it freely available to all interested parties, including full commercial usage by EDA companies.

The SCRC is developed by the Center for Embedded Computer Systems (CECS) at the University of California, Irvine (UCI) [CECS], as a project for the SpecC Technology Open Consortium (STOC) [STOC]. The SCRC project was approved by STOC in a general meeting in January 2001.

This report describes the software architecture and implementation of the SpecC Reference Compiler. In other words, it describes the organization of the SCRC software. In particular, the data structure and the application procedural interface (API) of the SpecC Internal Representation (SIR) is addressed in detail in this document. The SIR data structure is used by the SpecC Reference Compiler to build, store, and output designs specified in the SpecC language.

First, in Section 2, the program flow of the SpecC Reference Compiler is described. Then, Section 3 takes a closer look at the SIR data structure by introducing the tree of class objects which contains all data of a design in the SCRC.

In Section 4, this report documents in detail the SIR API which is implemented by use of C++ classes. For each object class, its data fields and the public methods are listed, along with a brief description
of the purpose and usage of the class and its API methods. Thus, the SIR API is documented and explained in detail.

In Section 5, the use of the SIR as a separate module to be used as a basis for new SpecC tools is described. Also, in Section 6, error handling in the SIR is discussed.

In addition, this report contains in Section 7 an example program that demonstrates the use of the API by generating a simple SpecC design example. In particular, the main portions of the SIR data structure are shown graphically for the example. Also, the C++ program code is listed demonstrating how the SIR API can be used to easily construct, modify and export SpecC designs.

Finally, Section 8 concludes this report with a brief summary and discussion of future work.

2 The program flow of the SpecC reference compiler

Figure 1 shows the program flow of scrc, the SpecC reference compiler. The default flow starts on the top of the graph. A file with suffix .sc (Design.sc) containing the SpecC source code of a design is processed by the Preprocessor producing a .si file (Design.si) which is then fed into the SpecC Parser. From the preprocessed SpecC code, the Parser builds the SIR data structure in memory which then can be accessed with the SIR API.

By default, the SIR data structure is used by the Translator to generate a C++ description for simulation (Design.cc and Design.h) which then is fed into the C++ Compiler (producing Design.o) and the Linker which finally builds an executable file (Design).

Alternatively, in order to create a readable text file from the internal SIR, the SIR can be processed by the Deparser which regenerates SpecC source code (Design.sc)

The next section describes in detail the contents of the central container, the SIR data structure and its API.

3 The internal design representation

The internal representation of a SpecC design is a complex data structure, which can be viewed as a graph. The nodes of the graph are represented by C++ class objects, whereas the edges are represented by C++ pointers.

The nodes in the SIR graph are of different types. For example, a node representing a behavior declaration is of type SIR.Symbol, whereas nodes representing statements and expressions are of type SIR.Statement and SIR.Expression, respectively. For each node type, a C++ class defines the data members and API methods available for the node. These SIR classes are described in detail in Section 4.

As shown below, Figure 2 lists the classes of SIR data structure that comes with the SCRC.
Figure 1: Program flow of the SpecC reference compiler.
The edges of a SIR graph, representing relations among the nodes, can also be classified into two groups, which will be called pointers and links. Although all edges are implemented as standard C++ pointers, it is important to distinguish them in the SIR data structure.

A pointer represents a containment relation of two objects. For example, a compound statement contains a list of statements and therefore there exists a pointer from the compound statement object to the header of the statement list (and also from the header of the list to the elements of the list).

A link represents a (loose) connection between two objects, that does not imply any containment. For example, expressions and symbols have a link to a node representing their type.

### 3.1 The design tree

The separation between pointers and links in the SIR allows to view the SCRC data structure as a generic tree. The SIR graph becomes a tree, if the edges classified as links are ignored and only pointer edges build the arcs between the nodes. Such a graph is called a design tree.

Figure 2 shows the generic tree representation of the SIR that comes with the SCRC.

The root of this tree is represented by an object of class `SIR_Design`, as shown in the figure. This root object contains a list of source files (`SIR_FileList`), a list of imported binary files (`SIR_ImportList`), the global type table (`SIR_Types`), the global symbol table (`SIR_Symbols`), and optionally a list of global annotations (`SIR_Notes`). Each of the lists then can contain list elements, which again can contain objects, and so on.

The design tree is used mainly for two purposes. First, whenever some sort of traversal is performed over the SIR data structure, the traversal is done on the design tree. All iterators provided by the classes operate on the design tree only (they follow all pointers, but never follow a link).

Second, many methods offered by the classes operate not only on the object itself, but also on the subtree below. For example, many destructor methods behave this way. So when, for example, a behavior (an object of class `SIR_Symbol`) is deleted, all its local variables and functions including their contents are deleted as well. In particular, when the root node of a design (class `SIR_Design`) is deleted, all the memory occupied by the SIR data structure for this design is freed.

### 3.2 The base classes

In order to keep the amount of source code for the SIR data structure implementation minimal, base classes are used whenever the same functionality is provided by different standard classes.

Almost all classes in the design trees are derived from the template classes `SIR_List` or `SIR_ListElem`. `SIR_List` represents (a header of) a double-linked list containing objects of class `SIR_ListElem`.
Figure 2: Generic SIR design tree in the SCRC
Also, all classes are derived from class `SIR::Unit` which provides very basic services (in the SCRC, the class `SIR::Unit` only contains a color marker). Furthermore, almost all classes are based on class `SIR::Node` which allows to store source code location information (file name and line number) with each object.

For more details on these base classes, please refer to Section 4.

### 3.3 The API layers

As shown in Figure 3, the SpecC environment is designed of several layers of design representations and interfaces. At the highest level, a Graphical User Interface (GUI) is used for interaction with the user. Alternatively, a Command Line Interface (CLI) to the SpecC tools is available which, for the advanced user, allows direct control of the tools and the use of script languages (shells). Both of these interfaces operate on the internal design representation, namely the SIR.

The SIR itself provides an Application Procedural Interface (API), as mentioned earlier. As shown in Figure 3, this SIR API consists of two layers, namely the Kernel (layer 1) and the Hierarchy Layer (layer 2).

The Kernel, as the innermost design representation, represents the lowest level of abstraction in terms of API methods. Use of these methods requires very detailed knowledge about the internals of the SIR data structure. Almost no semantic or syntactic error checking is performed. The user is in full responsibility for memory allocation/deallocation, insertion/removal of objects, and pointer consistency.

For the SpecC tool programmer, the Hierarchy Layer and its API provides the right level of abstraction. Here, the API layer 2 offers easy-to-use methods which guarantee consistency of the data structure after each call (even in case of errors). Guaranteed consistency means that the design is a valid and legal SpecC model at any time. Syntactic and semantic checking is performed for every call. If an operation fails, an error is reported to the caller and the SIR data structure is left unmodified (unless otherwise documented). Memory allocation/deallocation is automatically performed with the creation and deletion of objects.
As of this point in time, the SpecC reference compiler SCRC comes only with the API layer 1. This is sufficient for compilation purposes. API layer 2, the API for SpecC tool developers, is not made available at this time.

In the next section, every class of the SCRC is listed with all data members and methods. Although some of these represent very low-level information, most of them help in understanding the SIR data structure.
4 The SIR classes

In this section, a complete list of the classes used in the SpecC Internal Representation is given, along with a brief description of the purpose and the contents of each class.

4.1 Class SIR BitSlice

The class SIR_BitSlice is declared in file IntRep/PortMap.h.

4.1.1 Declaration:

```c
1 class SIR_BitSlice : /* bitslice in port mapping */
2     public SIR_ListElem<SIR_BitSlice>    /* is a list element */
3 {
4     public:
5     sir_symbol *Symbol;     /* link to symbol table entry (NULL if open) */
6     sir_constant *Constant; /* constant mapping (or NULL if open) */
7     int LeftBound,          /* bounds of slice (if BITVECTOR, else 0) */
8     RightBound;
9
10 //+++++++++++++++++++++++API Layer 1+++++++++++++++++++/
11
12
13 SIR_BitSlice( /* constructor #1 (standard) */
14     sir_symbol *Symbol,    /* NULL for open connection */
15     int LeftBound = 0,
16     int RightBound = 0);
17
18 SIR_BitSlice( /* constructor #2 (constant) */
19     sir_constant *Constant);
20
21 SIR_BitSlice( /* constructor #4 (duplicator) */
22     sir_bitslice *Original);
23
24 ~SIR_BitSlice(void);    /* destructor */
25
26 static sir_bitslice *New( /* create a new bitslice #1b */
27     sir_symbol *Mapping,   /* (returns NULL if SIR_Error) */
28     int LeftBound = 0,     /* bounds for bitvectors */
29     int RightBound = 0);   
30
31 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
32     sir_node_mptr MemberFct,
33```
4.1.2 Purpose:

An object of class SIR.BitSlice represents a mapping of an instance port or port slice. This mapping is either an open mapping (Symbol and Constant are NULL), a direct mapping onto a variable, channel or port (Symbol is valid, Constant is NULL), a mapping onto a constant (Constant is valid, Symbol is NULL), or one slice of a partial mapping onto a slice of a bit vector variable or port.

4.1.3 Notes:

- The link Symbol points to the symbol onto which this slice is mapped, or is NULL.
- The pointer Constant points to the constant onto which this slice is mapped, or is NULL.
- The members LeftBound and RightBound determine the left and right bounds of this slice. For a mapped symbol of non-bitvector type, LeftBound and RightBound are not used and should be set to 0.
- The method New creates a new bitslice object. Mapping specifies the target variable, channel or port. For sliced mappings, LeftBound and RightBound specify the left and right bounds of the slice.
- The method DFS.ForAllNodes performs a traversal over all nodes (this node and all below) in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.
4.2 Class SIR BitSlices

The class SIR_BitSlices is declared in file IntRep/PortMap.h.

4.2.1 Declaration:

```cpp
1 class SIR_BitSlices : /* list of (concatenated) bit slices */
2     public SIR_List<SIR_BitSlice> /* is simply a list of bit slices */
3     { /* with additional methods */
4 public:
5 //+++++++++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++++
6
7 SIR_BitSlices ( /* constructor #1 */
8     sir_bitslice *FirstEntry = NULL);
9
10 SIR_BitSlices ( /* constructor #2 (duplicator) */
11     sir_bitslices *Original);
12
13 ~SIR_BitSlices (void); /* destructor */
14
15 ERROR DFS_ForAllNodes ( /* iterator over all nodes (depth first) */
16     sir_node_ptr MemberFct,
17     sir_node_margin MemberFctArg);
18
19 void UnAlias (void); /* unalias all type, usertype, symbol links */
20 }
21```

4.2.2 Purpose:

An object of class SIR_BitSlices represents a list head. The list contains objects of type SIR_BitSlice.

4.2.3 Notes:

- The list of bit slices is sorted by concatenation order and can be traversed with the standard list operations.
- The method DFS_ForAllNodes performs a traversal over all nodes (this node and all below) in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.
4.3 Class SIR Constant

The class SIR_Constant is declared in file IntRep/Constant.h.

4.3.1 Declaration:

```cpp
1 class SIR_Constant : public SIR_Unit, public SIR_Node {
2     /* constant representation */
3     public:
4     /* is a unit of the design */
5     /* and a node */
6     union
7     {
8     bool B_Value; /* boolean constants */
9     char C_Value; /* character constants */
10    unsigned char UC_Value;
11    short int S_Value; /* integer constants */
12    unsigned short int US_Value;
13    int I_Value;
14    unsigned int UI_Value;
15    long int L_Value;
16    unsigned long int UL_Value;
17    LONG_LONG L_Value;
18    UNSIGNED_LONG_LONG ULL_Value;
19    float F_Value; /* floating constants */
20    double D_Value;
21    long double LD_Value;
22    sir_bit *BIT_Value; /* bit vector constant */
23    string *CS_Value; /* character string constant */
24    }
25    ;
26
27   //+++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++
28
29   SIR_Constant ( /* constructor #1 */
30       SIR_CONSTTYPE Type,
31       int Value,
32    unsigned int Line = 0,
33       sir_fileinfo *FileInfo = NULL);
34
35   SIR_Constant ( /* constructor #2 */
36       SIR_CONSTTYPE Type,
37  LONG_LONG Value,
38    unsigned int Line = 0,
39       sir_fileinfo *FileInfo = NULL);
40
41   SIR_Constant ( /* constructor #3 */
42       /* constructor */
43    );
44
45```

SIR_CONSTYPE Type,
UNSIGNED_LONG_LONG Value,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Constant ( /* constructor #4 */
SIR_CONSTYPE Type,
long double Value,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Constant ( /* constructor #5 */
SIR_CONSTYPE Type, /* (the string is consumed!) */
string *Value,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Constant ( /* constructor #6 */
SIR_CONSTYPE Type, /* (the string is copied!) */
const char *Value,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Constant ( /* constructor #7 (for bitvectors) */
sir_bit Value, /* (the bitvector is copied) */
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Constant ( /* constructor #8 */
sir_constant *Original); /* (duplicator) */

~SIR_Constant (void); /* destructor */

ERROR DFS_ForAllNodes ( /* iterator over all nodes (depth first) */
sir_node_mptra MemberFct,
sir_node_marg MemberFctArg);

sir_constant *Promoted(void); /* promoted constant */

sir_constant *Converted ( /* converted constant */
SIR_CONSTYPE Type, /* (supports all legal conversions) */
int Length = 0); /* for bitvecs; 0 is natural length */

static const char *PrettyChar ( /* returns a C-style character */
const char *Char);

static void PrettyString ( /* returns a C-style string */
string *Result,
4.3.2 Purpose:

An object of class SIR_Constant represents a SpecC constant.

4.3.3 Notes:

- The member Type represents the type of this constant and is one of the following values:
  SIR_CONST_BOOL, SIR_CONST_CHAR, SIR_CONST_UCHAR, SIR_CONST_SHORT, SIR_CONST_USHORT, SIR_CONST_INT,
  SIR_CONST_UINT, SIR_CONST_LONG, SIR_CONST_ULONG, SIR_CONSTONGLONG, SIR_CONST_ULONGLONG,
  SIR_CONST_FLOAT, SIR_CONST_DOUBLE, SIR_CONST_LONDDOUBLE, SIR_CONST_BIT, SIR_CONST_UBIT,
  SIR_CONST_CHARSTRING, SIR_CONST_TIME.

- Depending on Type, exactly one of the union members **XX_Value** is valid and contains the value of this constant.

- The method DFS_ForAllNodes performs a traversal over all nodes (this node and all below) in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.

- The method Converted converts this constant (and its value) to the new type Type. Only legal conversions are supported.

- The method Print generates a text representation of this constant in SpecC syntax. Optionally, CppNotation allows to generate C++ syntax instead.
4.4 Class SIR Constraint

The class SIR_Constraint is declared in file IntRep/Constraint.h.

4.4.1 Declaration:

```cpp
1 class SIR_Constraint : /* constraint */
2   public SIR_Node, /* is a node */
3   public SIR_ListElem<SIR_Constraint> /* and a list element */
4 {
5   public:
6   sir_label *Label1; /* first label entry */
7   sir_label *Label2; /* second label entry */
8   sir_constant *MinTime; /* minimum time (NULL if unspecified) */
9   sir_constant *MaxTime; /* maximum time (NULL if unspecified) */
10
11 //++++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++
13
14 SIR_Constraint( /* constructor #1 */
15     sir_label *Label1,
16     sir_label *Label2,
17     sir_constant *MinTime,
18     sir_constant *MaxTime,
19     unsigned int Line = 0,
20     sir_fileinfo *FileInfo = NULL);
22
23 SIR_Constraint( /* constructor #3 (duplicator) */
24     sir_constant *Original);
25
26 SIR_Constraint(void); /* destructor */
27
28 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
29     sir_node_mptr MemberFct,
30     sir_node_marg MemberFctArg);
32
```

4.4.2 Purpose:

An object of class SIR_Constraint represents a timing constraint. More specifically, it is used in the SpecC do-timing construct (a timing diagram) and represents a range statement.
4.4.3 Notes:

- The links \texttt{Label1} and \texttt{Label2} point to the labels of the \texttt{range} statement.

- The members \texttt{MinTime} and \texttt{MaxTime} contain the minimum and maximum times of the \texttt{range} statement. Either of them can be \texttt{NULL}, in which case this represents $-\infty$ and $+\infty$, respectively.

- The method \texttt{DFSForAllNodes} performs a traversal over all nodes (this node and all below) in depth-first search order. During the traversal, \texttt{MemberFct} is called for each node with the argument \texttt{MemberFctArg}.
4.5 Class SIR Constraints

The class SIR_Constraints is declared in file IntRep/Constraint.h.

4.5.1 Declaration:

```cpp
1 class SIR_Constraints : /* constraints list */
2   public SIR_List<SIR_Constraint> /* is simply a list of constraints */
3       /* with additional member(function)s */
4 public:
5
6 //+++++++++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++
7
8 SIR_Constraints ( /* constructor #1 */
9   sir_constraint *FirstEntry = NULL);
10
11 SIR_Constraints ( /* constructor #2 (duplicator) */
12   sir_constraints *Original);
13
14 ~SIR_Constraints (void); /* destructor */
15
16 ERROR DFS_ForAllNodes ( /* iterator over all nodes (depth first) */
17   sir_node_mptr MemberFct,
18   sir_node_marg MemberFctArg);
19
20 sir_constraint *Insert ( /* insert a new constraint */
21   sir_constraint *NewOne); /* (must not exist) */
22
23 sir_constraint *Find ( /* find the first constraint beginning here */
24   sir_label *Lab1); /* (returns NULL if not found) */
25
26 sir_constraint *Find ( /* find a constraint */
27   sir_label *Lab1, /* (returns NULL if not found) */
28
29 sir_label *Lab1, /* (returns NULL if not found) */
30   sir_label *Lab2);
31
32 }
```

4.5.2 Purpose:

An object of class SIR_Constraints represents a list head. The list contains objects of type SIR_Constraint.
4.5.3 Notes:

- The list of constraints is sorted by Label1 and declaration order, and can be traversed with the standard list operations.

- The method DFS_ForAllNodes performs a traversal over all nodes (this node and all below) in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.

- The methods Insert inserts a new constraint into the sorted list.

- The methods Find search for a constraint in this list by the label(s) Label1 (and Label2). NULL (but no error) is returned if no constraint is found.
4.6 Class SIR Design

The class SIR_Design is declared in file IntRep/Design.h.

4.6.1 Declaration:

```c
1 class SIR_Design : /* root of the data structure */
2    public SIR_Unit /* is a unit of the design */
3 {
4    public:
5        string Name; /* name of this design */
6    sir_file_list *FileList; /* list of (source) files */
7    sir_import_list *ImportList; /* list of imported (binary) files */
8    sir_types *Types; /* list of all types in the description */
9    sir_symbols *Symbols; /* global symbol table */
10   sir_notes *Notes; /* global notes (NULL if none) */
11
12    // ++++++++ API Layer 1 ++++++++/
13
14    SIR_Design( /* constructor #1 */
15        const char *Name);
16
17    SIR_Design( /* constructor #2 (duplicate) */
18        sir_design *Original);
19
20    ~SIR_Design( void); /* destructor */
21
22    ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
23        sir_node_mptr MemberFct,
24        sir_node_marg MemberFctArg);
25
26    ERROR DFS_ForAllSymbols( /* iterator over all symbols (depth first) */
27        sir_symbol_mptr MemberFct,
28        sir_symbol_marg MemberFctArg);
29
30    ERROR DFS_ForAllUserTypes( /* iterator over all usertypes (depth first) */
31        sir_usertp_mptr MemberFct,
32        sir_usertp_marg MemberFctArg);
33
34    ERROR DFS_ForAllNotes( /* iterator over all notes (depth first) */
35        sir_note_mptr MemberFct,
36        sir_note_marg MemberFctArg);
37
38    ERROR DFS_ForAllStatements( /* iterator over all statements (depth first) */
39        sir_stmt_mptr MemberFct,
40        sir_stmt_marg MemberFctArg);
41
42
43
```

4.6.2 Purpose:

An object of class SIRDesign represents one complete design model. It is the root of the design tree.

4.6.3 Notes:

- The member Name holds the name of this design.

- The member FileList contains the list of source code file names which contain the specification of this design.

- The member ImportList contains the list of imported files.
• The member **Types** contains the (global) type table of this design.

• The member **Symbols** contains the global symbol table of this design.

• The member **Notes** contains the annotations attached to this design.

• The method **DFS_ForAllNodes** performs a traversal over all nodes in the design in depth-first search order. During the traversal, **MemberFct** is called for each node with the argument **MemberFctArg**.

• The method **DFS_ForAllSymbols** performs a traversal over all symbols in the design in depth-first search order. During the traversal, **MemberFct** is called for each symbol with the argument **MemberFctArg**.

• The method **DFS_ForAllUserTypes** performs a traversal over all user-defined types in the design in depth-first search order. During the traversal, **MemberFct** is called for each user-defined type with the argument **MemberFctArg**.

• The method **DFS_ForAllNotes** performs a traversal over all annotations in the design in depth-first search order. During the traversal, **MemberFct** is called for each annotation with the argument **MemberFctArg**.

• The method **DFS_ForAllStatements** performs a traversal over all statements in the design in depth-first search order. During the traversal, **MemberFct** is called for each statement with the argument **MemberFctArg**.

• The method **WriteSC** generates SpecC source code from this design in the file specified by **Filename**.

• The method **WriteH** generates C++ header code for simulation of this design in the file specified by **Filename**.

• The method **WriteCC** generates C++ main code for simulation of this design in the file specified by **Filename**.

• The method **Annotate** attaches the annotation **NewNote** at this design.
4.7 Class SIR Event

The class SIREvent is declared in file IntRep/Symbol.h.

4.7.1 Declaration:

```c
1 typedef class SIR_SymbolPtr SIR_Event; /* alias type */
2 typedef class SIR_SymbolPtr sir_event; /* alias type */
```

4.7.2 Purpose:

An object of class SIREvent represents a reference to an event.

4.7.3 Notes:

- The class SIREvent is exactly the same as the class SIR_SymbolPtr. Please refer to Section 4.37 on page 113 for more information.
4.8 Class SIR Events

The class SIR_Events is declared in file IntRep/Symbol.h.

4.8.1 Declaration:

```c
1 typedef class SIR_SymbolPtrs SIR_Events; /* alias type */
2 typedef class SIR_SymbolPtrs sir_events; /* alias type */
```

4.8.2 Purpose:

An object of class SIR_Events represents a list head. The list contains objects of type SIR_Event.

4.8.3 Notes:

- The class SIR_Events is exactly the same as the class SIR_SymbolPtrs. Please refer to Section 4.38 on page 115 for more information.
4.9 Class SIR Exception

The class SIR_Exception is declared in file IntRep/Exception.h.

4.9.1 Declaration:

```cpp
1 class SIR_Exception : public SIR_Node, public SIR_ListElem<SIR_Exception> {
2   public:
3     SIR_EXCEPT_TYPE ExceptionType; /* type of the exception */
4     sir_symbol_ptrs *Events; /* events */
5     sir_statement *Handler; /* handler to go to */
6     unsigned int Line = 0,
7     sir_fileinfo *FileInfo = NULL;
8
9   SIR_Exception() /* constructor #1 */
10     SIR_EXCEPT_TYPE ExceptionType,
11     sir_symbol_ptrs *Events,
12     sir_statement *Handler,
13     unsigned int Line = 0,
14     sir_fileinfo *FileInfo = NULL;
15
16   SIR_Exception() /* constructor #2 (duplicator) */
17     sir_exception *Original;
18
19   SIR_Exception() /* destructor */
20
21   sir_exceptions *GetList(void); /* determines the list of this exception */
22   /* (returns NULL if not in a list) */
23
24   ERROR DFS_ForAllNodes(void) /* iterator over all nodes (depth first) */
25     sir_node_mptr MemberFct,
26     sir_node_marg MemberFctArg;
27
28   ERROR DFS_ForAllSymbols(void) /* iterator over all symbols (depth first) */
29     sir_symbol_mptr MemberFct,
30     sir_symbol_marg MemberFctArg;
31
32   ERROR DFS_ForAllUserTypes(void) /* iterator over all usertypes (depth first) */
33     sir_usertype_mptr MemberFct,
34     sir_usertype_marg MemberFctArg;
35
36   ERROR DFS_ForAllNotes(void) /* iterator over all notes (depth first) */
```
4.9.2 Purpose:

An object of class `SIR_Exception` represents an exception handler. More specifically, it is used for the SpecC `try - trap - interrupt` statement and represents either a `trap` or an `interrupt` statement.

4.9.3 Notes:

- The member `ExceptionType` determines the type of this exception. The type is either `SIR_EXCEPTION_TRAP` (abortion) or `SIR_EXCEPTION_INTERRUPT` (interrupt).
- The member `Events` contains the list of events which trigger this exception.
- The member `Handler` contains the statement to be executed in case of this exception.
- The method `GetList` returns a handle to the list of exceptions.
- The method `DFS_ForAllNodes` performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, `MemberFct` is called for each node with the argument `MemberFctArg`.
- The method `DFS_ForAllSymbols` performs a traversal over all symbols in this subtree in depth-first search order. During the traversal, `MemberFct` is called for each symbol with the argument `MemberFctArg`.

```c
sir_note_mptr  MemberFct,
sir_note_marg  MemberFctArg);

ERROR DFS_ForAllStatements( /* iterator over all statements (depth first) */
sir_stmt_mptr  MemberFct,
sir_stmt_marg  MemberFctArg);

ERROR DFS_ForAllExpressions( /* iterator over all expressions (dfs) */
sir_expr_mptr  MemberFct,
sir_expr_marg  MemberFctArg);

void SetAlias(        /* sets all type, usertype, symbol alias */
sir_exception *Alias); /* iterates over symbols and usertypes */

void UnAlias(        /* unalias all type, usertype, symbol links */
sir_symbols *GlobalSymbols);
```
The method DFS_ForAllUserTypes performs a traversal over all user-defined types in this subtree in depth-first search order. During the traversal, MemberFct is called for each user-defined type with the argument MemberFctArg.

The method DFS_ForAllNotes performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, MemberFct is called for each annotation with the argument MemberFctArg.

The method DFS_ForAllStatements performs a traversal over all statements in this subtree in depth-first search order. During the traversal, MemberFct is called for each statement with the argument MemberFctArg.

The method DFS_ForAllExpressions performs a traversal over all expressions in this subtree in depth-first search order. During the traversal, MemberFct is called for each expression with the argument MemberFctArg.
4.10 Class SIR Exceptions

The class SIR_Exceptions is declared in file IntRep/Exception.h.

4.10.1 Declaration:

```cpp
1 class SIR_Exceptions : /* exceptions list */
2   public SIR_List<SIR_Exception> /* is simply a list of exceptions */
3   { /* with additional member(function)s */
4   public:
5
6 //+++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++++/
7
8 SIR_Exceptions() /* constructor #1 */
9   sir_exception *FirstEntry = NULL);
10
11 SIR_Exceptions() /* constructor #2 (duplicator) */
12   sir_exceptions *Original);
13
14 ~SIR_Exceptions( void); /* destructor */
15
16 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
17   sir_node_mptr MemberFct,
18   sir_node_marg MemberFctArg);
19
20 ERROR DFS_ForAllSymbols( /* iterator over all symbols (depth first) */
21   sir_symbol_mptr MemberFct,
22   sir_symbol_marg MemberFctArg);
23
24 ERROR DFS_ForAllUserTypes( /* iterator over all usertypes (depth first) */
25   sir UserType_mptr MemberFct,
26   sir UserType_marg MemberFctArg);
27
28 ERROR DFS_ForAllNotes( /* iterator over all notes (depth first) */
29   sir note_mptr MemberFct,
30   sir note_marg MemberFctArg);
31
32 ERROR DFS_ForAllStatements( /* iterator over all statements (depth first) */
33   sir stmt_mptr MemberFct,
34   sir stmt_marg MemberFctArg);
35
36 ERROR DFS_ForAllExpressions( /* iterator over all expressions (dfs) */
37   sir expr_mptr MemberFct,
38   sir expr_marg MemberFctArg);
39
40 void UnAlias( /* unalias all type, usertype, symbol links */
41
42```
sir::symbols *GlobalSymbols);

void SetAlias(
    +/-'sets all type, usertype, symbol alias' +/-
    sir::exceptions *Alias); +/-'iterates over symbols and usertypes' +/-

4.10.2 Purpose:

An object of class SIR::Exceptions represents a list head. The list contains objects of type SIR::Exception.

4.10.3 Notes:

- The list of exceptions is sorted by priority (declaration order) and can be traversed with the standard list operations.

- The method DFS::ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.

- The method DFS::ForAllSymbols performs a traversal over all symbols in this subtree in depth-first search order. During the traversal, MemberFct is called for each symbol with the argument MemberFctArg.

- The method DFS::ForAllUserTypes performs a traversal over all user-defined types in this subtree in depth-first search order. During the traversal, MemberFct is called for each user-defined type with the argument MemberFctArg.

- The method DFS::ForAllNotes performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, MemberFct is called for each annotation with the argument MemberFctArg.

- The method DFS::ForAllStatements performs a traversal over all statements in this subtree in depth-first search order. During the traversal, MemberFct is called for each statement with the argument MemberFctArg.

- The method DFS::ForAllExpressions performs a traversal over all expressions in this subtree in depth-first search order. During the traversal, MemberFct is called for each expression with the argument MemberFctArg.
4.11 Class SIR Expression

The class SIR_Expression is declared in file IntRep/Expression.h.

4.11.1 Declaration:

```c++
class SIR_Expression : /* expression node */
{
public:
    SIREXPRTYPE ExprType;    /* type of this expression */
    *Constant;        /* constant value (for constants) */
    Symbol;           /* link to symbol entry (for identifiers) */
    *Arg1;            /* first argument (eg. unary expr.) */
    *Arg2;            /* second argument (eg. right argument) */
    *Arg3;            /* third argument (for ternary expr.) */
    *Args;            /* list of arguments (for n-ary expr.) */
    *TypeArg;         /* link to type argument (sizeof() or cast) */
    *Type;            /* link to result type of this expression */
    LeftBound;        /* slice bounds (for bitvector slicing) */
    RightBound;

    //+++++API Layer 1+++++++/

    SIR_Expression(          /* constructor #1 (for constants) */
                           ExprType,
                           *Constant,
                           *Type,
                           Line = 0,
                           *FileInfo = NULL);

    SIR_Expression(          /* constructor #2 (for identifiers) */
                           ExprType,
                           Symbol,
                           Line = 0,
                           *FileInfo = NULL);

    SIR_Expression(          /* constructor #3 */
                           ExprType,        /* (for std. expr. with 0-3 args.) */
                           *Type,
                           *Arg1 = NULL,
                           *Arg2 = NULL,
                           *Arg3 = NULL,
                           Line = 0,
                           *FileInfo = NULL);
```

35
SIR_Expression (       /* constructor #4 */
  SIR_EXPRTYPE ExprType,     /* (for function calls) */
  sir_type *Type,
  sir_expression *Arg1,
  sir_expressions *Args,
  unsigned int Line = 0,
  sir_fileinfo *FileInfo = NULL);

SIR_Expression (       /* constructor #5 */
  SIR_EXPRTYPE ExprType,     /* (for member access) */
  sir_type *Type,
  sir_expression *Arg1,
  sir_symbol *Symbol,       /* (Symbol==NULL for bhvr. main()) */
  unsigned int Line = 0,
  sir_fileinfo *FileInfo = NULL);

SIR_Expression (       /* constructor #6 */
  SIR_EXPRTYPE ExprType,     /* (for sizeof(type) or casting) */
  sir_type *Type,
  sir_type *TypeArg,
  sir_expression *Arg1 = NULL,
  unsigned int Line = 0,
  sir_fileinfo *FileInfo = NULL);

SIR_Expression (       /* constructor #7 */
  SIR_EXPRTYPE ExprType,     /* (for bit slicing) */
  sir_type *Type,
  sir_expression *Arg1,
  int LeftBound,
  int RightBound,
  unsigned int Line = 0,
  sir_fileinfo *FileInfo = NULL);

SIR_Expression (       /* constructor #9 (duplicator) */
  sir_expression *Original);  /* (recursive!) */

~SIR_Expression (void);      /* destructor */

static sir_expression *New(   /* creates an identifier expression (#2b) */
  sir_symbol *Symbol);  /* (returns NULL if SIR_Error) */

static sir_expression *New(   /* creates a special expression (#8b) */
  SIR_EXPRTYPE ExprType,      /* (supports VOID, THIS) */
  sir_types *TypeTable,  /* (returns NULL if SIR_Error) */
  sir_symbol *ClassSymbol = NULL);  /* class for THIS (or NULL) */

BOOL IsLvalue (void);      /* checks if this expr. is an lvalue */

BOOL IsModifiableLvalue (void); /* checks if this expr. is a modifiable lv. */
95 BOOL IsWritable (void);         /* checks if this expr. allows write access */
96 BOOL IsReadable (void);        /* checks if this expr. allows read access */
100 ERROR CheckWriteAccess (void); /* checks for error regarding write access */
101 ERROR CheckReadAccess (void); /* checks for error regarding read access */
104 BOOL IsDependant (            /* checks if this expr. depends on symbol */
105        sir_symbol *ThatSymbol);  
108 ERROR DFS_ForAllNodes (       /* iterator over all nodes (depth first) */
109        sir_node_nptr MemberFct,
110        sir_node_marg MemberFctArg); 
112 ERROR DFS_ForAllExpressions (/* iterator over all expressions (dfs) */
113        sir_expr_nptr MemberFct,
114        sir_expr_marg MemberFctArg); 
116 BOOL DFS_FindDependant (      /* searches for dependants (depth first) */
117        sir_symbol *ThatSymbol,
118        sir_expression **DepExpr); 
121 ERROR MarkUsedTypes (        /* marks the type entries used here */
122        sir_expr_marg /* Unused */);  
125 void UnAlias (void);          /* unalias all type, usertype, symbol links */
128 static void ExplicitTypeConv( /* perform explicit type conversion */
129        SIR_CONSTTYPE FromType,
130        SIR_CONSTTYPE ToType,
131        const char **Prefix,
132        const char **Suffix);        /* (caller must substitute length !) */
134 ERROR WriteSC (               /* (re-) generates SpecC source code */
135        FILE *File,
136        BOOL WriteNotes;
137        BOOL CplusplusMode = FALSE,
138        SIR_TYPETYPE ExpectedType = SIR_TYPETYPE_ANY_TYPE,
139        int ExpectedLength = 0);        /* for expected bitvectors */
141 ERROR WriteSC2 (              /* (re-) generates SpecC source code */
142        const char *Op,        /* (for binary ops only) */
143        FILE *File,
144        BOOL WriteNotes,
145        BOOL CplusplusMode = FALSE,
SIR_TYPE Type ExpectedType1 = SIR_TYPE_ANY_TYPE,
SIR_TYPE Type ExpectedType2 = SIR_TYPE_ANY_TYPE,
int ExpectedLength1 = 0, /* for expected bit vectors */
int ExpectedLength2 = 0; /* for expected bit vectors */
sir_constant *Eval(void); /* evaluate a constant expression */
    /* (returns a new Constant */
    /* or NULL and SIR_Error */
int IntegerEval(void); /* evaluate a constant expression to integer */
    /* (may set SIR_Error */
sir_lineinfo *GetFirstLineInfo( /* obtain first line info of the tree */
    void);

static sir_expression *New( /* creates a constant expression (#1) */
    sir_constant *NewConst, /* (returns NULL if SIR_Error */
    sir_types *TypeTable);
static sir_expression *New( /* creates an expr. with 1–3 arguments (#3) */
    SIR_EXPRTYPE ExprType, /* (returns NULL if SIR_Error */
    sir_expression *NewArg1,
    sir_expression *NewArg2 = NULL,
    sir_expression *NewArg3 = NULL);
static sir_expression *New( /* creates a function call expression (#4) */
    sir_expression *NewArg1, /* (returns NULL if SIR_Error */
    sir_expressions *NewArgs);
static sir_expression *New( /* creates a member access expression (#5) */
    SIR_EXPRTYPE ExprType, /* (returns NULL if SIR_Error */
    sir_expression *NewArg1,
    const char *MemberName);
static sir_expression *New( /* creates a sizeof(type) or cast expr. (#6) */
    SIR_EXPRTYPE ExprType, /* (returns NULL if SIR_Error */
    sir_type *TypeArg,
    sir_expression *NewArg1 = NULL);
static sir_expression *New( /* creates a bit slice expression (#7) */
    sir_expression *NewArg1, /* (returns NULL if SIR_Error */
    int LeftBound,
    int RightBound);
}

### 4.11.2 Purpose:

An object of class SIR_Expression represents an expression (tree).
4.11.3 Notes:

- The member ExprType determines the type of this expression. It is one of the following values: SIR_EXPR_VOID, SIR_EXPR_CONSTANT, SIR_EXPR_IDENTIFIER, SIR_EXPR_PARENTHESIS, SIR_EXPR_THIS, SIR_EXPRARRAY_ACCESS, SIR_EXPR_FUNCTION_CALL, SIR_EXPR_MEMBER_ACCESS, SIR_EXPR_MEMBER_POINTER, SIR_EXPR_POST_INCREMENT, SIR_EXPR_POST_DECREMENT, SIR_EXPR_BITSlice, SIR_EXPR_PRE_INCREMENT, SIR_EXPR_PRE_DECREMENT, SIR_EXPR_ADDRESS_OF, SIR_EXPR_CONTENT_OF, SIR_EXPR_POSITIVE, SIR_EXPR_NEGATIVE, SIR_EXPR_NOT, SIR_EXPR_LOGICAL_NOT, SIR_EXPR_SIZEOF_EXPR, SIR_EXPR_SIZEOF_TYPE, SIR_EXPR_TYPE_CONVERSION, SIR_EXPR CONCATENATION, SIR_EXPR_MULTIPLY, SIR_EXPR_DIVIDE, SIR_EXPR_MODULO, SIR_EXPR_ADD, SIR_EXPR_SUBTRACT, SIR_EXPR SHIFT_LEFT, SIR_EXPR SHIFT_RIGHT, SIR_EXPR LESS, SIR_EXPR GREATER, SIR_EXPR LESS_EQUAL, SIR_EXPR GREATER_EQUAL, SIR_EXPR_EQUAL, SIR_EXPR NOT_EQUAL, SIR_EXPR AND, SIR_EXPR OR, SIR_EXPR LOGICAL AND, SIR_EXPR LOGICAL OR, SIR_EXPR CONDITION, SIR_EXPR ASSIGNMENT, SIR_EXPR_MUL_ASSIGN, SIR_EXPR DIV_ASSIGN, SIR_EXPR MOD_ASSIGN, SIR_EXPR_ADD_ASSIGN, SIR_EXPR_SUB_ASSIGN, SIR_EXPR SHL_ASSIGN, SIR_EXPR SHR_ASSIGN, SIR_EXPR AND_ASSIGN, SIR_EXPR EOR_ASSIGN, SIR_EXPR OR_ASSIGN, SIR_EXPR COMMA.

- The member Constant is a pointer to the constant for SIR_EXPR_CONSTANT, otherwise this is NULL.

- The link Symbol points to a symbol for SIR_EXPR_IDENTIFIER, SIR_EXPR_MEMBER_ACCESS and SIR_EXPR_MEMBER_POINTER, otherwise this is NULL.

- The member Arg1 is a pointer to the first (left) sub-expression of this expression, or NULL if not applicable.

- The member Arg2 is a pointer to the second (right) sub-expression of this expression, or NULL if not applicable.

- The member Arg3 is a pointer to the third sub-expression of this expression, or NULL if not applicable.

- The member Args is a pointer to the list of arguments for SIR_EXPR_FUNCTION_CALL, or NULL otherwise.

- The member TypeArg is a link to the specified type for SIR_EXPR_SIZEOF_TYPE and SIR_EXPR_TYPE_CONVERSION, or NULL otherwise.

- The member Type is a link to the result type of this expression.

- The members LeftBound and RightBound are used for SIR_EXPR_BITSlice, otherwise these are 0.
The method IsLvalue checks whether this expression is an lvalue.

The method IsModifiableLvalue checks whether this expression is a modifiable lvalue.

The method IsWritable checks whether this expression can be used as target for a write access.

The method IsReadable checks whether this expression can be used as target for a read access.

The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.

The method DFS_ForAllExpressions performs a traversal over all expressions in this subtree in depth-first search order. During the traversal, MemberFct is called for each expression with the argument MemberFctArg.

The method WriteSC generates SpecC (or C++) source code for this expression tree.

The method Eval evaluates this (constant) expression and returns a (newly created) constant, or NULL in case of an error.

The method IntegerEval evaluates this (constant) expression to an integer. SIR_Error will be set to indicate any errors.

The method EvaluateToInt evaluates this (constant) expression to an integer (int) which is stored in ResultPtr.

The methods New create a new expression object. The new object is returned, or NULL in case of an error. Please refer to the comments in the class declaration to determine which of the overloaded methods to use for which expression type.
4.12 Class SIR Expressions

The class SIR_Expressions is declared in file IntRep/Expression.h.

4.12.1 Declaration:

```cpp
1 class SIR_Expressions : /* list of expressions */
2     public SIR_List<SIR_Expression>/* is simply a list of expressions */
3     { /* with additional methods */
4     public :
5     //++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++
6     SIR_Expressions ( /* constructor #1 */
7     sir_expression *NewFirstEntry = NULL);
8     SIR_Expressions ( /* constructor #2 (duplicator) */
9     sir_expressions *Original); /* (recursive !) */
10    ~SIR_Expressions (void); /* destructor */
11   
12    ERROR DFS_ForAllNodes ( /* iterator over all nodes (depth first) */
13    sir_node_mptr MemberFct,
14    sir_node_marg MemberFctArg);
15   
16    ERROR DFS_ForAllExpressions ( /* iterator over all expressions (dfs) */
17    sir_expr_mptr MemberFct,
18    sir_expr_marg MemberFctArg);
19  }
```

4.12.2 Purpose:

An object of class SIR_Expressions represents a list head. The list contains objects of type SIR_Expression.

4.12.3 Notes:

- The list of expressions is sorted by declaration order and can be traversed with the standard list operations.

- The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.
• The method `DFS_ForAllExpressions` performs a traversal over all expressions in this subtree in depth-first search order. During the traversal, `MemberFct` is called for each expression with the argument `MemberFctArg`. 
4.13 Class SIRFileInfo

The class SIRFileInfo is declared in file IntRep/FileInfo.h.

4.13.1 Declaration:

```cpp
1 class SIR_FileInfo : /* (source) file information */
2     public SIR_ListElem<SIR_FileInfo> /* is a list element */
3 {
4 public:
5     string Filename; /* file name */
6
7     sir_fileinfo *Alias; /* alias pointer (temporary only) */
8
9 //++++++++++++++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++++++++++++++/
11
12 SIR_FileInfo ( /* constructor #1 */
13     const char *Filename);
15 SIR_FileInfo ( /* constructor #3 (duplicator) */
17     sir_fileinfo *Original);
18
19 ~SIR_FileInfo (void); /* destructor */
20 );
```

4.13.2 Purpose:

An object of class SIRFileInfo represents the name of a SpecC source file.

4.13.3 Notes:

- The member Filename contains the file name of the SpecC source file represented by this object.
- The link Alias is used internally.
4.14 Class SIR FileList

The class SIR/FileList is declared in file IntRep/FileInfo.h.

4.14.1 Declaration:

```cpp
1 class SIR_FileList : /* list of (source) files */
2     public SIR_List<SIR_FileInfo> /* is a basically a list */
3 { /* with additional functions */
4 public:
5
6 //++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++/
7
8 SIR_FileList( /* constructor #1 */
9     sir_fileinfo *NewFirstEntry = NULL);
10
11 SIR_FileList( /* constructor #2 (duplicator) */
12     sir_file_list *Original);
13
14 ~SIR_FileList(void); /* destructor */
15
16 ERROR Integrate( /* integrates imported filelist */
17     sir_file_list *Imported);
18
19 void GarbageCollection( /* garbage collector */
20     sir_design *Design);
21
22 sir_fileinfo *FindOrInsert( /* find an entry or insert it */
23     const char *Filename); /* if it not exists */
24
25 sir_fileinfo *Find( /* find an entry */
26     const char *Filename); /* (returns NULL if not found) */
27 }
```

4.14.2 Purpose:

An object of class SIR/FileList represents a list head. The list contains objects of type SIR/FileInfo.

4.14.3 Notes:

- The list of source files is sorted in alphabetical order and can be traversed with the standard list operations.
• The method **FindOrInsert** inserts the given source file into this list, unless it exists already. In any case, a pointer to the entry (new or found) is returned.

• The method **Find** searches for a source file in this list by **Filename**. **NULL** (but no error) is returned if no source file is found.
4.15 Class SIR_Import

The class SIR_Import is declared in file IntRep/Import.h.

4.15.1 Declaration:

```cpp
1 class SIR_Import : /* import file */
2 public SIR_ListElem<SIR_Import> /* is a list element */
3 {
4 public:
5 string ImportName; /* design name */
6
7 sir_import *Alias; /* alias pointer (temporary only) */
8
9 //+++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++/
11
12 SIR_Import( /* constructor #1 */
13 const char *ImportName);
14
15 SIR_Import( /* constructor #3 (duplicator) */
16 sir_import *Original);
17
18 ~SIR_Import( void); /* destructor */
20
22 void TakeOverAndRemove( /* takes over these imports into the design */
23 sir_design *Design); /* and removes this import entry */
24 ");
```

4.15.2 Purpose:

An object of class SIR_Import represents the name of a design imported with the SpecC import construct.

4.15.3 Notes:

- Objects of class SIR_Symbol, SIR_UsersType and SIR_Note contain a link to a SIR_Import object denoting their origin if they were imported from a binary SIR file.

- The member ImportName contains the imported design name.

- The link Alias is used internally.
4.16 Class SIR ImportList

The class SIR_ImportList is declared in file IntRep/Import.h.

4.16.1 Declaration:

```cpp
1 class SIR_ImportList : /* list of imported files */
2    public SIR_List<SIR_Import> /* is a basically a list */
3    { /* with additional functions */
4 public:
5
6 //++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++
7
8 SIR_ImportList( /* constructor #1 */
9    sir_import *NewFirstEntry = NULL);
10
11 SIR_ImportList( /* constructor #2 (duplicator) */
12    sir_import_list *Original);
13
14 ~SIR_ImportList(void); /* destructor */
15
16 ERROR Integrate( /* merges with imported import list */
17    sir_import_list *ImportedList);
18
19 void GarbageCollection( /* garbage collector */
20    sir_design *Design);
21
22 sir_import *FindOrCreate( /* find an entry or insert it */
23    const char *ImportName); /* if it not exists */
24
25 sir_import *Insert( /* insert a new entry */
26    const char *ImportName);
27
28 sir_import *Find( /* find an entry */
29    const char *ImportName);
30
31 }
32
33 }
34
```

4.16.2 Purpose:

An object of class SIR_ImportList represents a list head. The list contains objects of type SIR_Import.
4.16.3 Notes:

- The list of imported designs is sorted in alphabetical order and can be traversed with the standard list operations.

- The method \texttt{FindOrInsert} inserts the given import file into this list, unless it exists already. In any case, a pointer to the entry (new or found) is returned.

- The method \texttt{Insert} inserts the given import file into this list. The entry must not exist before. A pointer to the new entry is returned.

- The method \texttt{Find} searches for a imported design in this list by \texttt{ImportName}. \texttt{NULL} (but no error) is returned if no imported design is found.
4.17 Class SIR Initializer

The class SIRInitializer is declared in file IntRep/Initializer.h.

4.17.1 Declaration:

```c
1 class SIR_Initializer : /* initializer of a variable */
2 public SIR_ListElem<SIR_Initializer> /* is a list element */
3 {
4 public:
5   sir_initials *InitList; /* either: list of initializers */
6   sir_constant *Initializer; /* or: one constant */
7
8 //++++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++++++++++++++++++++/
9
10 SIR_Initializer ( /* constructor #1 */
11   sir_initials *InitList,
12   sir_constant *Initializer );
13
14 SIR_Initializer ( /* constructor #3 (duplicator) */
15   sir_initializer *Original );
16
17 ~SIR_Initializer (void); /* destructor */
18
19 ERROR DFS_ForAllNodes ( /* iterator over all nodes (depth first) */
20   sir_node_nptr MemberFct,
21   sir_node_marg MemberFctArg);
22
23 ERROR WriteSC ( /* (re-) generates SpecC source code */
24   FILE *File,
25   BOOL CplusplusMode );
26 }
```

4.17.2 Purpose:

An object of class SIRInitializer represents the initial value of variables that are explicitly initialized.

4.17.3 Notes:

- An object of class SIRInitializer contains either a constant or a list of initializers.
- The member InitList contains the list of initializers, or NULL.
• The member **Initializer** contains the constant, or **NULL**.

• The method **DFS_ForAllNodes** performs a traversal over all nodes (this node and all below) in depth-first search order. During the traversal, **MemberFct** is called for each node with the argument **MemberFctArg**.

• The method **WriteSC** generates **SpecC** or **C++** source code for this initializer.
4.18 Class SIR Initials

The class SIR_Initials is declared in file IntRep/Initializer.h.

4.18.1 Declaration:

```cpp
1 class SIR_Initials : /* list of initializers */
2     public SIR_List<SIR_Initializer> /* is simply a list of initializers */
3 { /* with additional methods */
4 public:
5 //+++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++
7 9 SIR_Initials ( *sir_initializer *NewFirstEntry = NULL);
10     /* constructor #1 */
11 12 SIR_Initials ( *sir_initials *Original); /* (returns NULL if SIR_Error) */
13 14 ~ SIR_Initials ( void); /* destructor */
16 17 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
19     *sir_node_ptr MemberFct,
20     *sir_node marg MemberFctArg);
22 );
```

4.18.2 Purpose:

An object of class SIR_Initials represents a list head. The list contains objects of type SIR_Initializer.

4.18.3 Notes:

- The list of initializers is sorted by declaration order and can be traversed with the standard list operations.

- The method DFS_ForAllNodes performs a traversal over all nodes (this node and all below) in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.
4.19 Class SIR Label

The class SIR_Label is declared in file IntRep/Label.h.

4.19.1 Declaration:

```cpp
class SIR_Label {
   /* statement label */
   public SIRListElem<SIR_Label> /* is a list element */
};

class SIR_Label {
   /* constructor #1 */
   const char *LabelName; /* identifier of the label */
   sir_statement *Statement; /* link to labeled stmt. (NULL if unknown) */
   sir_notes *Notes; /* notes for this label (NULL if none) */

   // API Layer 1

   SIR_Label(); /* constructor #2 (duplicator) */
   ~SIR_Label(); /* destructor */

   ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
      sir_node_mptr MemberFct,
      sir_node_marg MemberFctArg);

   ERROR DFS_ForAllNotes( /* iterator over all notes (depth first) */
      sir_node_mptr MemberFct,
      sir_node_marg MemberFctArg);

   ERROR Annotate( /* attach a note to the label */
      sir_note *NewNote); /* (consumes NewNote) */
};
```

4.19.2 Purpose:

An object of class SIR_Label represents a label attached to a statement.
4.19.3 Notes:

- The member **LabelName** contains the name of this label.
- The member **Statement** is a link to the labeled statement.
- The member **Notes** contains the annotations attached to this label, or is **NULL** if there are none.
- The method **DFS_ForAllNodes** performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, **MemberFct** is called for each node with the argument **MemberFctArg**.
- The method **DFS_ForAllNotes** performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, **MemberFct** is called for each annotation with the argument **MemberFctArg**.
- The method **Annotate** annotates the label with **NewNote**.
4.20 Class SIR Labels

The class SIR_Labels is declared in file IntRep/Label.h.

4.20.1 Declaration:

```cpp
class SIR_Labels : /* list of all labels in a function */
  public SIR_List<SIR_Label> /* is simply a list of labels */
  {
    /* with additional member(function)s */
  public:

  //+++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++ //

  SIR_Labels (void); /* constructor #1 (empty label list) */

  SIR_Labels (sir_label *Original); /* constructor #2 (duplicator) */

  ~SIR_Labels (void); /* destructor */

  ERROR DFS_ForAllNodes (/* iterator over all nodes (depth first) */
    sir_node_nptr MemberFct,
    sir_node_marg MemberFctArg);

  ERROR DFS_ForAllNotes (/* iterator over all notes (depth first) */
    sir_node_nptr MemberFct,
    sir_node_marg MemberFctArg);

  sir_label *Define( /* define a label */
    const char *LabelName, /* (might return NULL and SIR_Error) */
    sir_statement *Statement );

  ERROR CheckDefStatements (void); /* check if all labels are defined */

  void DeleteMarkedLabels (void); /* deletes the marked labels from the list */

  sir_label *FindOrInsert ( /* find or insert a label */
    const char *LabelName);

  sir_label *Insert ( /* insert a new label */
    sir_label *Label);)

  sir_label *Find ( /* find a label */
    const char *LabelName); /* (returns NULL if not found) */
```

57
4.20.2 Purpose:

An object of class \texttt{SIR\_Labels} represents a list head. The list contains objects of type \texttt{SIR\_Label}.

4.20.3 Notes:

- The list of labels is sorted in alphabetical order and can be traversed with the standard list operations.
- The method \texttt{DFS\_ForAllNodes} performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, \texttt{MemberFct} is called for each node with the argument \texttt{MemberFctArg}.
- The method \texttt{DFS\_ForAllNotes} performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, \texttt{MemberFct} is called for each annotation with the argument \texttt{MemberFctArg}.
- The method \texttt{Define} defines the given label for the given statement. The label is inserted into this list. If the label exists already, \texttt{NULL} and an error code is returned.
- The method \texttt{FindOrInsert} inserts the given label into this list, unless it exists already. In any case, a pointer to the entry (new or found) is returned.
- The method \texttt{Insert} inserts the given label into this list. The entry must not exist before. A pointer to the new entry is returned.
- The method \texttt{Find} searches for a label in this list by \texttt{LabelName}. \texttt{NULL} (but no error) is returned if no label is found.
4.21 Class SIR LineInfo

The class SIR.LineInfo is declared in file IntRep/FileInfo.h.

4.21.1 Declaration:

```c++
1 class SIR_LineInfo : /* (source) line information */
2     public SIR_Unit /* is a unit of the design */
3 {
4     public:
5     static BOOL Enabled; /* internal stuff */
6     static unsigned int LastLine;
7     static sir_fileinfo *LastFile;
8     static int Tabulator;
9     static const char *CurrentFile;
10    static unsigned int CurrentLine;
11
12    unsigned int Line; /* line number */
13    sir_fileinfo *File; /* link to file information */
14
15    //+++++++++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++
16
17    SIR_LineInfo( /* constructor #1 */
18         unsigned int Line,
19         sir_fileinfo *File);
20
21    ~SIR_LineInfo( void); /* destructor */
22
23    static void InitWriteSC( /* prepares for writing new SpecC line infos */
24         bool Enable,
25         const char *ThisFilename = NULL, /* default location if */
26         unsigned int ThisLineNumber = 1); /* line info unknown */
27
28    ERROR WriteSC( /* writes line and file in SpecC source */
29         FILE *OutFile,
30         bool PutSpace = FALSE);
31
32    static void WriteNL( /* writes a newline in SpecC source */
33         FILE *OutFile); /* (regardless of any line info) */
34
35    static void WriteSPACE( /* writes a space in SpecC source */
36         FILE *OutFile); /* (regardless of any line info) */
37
38    static void WriteNewSC( /* writes unknown line info */
39         FILE *OutFile, /* (probably newly created stuff) */
40         bool PutSpace = FALSE);
```

59
44 static void Add2Tabulator ( /* increments tabulator for indenting */
45 int Tabs);  /* (decrements for Tabs < 0) */
47 }

4.21.2 Purpose:

An object of class SIR.LineInfo represents a source line information for a node in the SIR data structure.

4.21.3 Notes:

- Although declared public, the static members Enabled, LastLine, LastFile, Tabulator, CurrentFile and CurrentLine should be considered private. These are used internally only.

- The member Line contains the line number in the source file.

- The link File points to the source file.

- The provided methods are used internally during code generation.
4.22 Class SIR List

The class SIR_List is a base class that is declared in file IntRep/List.h.

4.22.1 Declaration:

```cpp
1 template <class item_type> /* template class for */
2 class SIR_List : /* any list header */
3 public SIR_Unit /* is a unit of the design */
4 {
5 item_type *FirstItem; /* internal data */
6 item_type *LastItem;
7 item_type *CurItem;
8 unsigned int NumberItems;
9
10 public:
11
12 //++++++++++++++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++++++++++++++
13
14 SIR_List ( /* constructor #1 */
15 item_type *FirstElement = NULL);
16
17 ~SIR_List (void); /* destructor */
18
19 bool Empty (void); /* test for empty list? */
20
21 unsigned int NumElements (void); /* number of list elements */
22
23 item_type *First (void); /* first element (NULL if empty) */
24
25 item_type *Last (void); /* last element (NULL if empty) */
26
27 item_type *Previous (void); /* previous element (NULL if none) */
28
29 item_type *Curr (void); /* current element (NULL if none) */
30
31 item_type *Next (void); /* next element (NULL if none) */
32
33 item_type *Prepend ( /* prepend one element */
34 item_type *Elem);
35
36 item_type *Append ( /* append one element */
37 item_type *Elem);
38
39 item_type *InsertBefore ( /* insert before current (#1) */
40 item_type *Elem);
41
42
43
61
```
44 item_type * InsertBefore ( item_type *Elem, item_type *Succ );
47
48 item_type * InsertAfter ( item_type *Elem );
50
51 item_type * InsertAfter ( item_type *Elem, item_type *Pred );
54
55 item_type * Remove( item_type *Elem );
58 SIR_ListElem <item_type> *Remove( SIR_ListElem <item_type> *Elem );
61 item_type *Remove( void );
63 SIR_List <item_type> * Concat( SIR_List <item_type> *Appendix );
65 SIR_List <item_type> * Precat ( SIR_List <item_type> *Prependix );
68 );

4.22.2 Purpose:

The class SIR_List is a template class for a list of elements. It is used as a base class for all list heads in the SIR data structure.

4.22.3 Notes:

- The template class SIR_List implements a double-linked list. This is the head of the list.

- The private members FirstItem, LastItem, CurrItem and NumberOfItems are used internally only.

- The method Empty determines whether this list is empty.

- The method NumElements returns the number of elements in this list.

- The method First returns the first element of this list, or NULL (no error) if the list is empty.

- The method Last returns the last element of this list, or NULL (no error) if the list is empty.

- The method Previous returns the previous element of this list, or NULL (no error) if not available.
• The method `Curr` returns the current element of this list, or `NULL` (no error) if not available.

• The method `Next` returns the next element of this list, or `NULL` (no error) if not available.

• The method `Prepend` inserts `Elem` at the beginning of this list.

• The method `Append` inserts `Elem` at the end of this list.

• The methods `InsertBefore` insert `Elem` in this list directly before the current element, or directly before `Succ`, respectively.

• The methods `InsertAfter` insert `Elem` in this list directly after the current element, or directly after `Pred`, respectively.

• The methods `Remove` remove `Elem` (or the current element, respectively) from this list. The removed element is not deleted.

• The method `Concat` concatenates the list `Appendix` to this list.

• The method `Precat` inserts the list `Prependix` at the beginning of this list.
4.23  Class SIR ListElem

The class SIR_ListElem is a base class that is declared in file IntRep/List.h.

4.23.1  Declaration:

```cpp
template <class item_type>  // template class for *
class SIR_ListElem :  // any list element */
  public SIR_Unit  // is a unit of the design */
{
  friend class SIR_List<item_type>;

  item_type *SuccItem;  // internal data */
  item_type *PredItem;
  SIR_List<item_type> *ListHead;

  public:

  //+++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++/

  SIR_ListElem (void);  // constructor #1 */
  ~SIR_ListElem (void);  // destructor */

  item_type *Succ(void);  // Successor */
  item_type *Pred(void);  // Predecessor */

  SIR_List<item_type> *Head(void);  // List head */

  void Remove(void);  // remove myself */
};
```

4.23.2  Purpose:

The class SIR_ListElem is a template class for a list element. It is used as a base class for all list elements in the SIR data structure.

4.23.3  Notes:

- The template class SIR_ListElem implements one element in a double-linked list.

- The private members SuccItem, PredItem and ListHead are used internally only.
• The method Succ returns the successor of this element, or NULL (no error) if not available.
• The method Pred returns the predecessor of this element, or NULL (no error) if not available.
• The method Head returns a pointer to the list head, or NULL (no error) if not available.
• The method Remove removes this element from the list. The removed element is not deleted.
4.24 Class SIR Member

The class SIR_member is declared in file IntRep/Member.h.

4.24.1 Declaration:

```cpp
class SIR_member : /* member of user-defined type */
public SIR_node, /* is a node */
public SIR_listelem <SIR_member> /* and a list element */
{
public:
  *type; /* link to type table entry */
  *symbol; /* link to symbol table entry (NULL if none) */
  unsigned int BitFieldSize; /* bitfield size (0 if none) */

  //+++++++++++++++++++++++API Layer 1+++++++++++++++++++++++/

  //SIR_member( /* constructor #1 */
  *type,
  *symbol,
  BitFieldSize = 0,
  Line = 0,
  FileInfo = NULL);

  SIR_member( /* constructor #3 (duplicator) */
  *Original);

  ~SIR_member( void); /* destructor */

  ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
  MemberFct, MemberFctArg);
};
```

4.24.2 Purpose:

An object of class SIR_member represents a member of a struct, union, or enum type definition.

4.24.3 Notes:

- The link Type points to the type of this member.
• The link Symbol points to the symbol of this member, or is NULL in case of an unnamed structure padding entry or an unused bit field.

• The member BitFieldSize determines the size of this bitfield. It is 0 if no bitfield is defined.

• The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.
4.25 Class SIR Members

The class SIR_Members is declared in file IntRep/Member.h.

4.25.1 Declaration:

```cpp
1 class SIR_Members : /* list of members */
2 public SIR_List <SIR_Member> /* is simply a list of members */
3 {
4 public:
5 //+++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++++
6 
7 SIR_Members( /* constructor #1 */
8 sir_member *FirstEntry = NULL);
9 
10 SIR_Members( /* constructor #2 (duplicator) */
11 sir_members *Original);
12 
13 ~SIR_Members( void ); /* destructor */
14 
15 
16 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
17 sir_node_nptr MemberFct,
18 sir_node_narg MemberFctArg);
19 
20 void UnAlias( void ); /* unalias all type, usertype, symbol links */
21 
22 sir_member *Find( /* searches for a specific entry */
23 sir_symbol *Symbol); /* returns NULL if not found */
24 }
```

4.25.2 Purpose:

An object of class SIR_Members represents a list head. The list contains objects of type SIR_Member.

4.25.3 Notes:

- The list of members is sorted by declaration order and can be traversed with the standard list operations.
• The method **DFS_ForAllNodes** performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, **MemberFct** is called for each node with the argument **MemberFctArg**.

• The method **Find** searches for a member in this list. **NULL** (but no error) is returned if no member is found.
4.26 Class SIR Node

The class SIR_Node is a base class that is declared in file IntRep/Node.h.

4.26.1 Declaration:

```c
1 class SIR_Node /* base class for all nodes */
2 {
3   public:
4   sir_lineinfo *LineInfo; /* line information (NULL if unknown) */
5
6   //+++++++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++++++++++/
7
8   SIR_Node(void); /* constructor #1 (nodes without lineinfo) */
9   SIR_Node(/* LineInfo */); /* constructor #2 (nodes with lineinfo) */
10  unsigned int Line;
11  sir_fileinfo *FileInfo; /* constructor #3 (dupilcator) */
12  sir_node *Original; /* destructor */
13
14  ERROR UnaliasFilePointer ( /* unalias the file info (if any) */
15      sir_node_marg /* Unused */);
16
17  ERROR MarkUsedFiles ( /* marks the file entry used here */
18      sir_node_marg /* Unused */);
19
20  ERROR Strip ( /* removes the source information #1 */
21      sir_node_marg /* Unused */; /* (iterator version) */
22
23  void Strip(void); /* removes the source information #2 */
24
25  void UpdateLineInfo( /* update the source info (if any) */
26      unsigned int Line, /* (keeps old info if no new is specified) */
27      sir_fileinfo *FileInfo); /* sets the same source info */
28
29  void SetLineInfo ( /* sets the same source info */
30      sir_node *Node);}
31```

71
4.26.2 Purpose:

The base class SIR.Node represents source code location information. It is a base class of all objects for which source line and source file information is needed.

4.26.3 Notes:

- The link SIR.LineInfo represents the source line (and the source filename) of this node.
- The method Strip removes any source information from this node.
- The method UpdateLineInfo attaches source information to this node unless any source information is already present.
- The method SetLineInfo sets the given source information to this node (overwriting any previous information).
### 4.27 Class SIR Note

The class SIR\_Note is declared in file IntRep\_Note.h.

#### 4.27.1 Declaration:

```cpp
1 class SIR\_Note : /* note (entry in a list of notes) */
2   public SIR\_List\_Elem< SIR\_Note > /* is a list element */
3 {
4   public:
5     string Name; /* name of this note */
6     sir\_constant *Content; /* contents of this note */
7     sir\_symbol *Symbol; /* link back to symbol (or NULL) */
8     sir\_usertype *User\_Type; /* link back to usertype (or NULL) */
9     sir\_label *Label; /* link back to label (or NULL) */
10    sir\_import *Imported; /* link to import file (NULL if not imported) */
11
12    //+++++++API Layer 1+++++++/
13
14
15    SIR\_Note( /* constructor #1 */
16         const char *Name, /* name of this note */
17         sir\_constant *Content, /* contents of this note */
18         sir\_symbol *Symbol = NULL, /* max. one */
19         sir\_usertype *User\_Type = NULL, /* of these */
20         sir\_label *Label = NULL); /* not NULL */
21
22    SIR\_Note( /* constructor #3 (duplicator) */
23         *Sir\_note /*Original*/;
24
25    ~SIR\_Note( void); /* destructor */
26
27    ERROR DFS\_For\_All\_Nodes( /* iterator over all nodes (depth first) */
28         *Sir\_node\_mptr Member\_Fct,
29         *Sir\_node\_marg Member\_Fct\_Arg);
30
31    ERROR DFS\_For\_All\_Notes( /* iterator over all notes (depth first) */
34         *Sir\_note\_mptr Member\_Fct,
35         *Sir\_note\_marg Member\_Fct\_Arg);
36
37    ERROR Set\_Imported( /* sets the Imported pointer (if unset) */
39         *Sir\_note\_marg Import\_Ptr; /* or resets it if NULL */
40    ERROR Take\_Over\_Import( /* takes over an imported note */
42         *Sir\_note\_marg Import\_Ptr);
43
```

73
4.27.2 Purpose:

An object of class \texttt{SIR\_Note} represents an annotation attached to the design, to a symbol, to a user-defined type or to a label.

4.27.3 Notes:

- The member \texttt{Name} contains the name of this note.

- The member \texttt{Content} contains the constant value of this note.

- The link \texttt{Symbol} points to the symbol where this note is attached to, or \texttt{NULL} otherwise.

- The link \texttt{UserType} points to the user-defined type where this note is attached to, or \texttt{NULL} otherwise.

- The link \texttt{Label} points to the label where this note is attached to, or \texttt{NULL} otherwise.

- The link \texttt{Imported} points to the design name from which this annotation was imported, or is \texttt{NULL} otherwise.

- The method \texttt{DFS\_ForAllNodes} performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, \texttt{MemberFct} is called for each node with the argument \texttt{MemberFctArg}.

- The method \texttt{DFS\_ForAllNotes} performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, \texttt{MemberFct} is called for each annotation with the argument \texttt{MemberFctArg}.
• The method Print generates SpecC (or C++) source code for this annotation.
• The method WriteSC generates SpecC (or C++) source code for this annotation.
• The method Touch removes any source information for this annotation.
4.28 Class SIR Notes

The class SIR_Notes is declared in file IntRep/Note.h.

4.28.1 Declaration:

```c
1 class SIR_Notes : /* notes */
2 public SIR_List<SIR_Note> /* is simply a list of notes */
3 { /* with additional member(function)s */
4 public:
5
6 //++++++++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++
7
8 SIR_Notes( /* constructor #1 */
9 sir_note *FirstEntry = NULL);
10
11 SIR_Notes( /* constructor #2 (duplicator) */
12 sir_notes *Original);
13
14 ~SIR_Notes(void); /* destructor */
15
16 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
17 sir_node_nptr MemberFct,
18 sir_node_marg MemberFctArg);
19
20 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
21 sir_note_nptr MemberFct,
22 sir_note_marg MemberFctArg);
23
24 sir_note *Find( /* find an entry */
25 const char *Name); /* (returns NULL if not found) */
26
27 sir_note *Insert ( /* inserts a new note */
28 sir_note *NewNote); /* (may return NULL and SIR_Error) */
29
30 static ERROR Merge( /* merges two note-tables */
31 sir_notes **Notes1, /* (merges 2 into 1) */
32 sir_notes **Notes2); /* (supports automatic unaliasing) */
33
34 const char *Print( /* print ASCII definitions */
35 BOOL DoIt,
36 BOOL CplusplusMode = FALSE);
37
38 ERROR WriteSC( /* (re-) generates SpecC source code */
39 FILE *File );
40
41
77
```
4.28.2 Purpose:

An object of class SIRNotes represents a list head. The list contains objects of type SIRNote.

4.28.3 Notes:

- The list of annotations is sorted in alphabetical order and can be traversed with the standard list operations.

- The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.

- The method DFS_ForAllNotes performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, MemberFct is called for each annotation with the argument MemberFctArg.

- The method Find searches for an annotation in this list by Name. NULL (but no error) is returned if no annotation is found.

- The method Insert inserts the given annotation into this list.

- The method Print generates SpecC (or C++) source code for all the annotations in this list.

- The method WriteSC generates SpecC (or C++) source code for all the annotations in this list.
4.29 Class SIR Parameter

The class SIR_Parameter is declared in file IntRep/Parameter.h.

4.29.1 Declaration:

```cpp
class SIR_Parameter : /* function parameter (or beh./chan. port) */
    public SIR_ListElem<SIR_Parameter> /* is a list element */
{ public:
  5 sir_type *Type; /* link to type table entry */
  6 sir_symbol *Symbol; /* link to symbol table entry (or NULL) */

  // API Layer

  12 SIR_Parameter( /* constructor #1 (general) */
      sir_type *Type,
      sir_symbol *Symbol = NULL);
  15 SIR_Parameter( /* constructor #3 (duplicator) */
      sir_parameter *Original);
  19 SIR_Parameter( void); /* destructor */
};
```

4.29.2 Purpose:

An object of class SIR_Parameter represents a parameter (or argument) of a function, or a port of a behavior or channel.

4.29.3 Notes:

- The link Type points to the type of this parameter.
- The link Symbol points to the symbol defined with this parameter. This link may be NULL in case the function parameter or class port is unused.
4.30 Class SIR Parameters

The class SIR_Parameters is declared in file IntRep/Parameter.h.

4.30.1 Declaration:

```cpp
1 class SIR_Parameters : public SIR_List<SIR_Parameter>
2 { /* is simply a list of parameters */
3     /* with additional member(function)s */
4 public:
5 //++++++++++++++++++++++++++++++++++++++++++++++++++API Layer+++++++++++++++++++++++++++++++++++++
6
7 9 SIR_Parameters();
8     /* constructor #1 */
9     sir_parameter *FirstEntry = NULL;
10
11 12 SIR_Parameters();
12     /* constructor #2 (duplicator) */
13     sir_parameters *Original;
14
15 16 SIR_Parameters();
16     /* constructor #3 (from type parameters) */
17     sir_type_ptrs *TypeParams;
18
19 18 SIR_Parameters(void); /* destructor */
20
21 21 BOOL IsVoid(void); /* determines if there are no parameters */
22
23 24 void UnAlias(void); /* unalias all type, usertype, symbol links */
25
26 27 sir_parameter *Find(); /* find corresponding parameter */
27     sir_type_ptrs *TypeParams;
28     /* (returns NULL if not found) */
29     sir_type_ptr *TypePtr;
30 }
```

4.30.2 Purpose:

An object of class SIR_Parameters represents a list head. The list contains objects of type SIR_Parameter.

4.30.3 Notes:

- The list of parameters is sorted by declaration order and can be traversed with the standard list operations.
- The method IsVoid checks if the parameter list is void.
4.31 Class SIR PortMap

The class SIR_PortMap is declared in file IntRep/PortMap.h.

4.31.1 Declaration:

```c
class SIR_PortMap : /* port mapping (for one port) */
public SIR_ListElem<SIR_PortMap> /* is a list element */
{
  public:
  sir_bitslices *BitSlices; /* list of bitslices */

  //++++++++++++++++++API Layer 1++++++++++++++++++

  SIR_PortMap( /* constructor #1 (general) */
  sir_bitslices *BitSlices);

  SIR_PortMap( /* constructor #3 (dupliicator) */
  sir_portmap *Original);

  ~SIR_PortMap(void); /* destructor */

  static sir_portmap *New( /* create a new direct port mapping (#2b) */
  sir_symbol *DirectMapping); /* (returns NULL if SIR_Error) */

  ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
  sir_node_mptr MemberFct,
  sir_node_marg MemberFctArg);

  void UnAlias(void); /* unalias all type, usertype, symbol links */

  void WriteCC_Decl( /* generates C++ source code (part 1) */
  FILE *File,
  sir_type *PortType,
  int *BitBusNum,
  int *OpenPortNum,
  int *ConstPortNum);

  const char *WriteCC_Init( /* generates C++ source code (part 2) */
  sir_type *PortType, /* (returns NULL for "nothing to do") */
  int *BitBusNum,
  int *OpenPortNum,
  int *ConstPortNum);
```
void WriteCC( /* generates C++ source code (part 3) */
  FILE *File,
  sir_type *PortType,
  int *BitBusNum,
  int *OpenPortNum,
  int *ConstPortNum);

bool IsOpenMapping(void); /* determines whether mapping is open */

bool IsDirectlyMappable( /* determines port mapping type */
  sir_type *PortType);

ERROR Check( /* semantically check the port mapping */
  sir_type_ptr *Port,
  unsigned int PortNum,
  const char *SourceFile = NULL, /* default: no warnings */
  unsigned int SourceLine = 0);

4.31.2 Purpose:

An object of class SIR_ProtMap represents a mapping of a port of an instance onto a variable, a channel or a port, or onto a set of concatenated slices of bitvectors.

4.31.3 Notes:

- The member BitSlices contains the list of bitslices of this port mapping.
- The method New creates a new port mapping.
- The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.
- The method IsOpenMapping checks whether this represents an open mapping.
- The method Check performs a semantic checking of this mapping and returns any error.
4.32 Class SIR PortMaps

The class SIR_PortMaps is declared in file IntRep/PortMap.h.

4.32.1 Declaration:

```c
1 class SIR_PortMaps : public SIR_List<SIR_PortMap> { /* list of port mappings */
2     public SIR_List<SIR_PortMap> /* is simply a list of portmaps */
3     { /* with additional methods */
4     public:
5     } //++++++++++++++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++++++++++++++/
6
7 SIR_PortMaps( /* constructor #1 */
8     sir_portmap *FirstEntry = NULL);
9
10 SIR_PortMaps( /* constructor #2 (duplicator) */
11     sir_portmaps *Original);
12
13 ~SIR_PortMaps( void ); /* destructor */
14
15 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
16     sir_node_mptr MemberFct,
17     sir_node_marg MemberFctArg);
18
19 void UnAlias( void ); /* unalias all type, usertype, symbol links */
20
21 ERROR WriteSC( /* (re-) generates SpecC source code */
22     FILE *File );
23
24 ERROR Check( /* semantically check the port mappings */
25     sir_type_ptrs *PortList,
26     const char *SourceFile = NULL, /* default : no warnings */
27     unsigned int SourceLine = 0);
28 }
```

4.32.2 Purpose:

An object of class SIR_PortMaps represents a list head. The list contains objects of type SIR_PortMap.
4.32.3 Notes:

- The list of port maps is sorted by declaration order and can be traversed with the standard list operations.

- The length of this list of port maps must be equal to the number of ports of the instantiated behavior or channel.

- The method `DFS_ForAllNodes` performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, `MemberFct` is called for each node with the argument `MemberFctArg`.

- The method `WriteSC` generates SpecC source code for the mappings.

- The method `Check` performs a semantic checking of all these mappings and returns the first error.
4.33 Class SIR Statement

The class SIR_Statement is declared in file `IntRep/Statement.h`.

4.33.1 Declaration:

```cpp
1 class SIR_Statement : /* statement */
2     public SIR_Node, /* is a node */
3     public SIR_ListElem<SIR_Statement> /* and a list element */
4 {
5     public:
6         SIR_STMTNT_TYPE StmntType; /* statement type */
7         SIR_symbol *Function; /* link back to the function this is part of */
8         SIR_expression *Expression1; /* first expression (or NULL) */
9         SIR_expression *Expression2; /* second expression (or NULL) */
10        SIR_expression *Expression3; /* third expression (or NULL) */
11        SIR_constant *Constant; /* constant (or NULL) */
12        SIR_statement *Statement1; /* first sub-statement (NULL if none) */
13        SIR_statement *Statement2; /* second sub-statement (NULL if none) */
14        SIR_label *Label; /* pointer to label (NULL if unlabeled) */
15        SIR_symbols *Scope; /* local scope (or NULL) */
16        SIR_statements *Statements; /* list of statements (or NULL) */
17        SIR_symbol_ptrs *Events; /* list of events (or NULL) */
18        SIR_exceptions *Exceptions; /* exceptions (or NULL) */
19        SIR_constraints *Constraints; /* constraints (or NULL) */
20        SIR_transitions *Transitions; /* transitions (or NULL) */
21
22 //++++++++++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++++++++++/
23
24 SIR_Statement( /* constructor #1 */
25     SIR_STMTNT_TYPE StmntType, /* (for general statements) */
26     SIR_symbol *Function,
27     SIR_expression *Expression1 = NULL,
28     SIR_expression *Expression2 = NULL,
29     SIR_expression *Expression3 = NULL,
30     Line = 0,
31     SIR_fileinfo *FileInfo = NULL,
32     SIR_statement *Statement1 = NULL,
33     SIR_statement *Statement2 = NULL,
34     unsigned int Line = 0,
35     SIR_statements *Statements,
36     SIR_symbols *Scope,
37 SIR_Statement( /* constructor #2 */
38     SIR_STMTNT_TYPE StmntType, /* (for compound statements) */
39     SIR_symbol *Function,
40     SIR_symbols *Scope,
41     SIR_statements *Statements,
42     Line = 0,
43     SIR_fileinfo *FileInfo = NULL);```
SIR_Statement() /* constructor #3 */
SIR_STMTNT_TYPE StmntType, /* (for case statements) */
sir_symbol *Function,
sir_constant *Constant,
sir_statement *Statement1 = NULL,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Statement() /* constructor #4 */
SIR_STMTNT_TYPE StmntType, /* (for statements with labels) */
sir_symbol *Function,
sir_label *Label,
sir_statement *Statement1 = NULL,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Statement() /* constructor #5 */
SIR_STMTNT_TYPE StmntType, /* (for statements with events) */
sir_symbol *Function,
sir_symbol_ptrs *Events,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Statement() /* constructor #6 */
SIR_STMTNT_TYPE StmntType, /* (for exception statements) */
sir_symbol *Function,
sir_statement *Statement1,
sir_exceptions *Exceptions,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Statement() /* constructor #7 */
SIR_STMTNT_TYPE StmntType, /* (for timing statement) */
sir_symbol *Function,
sir_statement *Statement1,
sir_constraints *Constraints,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Statement() /* constructor #8 */
SIR_STMTNT_TYPE StmntType, /* (for fsm statement) */
sir_symbol *Function,
sir_transitions *Transitions,
unsigned int Line = 0,
sir_fileinfo *FileInfo = NULL);

SIR_Statement() /* constructor #10 (duplicator) */
sir_statement *Original);
SIR_Statement(void); /* destructor */
96
97 BOOL IsCleanBehaviorCall (void); /* checks for "clean" behavior call */
98 /* (e.g. "b1.main();") */
99
100 BOOL IsCleanListOfBehaviorCalls ( /* checks for "clean" comp. behavior */
101     BOOL OneCallMax = FALSE, /* (e.g. "{ b1.main(); b2.main(); }") */
102     BOOL OneCallMin = TRUE);
103
104 ERROR CheckCompoundBehavior ( /* checks for "clean" comp. behavior */
105     BOOL OneCallMax = FALSE, /* (e.g. "{ b1.main(); b2.main(); }") */
106     BOOL OneCallMin = TRUE); /* (returns error if not clean) */
107
108 BOOL IsDirectDependant ( /* checks if stmt. depends on symbol */
109     sir_symbol *ThatSymbol); /* (directly) */
110
111 BOOL IsIndirectDependant ( /* checks if stmt. depends on symbol */
112     sir_symbol *ThatSymbol, /* (indirectly) */
113     sir_expression **DepExpr = NULL);
114
115 sir_statements *GetList (void); /* determines the list of this statement */
116 /* (returns NULL if not in a list) */
117
118
119 ERROR DFS_ForAllNodes ( /* iterator over all nodes (depth first) */
120     sir_node_mptr MemberFct,
121     sir_node_marg MemberFctArg);
122
123 ERROR DFS_ForAllSymbols ( /* iterator over all symbols (depth first) */
124     sir_symbol_mptr MemberFct,
125     sir_symbol_marg MemberFctArg);
126
127 ERROR DFS_ForAllUserTypes ( /* iterator over all usertypes (depth first) */
128     sir_usertype_mptr MemberFct,
129     sir_usertype_marg MemberFctArg);
130
131 ERROR DFS_ForAllNotes ( /* iterator over all notes (depth first) */
132     sir_node_mptr MemberFct,
133     sir_node_marg MemberFctArg);
134
135 ERROR DFS_ForAllStatements ( /* iterator over all statements (depth first) */
136     sir_stmt_mptr MemberFct,
137     sir_stmt_marg MemberFctArg);
138
139 ERROR DFS_ForAllExpressions ( /* iterator over all expressions (dfs) */
140     sir_expr_mptr MemberFct,
141     sir_expr_marg MemberFctArg);
142
143 BOOL DFS_FindDependant ( /* searches for dependants (depth first) */
144     sir_symbol *ThatSymbol,
145
89
4.33.2 Purpose:

An object of class SIRStatement represents a SpecC statement.

4.33.3 Notes:

- The member StmtType determines the type of this statement. It is one of the following values:
  SIR_STMT_LABELED, SIR_STMT_COMPOUND, SIR_STMT_EXPRESSION, SIR_STMT_IF_ELSE, SIR_STMT_SWITCH,
SIR_STMT_CASE, SIR_STMT_DEFAULT, SIR_STMT WHILE, SIR_STMT DO WHILE, SIR_STMT FOR, SIR_STMT GOTO, SIR_STMT CONTINUE, SIR_STMT BREAK, SIR_STMT RETURN, SIR_STMT PAR, SIR_STMT_PIPE, SIR_STMT_EXCEPTION, SIR_STMT_TIMING, SIR_STMT_FSM, SIR_STMT_WAIT, SIR_STMT_WAITFOR, SIR_STMT_NOTIFY, SIR_STMT_NOTIFYONE.

Depending on StmtType, only some of the other members are valid. Invalid members are set to NULL.

- The link Function points to the function symbol which contains this statement.
- The member Expression1 contains the first expression in this statement, or NULL if not applicable.
- The member Expression2 contains the second expression in this statement, or NULL if not applicable.
- The member Expression3 contains the third expression in this statement, or NULL if not applicable.
- The member Constant contains the constant for this statement, or NULL if not applicable.
- The member Statement1 contains the first sub-statement for this statement, or NULL if not applicable.
- The member Statement2 contains the second sub-statement for this statement, or NULL if not applicable.
- The member Label points to the label used with this statement, or NULL if not applicable.
- The member Scope contains the local symbol table for this statement, or NULL if not applicable.
- The member Statements contains the list of sub-statements for this statement, or NULL if not applicable.
- The member Events contains the list of events used with this statement, or NULL if not applicable.
- The member Exceptions contains the list of exceptions for SIR_STMT_EXCEPTION, or NULL otherwise.
- The member Constraints contains the list of timing constraints for SIR_STMT_TIMING, or NULL otherwise.
- The member Transitions contains the list of transitions for SIR_STMT_FSM, or NULL otherwise.
- The method GetList returns a handle to the list of statements.
- The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.
• The method DFS_ForAllSymbols performs a traversal over all symbols in this subtree in depth-first search order. During the traversal, MemberFct is called for each symbol with the argument MemberFctArg.

• The method DFS_ForAllUserTypes performs a traversal over all user-defined types in this subtree in depth-first search order. During the traversal, MemberFct is called for each user-defined type with the argument MemberFctArg.

• The method DFS_ForAllNotes performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, MemberFct is called for each annotation with the argument MemberFctArg.

• The method DFS_ForAllStatements performs a traversal over all statements in this subtree in depth-first search order. During the traversal, MemberFct is called for each statement with the argument MemberFctArg.

• The method DFS_ForAllExpressions performs a traversal over all expressions in this subtree in depth-first search order. During the traversal, MemberFct is called for each expression with the argument MemberFctArg.

• The method WriteSC generates SpecC or C++ source code for this statement.
4.34 Class SIR Statements

The class SIRStatements is declared in file IntRep/Statement.h.

4.34.1 Declaration:

```cpp
1 class SIR_Statements :  // sequential statements */
2     public SIR_List<SIR_Statement>  // is simply a list of statements */
3                     {                // with additional member(function)s */
4     public:
5         sir_symbols *CmpndScope;    // link to compound statement scope */
6
7  //////////////////////////////////////////////////////API Layer 1/////////////////////////////////////////
8
9 11 SIR_Statements (  // constructor #1 */
12         sir_symbols *CmpndScope = NULL,
13         sir_statement *FirstEntry = NULL);
14
15 SIR_Statements (  // constructor #3 (duplicator) */
16         sir_statements *Original);
17
18 ~SIR_Statements (void);  // destructor */
19
20 21 ERROR DFS_ForAllNodes (  // iterator over all nodes (depth first) */
22         sir_node_mptr MemberFct,
23         sir_node_marg MemberFctArg);
24
25 ERROR DFS_ForAllSymbols ( // iterator over all symbols (depth first) */
26         sir_symbol_mptr MemberFct,
27         sir_symbol_marg MemberFctArg);
28
29 ERROR DFS_ForAllUserTypes ( // iterator over all user types (depth first) */
30         sir_usertype_mptr MemberFct,
31         sir_usertype_marg MemberFctArg);
32
33 ERROR DFS_ForAllNotes (    // iterator over all notes (depth first) */
34         sir_note_mptr MemberFct,
35         sir_note_marg MemberFctArg);
36
37 ERROR DFS_ForAllStatements ( // iterator over all statements (depth first) */
38         sir_stmt_mptr MemberFct,
39         sir_stmt_marg MemberFctArg);
40
41 ERROR DFS_ForAllExpressions( // iterator over all expressions (dfs) */
42         sir_expr_mptr MemberFct,
43         sir_expr_marg MemberFctArg);
```
44 BOOL DFS_FindDependant(     /* searches for dependants (depth first)*/
45     sir_symbol   *ThatSymbol,
46     sir_statement **DepStmt,
47     sir_expression **DepExpr,
48     SIR_DEPENDENCY *Reason);
49
50 void SetAlias(     /* sets all type, usertype, symbol alias */
51     sir_statement *Alias);
52     /* (iterates over symbols and usertypes)*/
53
54 sir_statement *FindLabeledStmt(/* find a labeled statement in this list */
55     sir_label      *Label);     /* (returns NULL if not found) */
56 );

4.34.2 Purpose:

An object of class SIR_Statements represents a list head. The list contains objects of type SIR_Statement.

4.34.3 Notes:

- The list of statements is sorted in definition order and can be traversed with the standard list operations.

- A statement list belongs to a compound statement.

- The link CmpndScope points to the local scope of the compound statement.

- The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.

- The method DFS_ForAllSymbols performs a traversal over all symbols in this subtree in depth-first search order. During the traversal, MemberFct is called for each symbol with the argument MemberFctArg.

- The method DFS_ForAllUserTypes performs a traversal over all user-defined types in this subtree in depth-first search order. During the traversal, MemberFct is called for each user-defined type with the argument MemberFctArg.

- The method DFS_ForAllNotes performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, MemberFct is called for each annotation with the argument MemberFctArg.

94
• The method DFS_ForAllStatements performs a traversal over all statements in this subtree in depth-first search order. During the traversal, MemberFct is called for each statement with the argument MemberFctArg.

• The method DFS_ForAllExpressions performs a traversal over all expressions in this subtree in depth-first search order. During the traversal, MemberFct is called for each expression with the argument MemberFctArg.

• The method WriteSC generates SpecC or C++ source code for this statement.

• The method FindLabeledStmt searches for a labeled statement in this list. NULL is returned if it is not found.
4.35 Class SIR Symbol

The class SIR_Symbol is declared in file IntRep/Symbol.h.

4.35.1 Declaration:

```
1 class SIR_Symbol : /* symbol (entry in a symbol table) */
2     public SIR_Node, /* is a node */
3     public SIR_ListElem<SIR_Symbol> /* and a list element */
4 {
5 public:
6     SIR_SYMBOLCLASS Class; /* class of this symbol (see above) */
7     string Name; /* name of this symbol */
8     SIR_STORAGE StorageClass; /* storage class (see above) */
9     int PipeStages; /* number of pipeline stages (if PIPED) */
10    sir_type *Type; /* link to type table entry */
11    sir_notes *Notes; /* notes for this symbol (NULL if none) */
12    sir_initializer *Initializer; /* initialization info (or NULL) */
13    int EnumValue; /* value (if this is an enum-constant) */
14    sir_parameters *Parameters; /* parameter or port list (or NULL) */
15    sir_symbols *ParamScope; /* parameter scope (or NULL) */
16    sir_symbols *ClassScope; /* class scope (or NULL) */
17    sir_labels *Labels; /* label list (for functions) */
18    sir_statement *FctBody; /* function body (or NULL) */
19    sir_symbol_ptrs *Interfaces; /* list of implemented interfaces (or NULL) */
20    sir_portmaps *PortMappings; /* port mapping list for insts. (or NULL) */
21    sir_import *Imported; /* link to import file (NULL if not imported) */
22
23   sir_symbol *Alias; /* alias pointer (temporary only) */
24
25
26 //++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++
27
28 SIR_Symbol( /* constructor #1 (general) */
29     SIR_SYMBOLCLASS Class,
30     const char *Name,
31     sir_type *Type = NULL,
32     SIR_STORAGE StorageClass = SIR_STORAGE_NONE,
33     int PipeStages = 0,
34     sir_initializer *Initializer = NULL,
35     unsigned int Line = 0,
36     sir_fileinfo *FileInfo = NULL);
37
38 SIR_Symbol( /* constructor #2 (functions) */
39     const char *Name,
40     sir_type *Type,
41     SIR_STORAGE StorageClass,
42     sir_parameters *Parameters,
43 ```

97
sir_symbols  *ParamScope,
sir_labels  *Labels,
sir_statement  *FctBody,
unsigned int  Line = 0,
sir_fileinfo  *FileInfo = NULL);

SIR Symbol(  /* constructor #3 (classes) */
    SIR_SYMBOLCLASS  Class,
    const char  *Name,
sir_type  *Type,
sir_parameters  *Parameters,
sir_symbols  *ClassScope,
sir_symbol_ptrs  *Interfaces,
    unsigned int  Line = 0,
sir_fileinfo  *FileInfo = NULL);

SIR Symbol(  /* constructor #4 (instantiations) */
    const char  *Name,
sir_type  *Type,
sir_portmaps  *PortMappings,
    unsigned int  Line = 0,
sir_fileinfo  *FileInfo = NULL);

SIR Symbol(  /* constructor #6 (special for parser) */
    sir_declspec  *DeclSpec,
    sir_declarator  *Declarator,
sir_type  *Type,
    BOOL  StoreParameters = false);

SIR Symbol(  /* constructor #7 (special for parser) */
    SIR_SYMBOLCLASS  Class,
    sir_declarator  *Declarator,
sir_type  *Type,
sir_symbol_ptrs  *Interfaces,
    BOOL  StoreParameters = false);

SIR Symbol(  /* constructor #8 (duplicator) */
    sir_symbol  *Original);

~SIR Symbol(void);  /* destructor */

BOOL IsVariable(void);  /* check if IDENTIFIER and !TYPE_FUNCTION */
/* and !TYPE_BEHAVIOR and !TYPE_CHANNEL and */
/* !TYPE_INTERFACE and Direction != PORT_NONE */
/* and !IsEnumMember */

BOOL IsVariableDefinition(void);  /* check if IsVariable and !STORAGE_EXTERN */

BOOL IsFunction(void);  /* check if IDENTIFIER and TYPE_FUNCTION */
95 BOOL IsFunctionDefinition (void); /* check if IsFunction and FctBody!=NULL */
96 97 BOOL IsBehaviorInstance (void); /* check if IDENTIFIER and TYPE_BEHAVIOR */
98 99 BOOL IsChannelInstance (void); /* check if IDENTIFIER and TYPE_CHANNEL */
100 101 BOOL IsTypeDef (void); /* check if TYPEDEF */
102 103 BOOL IsBehavior (void); /* check if BEHAVIOR */
104 105 BOOL IsBehaviorDefinition (void); /* check if IsBehavior and ClassScope!=NULL */
106 107 BOOL IsChannel (void); /* check if CHANNEL */
108 109 BOOL IsChannelDefinition (void); /* check if IsChannel and ClassScope!=NULL */
110 111 BOOL IsInterface (void); /* check if INTERFACE */
112 113 BOOL IsInterfaceDefinition (void); /* check if IsInt. and ClassScope!=NULL */
114 115 BOOL IsClass (void); /* check if BEHAVIOR, CHANNEL or INTERFACE */
116 117 BOOL IsPort (void); /* check if IDENTIFIER and TYPE_INTERFACE */
118 /* or Direction != PORT_NONE */
119 120 BOOL IsParameter (void); /* check if defined in SCOPE_PARAMETER */
121 122 BOOL IsEnumMember (void); /* check if this is an enum member */
123 124 125 BOOL IsDirectDependant (                        /* checks if this symbol depends on that */
126     sir_symbol *ThatSymbol, /* (directly) */
127     SIR_DEPENDENCY *Reason = NULL);
128 129 BOOL IsIndirectDependant (                      /* checks if this symbol depends on that */
130     sir_symbol *ThatSymbol, /* (indirectly) */
131     SIR_DEPENDENCY *Reason = NULL,
132     sir_statement **DepStmt = NULL,
133     sir_expression **DepExpr = NULL);
134 135 BOOL MapsPortTo (      /* checks for port-instantiation */
136     sir_symbol *Symbol); /* (instantiations only) */
137 138 BOOL Instantiates (   /* checks for instantiation */
139     sir_symbol *ClassSymbol); /* (behaviors and channels only) */
140 141 BOOL Contains (       /* checks for containment */
142     sir_symbol *ClassSymbol); /* (behaviors and channels only) */
143 144 sir_symbols *GetScope (void); /* determines the scope of this symbol */

99
/* (returns NULL if not in a table) */

sir_design * GetDesign (void);  /* determines the design this symbol is in */
/* (returns NULL if type is not in design) */

ERROR DFS_ForAllNodes (  /* iterator over all nodes (depth first) */
sir_node_mptr MemberFct,
sir_node_marg MemberFctArg);

ERROR DFS_ForAllSymbols ( /* iterator over all symbols (depth first) */
sir_symbol_mptr MemberFct,
sir_symbol_marg MemberFctArg);

ERROR DFS_ForAllUserTypes ( /* iterator over all usertypes (depth f.) */
sir_usertp_mptr MemberFct,
sir_usertp_marg MemberFctArg);

ERROR DFS_ForAllNotes ( /* iterator over all notes (depth first) */
sir_note_mptr MemberFct,
sir_note_marg MemberFctArg);

ERROR SetImported ( /* sets the Imported pointer (if unset) */
sir_symbol_marg ImportPtr);  /* or resets it if NULL */

ERROR TakeOverImport ( /* takes over an imported symbol */
sir_symbol_marg ImportPtr);

ERROR MarkImportEntry ( /* marks its import entry being used */
sir_symbol_marg /* Unused */);

ERROR MarkUsedTypes ( /* marks the type entries used here */
sir_symbol_marg /* Unused */);

BOOL DFS_FindDependant ( /* searches for dependants (depth first) */
sir_symbol **SkipSymbol,  /* (internal, use the next ones!) */
sir_symbol *ThatSymbol,
sir_symbol **DepSymbol,
sir_statement **DepStmtnt,
sir_expression **DepExpr,
SIR_DEPENDENCY *Reason);

BOOL FindDependant ( /* finds symbols that depend on this one #1 */
sir_symbol **DepSymbol = NULL,  /* return symbol found */
sir_statement **DepStmtnt = NULL,  /* return statement found */
sir_expression **DepExpr = NULL,  /* return expression found */
SIR_DEPENDENCY *Reason = NULL,  /* return type of dependency */
sir_symbols *SearchScope = NULL,  /* search in this scope */
sir_symbol *SearchAfter = NULL);  /* continue search here */
BOOL FindDependant ( /* finds symbols that depend on this one */
  sir_statement *SearchStmt, /* search in this statement */
  sir_statement **DepStmt = NULL, /* return statement found */
  sir_expression **DepExpr = NULL, /* return expression found */
  SIR_DEPENDENCY *Reason = NULL); /* return type of dependency */

void SetAlias ( /* sets all type, usertype, symbol alias */
  sir_symbol *Alias); /* (iterates over symbols and usertypes) */

void UnAlias ( /* unalias all type, usertype, symbol inks */
  sir_symbols *GlobalSymbols);

const char *PrintStorageClass ( /* prints SpecC storage class */
  FILE *File, /* for non-class symbols */
  BOOL AppendSpace = false); /* "extern" or "extern\n"

static const char *PrintStorageClass ( /* prints SpecC storage class */
  SIR_STORAGE StorageClass, /* (e.g. "extern" or "extern\n"
  int PipeStages = 0;
  BOOL AppendSpace = false);

ERROR WriteSC ( /* (re-) generates SpecC source code */
  FILE *File, /* for non-class symbols */
  BOOL AsDeclaration = false,
  BOOL CplusMode = false,
  BOOL AsReference = false);

ERROR WriteSC2 ( /* (re-) generates SpecC source code */
  FILE *File, /* for class symbols */
  BOOL AsDeclaration = false,
  BOOL CplusMode = false);

ERROR WriteSC3 ( /* (re-) generates SpecC source code */
  FILE *File, /* for function definitions */
  BOOL AsDeclaration = false,
  BOOL CplusMode = false);

ERROR WriteCC2 ( /* generates C++ source code */
  FILE *File, /* for class symbols */
  BOOL AsDeclaration = false);

ERROR WriteCC2b ( /* generates C++ source code */
  FILE *File, /* for classes, method implementations */
  BOOL AsDeclaration = false);

ERROR WriteCC2IP ( /* generates C++ source code */
  BOOL AsDeclaration = false);
FILE
*File); /* (for IP classes) */

ERROR WriteC4ClassBody(  /* generates SpecC code for class bodies */  
    FILE *File, /* (for class definitions only) */
    BOOL WriteNotes = TRUE,
    BOOL WriteLines = TRUE,
    const char *ThisFilename = NULL, /* default location if */
    unsigned int ThisLineNumber = 1); /* line info unknown */

ERROR CheckInitializer(  /* checks whether initializer is legal */
    sir_initializer *Initializer = NULL, /* (if not specified, */
    sir_type *Type = NULL, /* own data is used) */
    SIR_STORAGE *StorageClass = NULL,
    int *PipeStages = NULL);

ERROR CheckStorageClass(  /* checks whether storage class is legal */
    SIR_STORAGE *StorageClass = NULL, /* (if not specified, */
    int *PipeStages = NULL, /* own data is used) */
    SIR_SYMBOLCLASS *Class = NULL,
    SIR_SCOPE *ScopeInfo = NULL);

ERROR Annotate(  /* attach a note to the symbol */
    sir_note *NewNote);  /* consumes NewNote */

void Touch(  /* modification update (remove import info) */
    BOOL ImportOnly = FALSE); /* (dflt : remove source info too) */

void Strip (  /* remove source location for the tree below */
    BOOL LocalOnly = FALSE); /* (dflt : work recursively) */
);

4.35.2 Purpose:

An object of class SIR.Symbol represents a SpecC symbol definition or declaration (such as a variable, a function, a behavior, etc.).

4.35.3 Notes:

- The member Class determines the class of this symbol. It is one of the following values: SIR_SYMBOL_IDENTIFIER, SIR_SYMBOL_TYPEDEF, SIR_SYMBOL_BEHAVIOR, SIR_SYMBOL_CHANNEL, SIR_SYMBOL_INTERFACE.

- The member Name contains the name of this symbol.
• The member StorageClass contains the storage class of this symbol. It is one of the following values:
  SIR_STORAGE_NONE, SIR_STORAGE_TYPEDEF, SIR_STORAGE_EXTERN, SIR_STORAGE_STATIC, SIR_STORAGE_AUTO,
  SIR_STORAGE_REGISTER, SIR_STORAGE_PIPE.

• The member PipeStages denotes the number of pipeline stages for SIR_STORAGE_PIPE. It is 0 otherwise.

• The link Type points to the type of this symbol.

• The member Notes contains the annotations attached to this symbol, or is NULL if there are none.

• The member Initializer contains the initial value of this symbol. This is NULL if not applicable.

• The member EnumValue contains the value of an enumerator variable. It is 0 otherwise.

• The member Parameters contains the list of parameters for function symbols, or the list of ports for
  behavior and channel symbols. It is NULL if not applicable or the symbol is just a declaration (not a
  definition).

• The member ParamScope contains the local symbol table for function parameters. It is NULL otherwise.

• The member ClassScope contains the local symbol table for classes (behaviors, channels, interfaces).
  It is NULL otherwise.

• The member Labels contains the list of defined labels for a function symbol. It is NULL otherwise.

• The member FctBody contains the top-level compound statement for a function symbol. It is NULL
  otherwise.

• The member Interfaces contains the list of implemented interfaces for behaviors and channels. It is
  NULL otherwise.

• The member PortMappings contains the port maps for instantiations of behaviors or channels. It is
  NULL otherwise.

• The link Imported points to the design name from which this symbol was imported, or is NULL
  otherwise.

• The link Alias is used internally only.

• The method IsVariable determines if this symbol is a variable.

• The method IsFunction determines if this symbol is a function or method.
• The method IsFunctionDefinition determines if this symbol is a function or method definition.
• The method IsBehaviorInstance determines if this symbol is a behavior instance.
• The method IsChannelInstance determines if this symbol is a channel instance.
• The method IsTypeDef determines if this symbol is a type definition.
• The method IsBehavior determines if this symbol is a behavior.
• The method IsBehaviorDefinition determines if this symbol is a behavior definition.
• The method IsChannel determines if this symbol is a channel.
• The method IsChannelDefinition determines if this symbol is a channel definition.
• The method IsInterface determines if this symbol is a interface.
• The method IsInterfaceDefinition determines if this symbol is a interface definition.
• The method IsClass determines if this symbol is a class (a behavior, channel or interface).
• The method IsPort determines if this symbol is a port.
• The method IsParameter determines if this symbol is a parameter of a function or method.
• The method IsEnumMember determines if this symbol is an enumerator constant.
• The method GetScope returns a pointer to the symbol table this symbol is in.
• The method GetDesign returns a pointer to the design this symbol is in.
• The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.
• The method DFS_ForAllSymbols performs a traversal over all symbols in this subtree in depth-first search order. During the traversal, MemberFct is called for each symbol with the argument MemberFctArg.
• The method DFS_ForAllUserTypes performs a traversal over all user-defined types in this subtree in depth-first search order. During the traversal, MemberFct is called for each user-defined type with the argument MemberFctArg.
• The method `DFS_ForAllNotes` performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, `MemberFct` is called for each annotation with the argument `MemberFctArg`.

• The method `PrintStorageClass` prints the storage class of this symbol.

• The method `WriteSC` generates SpecC (or C++) source code for this symbol. `WriteSC2` must be used for class symbols.

• The method `Annotate` annotates the label with `NewNote`.
4.36 Class SIR Symbols

The class SIR_Symbols is declared in file IntRep/Symbol.h.

4.36.1 Declaration:

```cpp
class SIR_Symbols : public SIR_List<SIR_Symbol> {
public:
    // symbol "table" (alias: Scope) */
    public:
    class SIR_Symbols {
        // constructor #1 */
        public:
        SIR_Symbols( *Parent, /* (creates UserTypes automatically) */
                     SIR_Scope *ScopeInfo,
                     *ParentSymbol = NULL,
                     *ParentUType = NULL,
                     *ParentStmtnt = NULL);
        // destructor */
        public:
        SIR_Symbols( void );

        // checks if the symbol can see this scope */
        public:
        BOOL IsAncestorOf( *Symbol);

        // iterator over all nodes (depth first) */
        public:
        typedef void MemberFct;

        // iterator over all symbols (depth first) */
        public:
        typedef void MemberFct;

        // iterator over all usertypes (depth first) */
        public:
        typedef void MemberFct;

        107
```
ERROR DFS_ForAllNotes(    /* iterator over all notes (depth first) */
    sir_note_mptr MemberFct,
    sir_note_marg MemberFctArg);

BOOL DFS_FindDependant( /* searches for dependants (depth first) */
    sir_symbol **SkipSymbol,
    sir_symbol *ThatSymbol,
    sir_symbol **DepSymbol,
    sir_statement **DepStmtn,
    sir_expression **DepExpr,
    SIRDEPENDENCY *Reason);

void SetAlias(        /* sets all type, usertype, symbol alias */
    sir_symbols *Alias); /* iterates over symbols and usertypes */

void UnAlias(         /* unalias all type, usertype, symbol links */
    sir_symbols *GlobalsSymbols);

ERROR Integrate(      /* integrates imported symbol table */
    sir_symbols *Imported);

ERROR CheckClassContainment( /* checks class hierarchy for cycles */
    void);

sir_symbol *Find(     /* find an entry globally (here and above) */
    const char *Name); /* returns NULL if not found */

sir_symbol *FindLocally( /* find an entry locally (only in this table) */
    const char *Name); /* returns NULL if not found */

sir_symbol *FindLocally( /* find a symbol with this scope locally */
    sir_symbols *Scope); /* returns NULL if not found */

sir_symbol *Insert(    /* inserts a new symbol */
    sir_symbol *NewSymbol); /* the argument is consumed! */

void FindName(         /* find a symbol or usertype globally */
    const char *Name,
    sir_symbol **SymbolFound, /* the symbol found (or NULL) */
    sir_usertype **UserTypeFound); /* user-type found (or NULL) */

void FindNameLocally(  /* find a symbol or usertype locally */
    const char *Name,
    sir_symbol **SymbolFound, /* the symbol found (or NULL) */
    sir_usertype **UserTypeFound); /* user-type found (or NULL) */

sir_symbol *Declare(   /* declares a new symbol (and prepares def.) */
An object of class SIR_Symbols represents a symbol table. A symbol table contains objects of type SIR_Symbol.
4.36.3 Notes:

- The symbol table is sorted in alphabetical order and can be traversed with the standard list operations.
- The link Parent points to the parent symbol table in the hierarchy of symbol tables. This link is NULL only for the global symbol table.
- The member UserTypes contains user-defined types defined at the same hierarchy level as this symbol table.
- The member ScopeInfo determines the scope of this symbol table. It is one of the following values: SIR_SCOPE_GLOBAL, SIR_SCOPE_CLASS, SIR_SCOPE_PARAMETER, SIR_SCOPE_STATEMENT, SIR_SCOPE_USERTYPE.
- The link ParentSymbol points to the function or class symbol containing this table. It is NULL otherwise.
- The link ParentUType points to the user-defined type containing this table. It is NULL otherwise.
- The link ParentStmt points to the compound statement containing this table. It is NULL otherwise.
- The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.
- The method DFS_ForAllSymbols performs a traversal over all symbols in this subtree in depth-first search order. During the traversal, MemberFct is called for each symbol with the argument MemberFctArg.
- The method DFS_ForAllUserTypes performs a traversal over all user-defined types in this subtree in depth-first search order. During the traversal, MemberFct is called for each user-defined type with the argument MemberFctArg.
- The method DFS_ForAllNotes performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, MemberFct is called for each annotation with the argument MemberFctArg.
- The method Find searches for a symbol by Name globally, that is, in this table and parent tables. NULL (but no error) is returned if no entry is found.
- The method FindLocally searches for a symbol in this list by Name. NULL (but no error) is returned if no entry is found.
• The methods `Declare` and `Define` are used by the SpecC parser to build the symbol table.

• The method `WriteSC` generates SpecC (or C++) source code for all the symbols in this table.
4.37 Class SIR SymbolPtr

The class SIR_SymbolPtr is declared in file IntRep/Symbol.h.

4.37.1 Declaration:

```cpp
1 class SIR_SymbolPtr : /* symbol pointer (alias event) */
2     public SIR_ListElem<SIR_SymbolPtr> /* is a list element */
3 {
4     public:
5     SIR_Symbol *Symbol; /* link to the symbol */
6
7 //++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++
9
10     SIR_SymbolPtr( /* constructor #1 */
12         SIR_Symbol *Symbol);
13
14     SIR_SymbolPtr( /* constructor #3 (duplicator) */
15         SIR_Symbol_ptr *Original);
16
17     SIR_SymbolPtr(void); /* destructor */
18 }
```

4.37.2 Purpose:

An object of class SIR_SymbolPtr represents a reference to a symbol (e.g. an event).

4.37.3 Notes:

- The link Symbol points to the referenced symbol.
4.38 Class SIR SymbolPtrs

The class SIR_SymbolPtrs is declared in file IntRep/Symbol.h.

4.38.1 Declaration:

```c
1 class SIR_SymbolPtrs : /* list of symbol pointers (alias event list) */
2   public SIR_List<SIR_SymbolPtr> /* is basically a list */
3       /* with additional functions */
4 public:
5
6 //++++++++++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++++++++++/
7
8 SIR_SymbolPtrs(     /* constructor #1 */
9     *FirstEntry = NULL);
10
12 SIR_SymbolPtrs(    /* constructor #2 (duplicator) */
13     *Original);
14
15 ~SIR_SymbolPtrs(void); /* destructor */
16
18 void UnAlias(void);   /* unalias all type, usertype, symbol links */
19
21 int CmpSymbolNames(  /* compare all symbol names */
22     *Others); /* (similar to strcmp) */
23
24 *Find( /* searches for a specific entry */
25     *Symbol); /* (returns NULL if not found) */
26 );
```

4.38.2 Purpose:

An object of class SIR_SymbolPtrs represents a list head. The list contains objects of type SIR_SymbolPtr.

4.38.3 Notes:

- The list of symbol pointers is sorted by declaration order and can be traversed with the standard list operations.

- The methods Find search for a symbol pointer in this list by Symbol. NULL (but no error) is returned if no symbol pointer is found.
4.39 Class SIR Transition

The class SIR_Transition is declared in file IntRep/Transition.h.

4.39.1 Declaration:

```cpp
1 class SIR_Transition : /* transition */
2    public SIR_Node, /* is a node */
3    public SIR_ListElem<SIR_Transition> /* and a list element */
4 {  
5 public:
6    sir_symbol *Function; /* link back to the function this is part of */
7    sir_symbol *CurrState; /* link to from-behavior */
8    sir_expression *Condition; /* condition (or NULL for true) */
9    sir_symbol *NextState; /* link to to-behavior (or NULL for break) */
10   unsigned int Line = 0,
11    sir_fileinfo *FileInfo = NULL);
12
13 SIR_Transition() /* constructor #1 */
14    sir_symbol *Function,
15    sir_symbol *CurrState,
16    sir_expression *Condition,
17    sir_symbol *NextState,
18    unsigned int Line = 0,
19    sir_fileinfo *FileInfo = NULL)
20 SIR_Transition() /* constructor #3 (duplicator) */
21    sir_transition *Original);
22
23 SIR_Transition( /* destructor */
24    sir_transition *Original);
25
26 ~SIR_Transition(void); /* destructor */
27
28 sir_transitions *GetList(void); /* determines the list of this transition */
29    /* (returns NULL if not in a list */
30
31 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
32    sir_node_mptr MemberFct,
33    sir_node_marg MemberFctArg);
34
35 ERROR DFS_ForAllExpressions( /* iterator over all expressions (dfs) */
36    sir_expr_mptr MemberFct,
37    sir_expr_marg MemberFctArg);
38
39 void UnAlias(void); /* unalias all type, usertype, symbol links */
40
41 };
```

117
4.39.2 Purpose:

An object of class $\text{SIR\_Transition}$ represents a transition in a $\text{SpecC\ fsm}$ construct.

4.39.3 Notes:

- The link $\text{Function}$ points to the function symbol which contains this state transition.
- The link $\text{CurrState}$ is a link to the behavior symbol denoting the current state of this transition.
- The member $\text{Condition}$ contains the condition for this transition. This expression can be $\text{NULL}$ denoting an implicit $\text{true}$ condition.
- The link $\text{NextState}$ is a link to the behavior symbol denoting the next state of this transition. It is $\text{NULL}$ in case of a $\text{break}$ transition.
- The method $\text{GetList}$ returns a handle to the list of transitions.
- The method $\text{DFS\_ForAllNodes}$ performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, $\text{MemberFct}$ is called for each node with the argument $\text{MemberFctArg}$.
- The method $\text{DFS\_ForAllExpressions}$ performs a traversal over all expressions in this subtree in depth-first search order. During the traversal, $\text{MemberFct}$ is called for each expression with the argument $\text{MemberFctArg}$.
4.40 Class SIR Transitions

The class SIR_Transitions is declared in file IntRep/Transition.h.

4.40.1 Declaration:

```cpp
1 class SIR_Transitions : /* transitions list */
2 public SIR_List< SIR_Transition > /* is simply a list of transitions */
3 { /* with additional member(function)s */
4 public:
5
6 //++++++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++++++++++//
7
8 SIR_Transitions ( /* constructor #1 */
9   sir_transition *FirstEntry = NULL);
10
11 SIR_Transitions ( /* constructor #2 (duplicator) */
12   sir_transitions *Original);
13
14 ~SIR_Transitions (void); /* destructor */
15
16 ERROR DFS_ForAllNodes( /* iterator over all nodes (depth first) */
17   sir_node_mptr MemberFct,
18   sir_node_marg MemberFctArg);
19
20 ERROR DFS_ForAllExpressions( /* iterator over all expressions (dfs) */
21   sir_expr_mptr MemberFct,
22   sir_expr_marg MemberFctArg);
23
24
25 void UnAlias( void); /* unalias all type, usertype, symbol links */
26
27
28 sir_transition *Find( /* find a specific transition */
29   sir_symbol *CurrState, /* (returns NULL if not found) */
30   sir_symbol *NextState);
31
32
33 sir_transition *FindCurrState( /* find the sublist with this start state */
34   sir_symbol *CurrState); /* (returns NULL if not found) */
35
36 sir_transition *FindNextState( /* find the first trans. with this next state */
37   sir_symbol *NextState); /* (returns NULL if not found) */
38 }
39
```
4.40.2 Purpose:

An object of class SIR_Transitions represents a list head. The list contains objects of type SIR_Transition.

4.40.3 Notes:

- The list of state transitions is sorted by CurrState in declaration order and can be traversed with the standard list operations.

- The method DFS_ForAllNodes performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, MemberFct is called for each node with the argument MemberFctArg.

- The method DFS_ForAllExpressions performs a traversal over all expressions in this subtree in depth-first search order. During the traversal, MemberFct is called for each expression with the argument MemberFctArg.

- The method Find searches for a state transition in this list by CurrState and NextState. NULL (but no error) is returned if no state transition is found.

- The method FindCurrState searches for a transition in this list by CurrState. NULL (but no error) is returned if the current state is not found.

- The method FindNextState searches for a transition in this list by NextState. NULL (but no error) is returned if the next state is not found.
4.41 Class SIR Type

The class SIR_Type is declared in file IntRep/Type.h.

4.41.1 Declaration:

```cpp
1 class SIR_Type : /* type (entry in the list of types) */
2     public SIR_ListElem<SIR_Type> /* is a list element */
3 {
4     public:
5         SIR_TYPETYPE Type; /* type of this type (see above) */
6         const char *Name; /* pointer to type name (NULL if unnamed) */
7         sir_type *SubType; /* subtype link (POINTER, ARRAY, FUNCTION) */
8         sir_user_type *UserType; /* link to user type (if STRUCT, UNION, ENUM) */
9         sir_symbol *ClassSymbol; /* link to class symbol (BEH., CHAN., INT.) */
10        sir_type_ptrs *Parameters; /* parameters or ports (FUNC., BEH., CHAN.) */
11        int LeftBound, /* bounds (if BIT or UBIT) */
12            RightBound;
13        int Size; /* array size (if ARRAY) */
14        bool Const; /* const qualifier */
15        bool Volatile; /* volatile qualifier */
16        SIR_DIRECTION Direction; /* in/out/inout qualifier (for ports) */
17        sir_type *Alias; /* alias pointer (temporary only) */
18 }
19
20 //+++++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++/
21
22 SIR_Type( /* constructor #1 (for built-in types) */
23         SIR_TYPETYPE Type,
24         bool Const = false,
25         bool Volatile = false,
26         int LeftBound = 0,
27         int RightBound = 0,
28         SIR_DIRECTION Direction = SIR_PORT_NONE);
29
30 SIR_Type( /* constructor #2 (for composite types) */
31         sir_type *SubType,
32         SIR_TYPETYPE Type,
33         sir_type_ptrs *Parameters = NULL,
34         int Size = 0,
35         bool Const = false,
36         bool Volatile = false,
37         SIR_DIRECTION Direction = SIR_PORT_NONE);
38
39 SIR_Type( /* constructor #3 (for class types) */
40         SIR_TYPETYPE Type,
41         const char *NamePtr,
42```
sir_symbol *ClassSymbol = NULL;
sir_type_ptrs *Parameters = NULL);

SIR_Type( /* constructor #4 (duplicator) */
  sir_type *Original);

SIR_Type( /* constructor #5 (for user-defined types) */
  SIR_TYPETYPE Type,
  const char *NamePtr,
  sir_user_type *UserType,
  bool Const = false,
  bool Volatile = false,
  SIR_DIRECTION Direction = SIR_PORT_NONE);

-SIR_Type(void); /* destructor */

BOOL IsPort (void); /* determines if this is a port type */

BOOL VoidParameters (void); /* determines if there are no parameters */

BOOL IsConvertibleTo (/* determines possibility of conversion */
  sir_type *TargetType); /* (silent standard conversions) */

BOOL SubTypeTreeContains (/* determines special dependency */
  sir_type *TargetType); /* (Bug fix 07/01/00, R.D.) */

static SIR_CONSTTYPE ConstType( /* determines the constant type of the type */
  SIR_TYPETYPE Type);

SIR_CONSTTYPE ConstType(void); /* determines the constant type of this type */

ERROR Check( /* performs general legal checks on this type */
  bool VoidIsOK = false,
  bool IsPort = false,
  bool IsExternOrTypedef = false,
  bool IsPointer = false, /* used in recursion */
  bool IsFunction = false, /* used in recursion */
  bool IsArray = false); /* used in recursion */

ERROR CheckInitializer( /* check if initializer is acceptable */
  sir_initializer *Initializer); /* (works recursively) */

void ConvertInitializer( /* convert initializer to expected type */
  sir_initializer *Initializer); /* (works recursively) */

sir_types *GetTable(void); /* determines the table this type is in */
/* (returns NULL if not in a table) */

sir_type *Modified (/* returns a (probably new) modified type */
```c
bool Const = false; /* must be in a table */
bool Volatile = false;
SIR_DIRECTION Direction = SIR_PORT_NONE;

100 unsigned int SizeOf(
    /* compute the size of this type */
101 )
102    unsigned int *Alignment = NULL, /* obtain alignment */
103    BOOL SpecSize = TRUE); /* Spec or C++ size */
104 static const char *PrintDirection( /* prints a SpecC port direction */
105    SIR_DIRECTION Direction, /* eg. "in" or "out" */
106    BOOL AppendSpace = false);
107
108 const char *PrettyString( /* prints a SpecC type */
109    string *Buffer, /* Buffer must be initialized */
110    BOOL IncludeNotes = FALSE, /* with SymbolName or "" for */
111    BOOL CPlusPlusMode = FALSE, /* abstract types */
112    BOOL UserTypeDefReq = FALSE,
113    BOOL UseShadowName = FALSE);
114
115 void MarkUsed(void); /* mark this type as being used */
116
117 SIR_TYPECLASS TypeClass(void); /* determines the class of this type */
118 }
```

### 4.41.2 Purpose:

An object of class SIR_Type represents a SpecC type.

### 4.41.3 Notes:

- The member Type determines the type of this Type. It is one of the following values: SIR_TYPE_BOOL, SIR_TYPE_CHAR, SIR_TYPE_uchar, SIR_TYPE_SHORT, SIR_TYPE_ushort, SIR_TYPE_INT, SIR_TYPE_UINT, SIR_TYPE_LONG, SIR_TYPE_JLONG, SIR_TYPE_JLONGLONG, SIR_TYPE_FLOAT, SIR_TYPE_DOUBLE, SIR_TYPE_JLONGDOUBLE, SIR_TYPE_BIT, SIR_TYPE_JBIT, SIR_TYPE_VOID, SIR_TYPE_EVENT, SIR_TYPE_POINTER, SIR_TYPE_STRUCT, SIR_TYPE_UNION, SIR_TYPE_ENUM, SIR_TYPE_ARRAY, SIR_TYPE_FUNCTION, SIR_TYPE_ANY_TYPE, SIR_TYPE_BEHAVIOR, SIR_TYPE_CHANNEL, SIR_TYPE_INTERFACE, SIR_TYPE_TIME.

Depending on the Type, only some of the other members are valid. Invalid members are set to NULL or 0.

- The link Name points to the (simplified) name of this type. For a fully specified name, use method Print instead.
• The link **SubType** points to the sub-type of this type for **SIR_TYPE_POINTER**, **SIR_TYPE_FUNCTION**, and **SIR_TYPE_ARRAY**. It is **NULL** otherwise.

• The link **UserType** points to the user-defined type. It is **NULL** if not applicable.

• The link **ClassSymbol** points to the behavior, channel, or interface symbol. It is **NULL** for non-class types.

• The member **Parameters** contains the list of function parameters or class ports. It is **NULL** otherwise.

• The members **LeftBound** and **RightBound** contain the bounds of the vector for bitvector types.

• The member **Size** specifies the size of array types. It is negative (−1) for an unspecified array size.

• The member **Const** contains whether this type is **const**.

• The member **Volatile** contains whether this type is **volatile**.

• The member **Direction** contains the direction for port types. It is **SIR_PORT_IN**, **SIR_PORT_OUT**, or **SIR_PORT_INOUT** for ports, and **SIR_PORT_NONE** for non-port types.

• The link **Alias** is used internally only.

• The method **IsPort** determines whether this type specifies a port.

• The method **IsConvertibleTo** determines whether this type can be converted to **TargetType**.

• The method **GetTable** returns a handle to the (global) type table.

• The method **Modified** returns a type that is different in its flags.

• The method **SizeOf** determines the size of this type.

• The method **PrettyString** generates readable text for an abstract type in SpecC syntax. Optionally, C++ syntax can be selected.

• The method **TypeClass** classifies this type as one of the following: **SIR_TYPECLASS_INTEGRAL**, **SIR_TYPECLASS_FLOATING**, **SIR_TYPECLASS_BITVECTOR**, **SIR_TYPECLASS_CLASS**, **SIR_TYPECLASS_OTHER**.
4.42 Class SIR Types

The class SIR_Types is declared in file IntRep/Type.h.

4.42.1 Declaration:

```cpp
1 class SIR_Types : /* list of all types */
2     public SIR_List <SIR_Type> /* is basically a list */
3     { /* with additional members/methods */
4         public:
5         sir_design *ThisDesign; /* pointer back to the whole design */
6
7  //++++++++++++++++++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++++++++++++++++++
8
9 11 SIR_Types( /* constructor #1 */
10         sir_design *ThisDesign);
11
12 SIR_Types( /* constructor #3 (duplicator) */
13         sir_types *Original,
14         sir_design *ThisDesign);
15
16 ~SIR_Types( void); /* destructor */
17
18 void UnAlias( void); /* unalias all type, usertype, symbol links */
19
20 ERROR Integrate( /* integrates imported type table */
21         sir_types *Imported);
22
23 void GarbageCollection( void); /* garbage collector */
24
25 sir_type *FindOrInsert( /* find an entry or insert it */
26         SIRCONSTTYPE ConstType, /* if it not exists (#1) */
27         int LeftBound = 0, /* (for constant types) */
28         int RightBound = 0);
29
30 sir_type *FindOrInsert( /* find an entry or insert it */
31         SIRTYPETYPE Type, /* if it not exists (#2) */
32         bool Const = false, /* (for simple types) */
33         bool Volatile = false,
34         int LeftBound = 0,
35         int RightBound = 0,
36         SIR_DIRECTION Direction = SIRPORTNONE);
37
38 sir_type *FindOrInsert( /* find an entry or insert it */
39         sir_basic_type *BasicType); /* if it not exists (#3) */
40```

125
4.42.2 Purpose:

An object of class \texttt{SIR\_Types} represents a type table. A type table contains objects of type \texttt{SIR\_Type}. 

126
4.42.3 Notes:

- The type table is global. There is always exactly one type table in a design.
- The type table is sorted in an internal order. Traversal with the standard list operations is possible, but does not make much sense (except for visiting every type).
- The link ThisDesign points to the design this type table is in.
- The methods FindOrInsert obtain a pointer to a type in the type table according to the specified arguments. If the type already exists, a pointer to the existing entry is returned, otherwise a new one is created, inserted and its pointer is returned.

Please refer to class SIR_Type for a description of the arguments to these methods. The argument names match the member names of that class.

- The method TypeCmp checks whether the two types Type1 and Type2 are equivalent. Both types must be in this type table.
4.43 Class SIR TypePtr

The class SIR_TypePtr is declared in file IntRep/Type.h.

4.43.1 Declaration:

```cpp
1  class SIR_TypePtr : /* type pointer */
2      public SIR_ListElem<SIR_TypePtr> /* is a list element */
3  {
4     public:
5       sir_type *Type; /* link to the type */
6     
7     //+++++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++++
9  
10    SIR_TypePtr( /* constructor #1 */
12       sir_type *Type);  
13 
14    SIR_TypePtr( /* constructor #3 (duplicator) */
15       sir_type_ptr *Original); 
16 
17    ~SIR_TypePtr (void); /* destructor */
18  };
```

4.43.2 Purpose:

An object of class SIR_TypePtr represents a reference to a type.

4.43.3 Notes:

- The link Type points to the referenced type.
4.44 Class SIR TypePtrs

The class SIR_TypePtrs is declared in file IntRep/Type.h.

4.44.1 Declaration:

    class SIR_TypePtrs :  /* list of type pointers */
    public SIR_List <SIR_TypePtr>  /* is basically a list */
    {  /* with additional functions */
    public:

    //++++++++++++++++++++++++++++++++++API Layer 1++++++++++++++++++++++++++++++++++/

    SIR_TypePtrs( /* constructor #1 */
    sir_type_ptr *FirstEntry = NULL);

    SIR_TypePtrs( /* constructor #2 (duplicator) */
    sir_type_ptrs *Original);

    SIR_TypePtrs( /* constructor #3 (from Parameters) */
    sir_parameters *Parameters);

    ~SIR_TypePtrs (void);  /* destructor */

    void UnAlias (void);  /* unalias all type, usertype, symbol links */

    static int TypeCmp( /* type comparision (like strcmp) */
    sir_type_ptr *TypeList1,
    sir_type_ptr *TypeList2);

    }

4.44.2 Purpose:

An object of class SIR_TypePtrs represents a list head. The list contains objects of type SIR_TypePtr.

4.44.3 Notes:

- The list of type pointers is sorted by declaration order and can be traversed with the standard list operations.

- The method TypeCmp checks whether the two type lists TypeList1 and TypeList2 are equivalent. All types must be in the same type table.
4.45 Class SIR Unit

The base class SIR_Unit is declared in file IntRep/Unit.h.

4.45.1 Declaration:

```cpp
1 class SIR_Unit /* base class for all units in the design */
2 {
3 public:
4 SIR_COLOR Color; /* color of this unit (for marker algorithms) */
5
6 //+++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++
7
8 SIR_Unit(void); /* constructor #1 */
9 ~SIR_Unit(void); /* destructor */
10 };```

4.45.2 Purpose:

The base class SIR_Unit contains common members and methods of all classes in the design tree.

4.45.3 Notes:

- The member Color determines the color of this object. Color is one of the following: SIR_WHITE, SIR_RED, SIR_YELLOW, SIR_GREEN, SIR_BLUE, SIR_BLACK.

This entry can be used by coloring (or marker) algorithms. By default, each objects color is SIR_WHITE. Every algorithm using the Color entries must clean up when its work is done, in other words, the Color must always be reset to SIR_WHITE.
4.46 Class SIR_UserType

The class SIR_UserType is declared in file IntRep/UserType.h.

4.46.1 Declaration:

```
1 class SIR_UserType : /* usertype (entry in a usertype table) */
2   public SIR_ListElem<SIR_UserType> /* is a list element */
3 {
4   public:
5   SIR_USERTYPE Class; /* class of this usertype (see above) */
6   string *Name; /* name of this usertype (NULL if unnamed) */
7   SIR_type *Type; /* link to the type table entry */
8   SIR_members *Members; /* list of members of this usertype */
9   SIR_symbols *Scope; /* local scope (NULL if none) */
10  SIR_notes *Notes; /* notes for this usertype (NULL if none) */
11  SIR_symbol *TypeDef; /* link to type definition (NULL if none) */
12  SIR_import *Imported; /* link to import file (NULL if not imported) */
13
14  SIR_usertype *Alias; /* alias pointer (temporary only) */
15
16  //+++++++++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++++++++
17
18  SIR_UserType( /* constructor #1 */
19   SIR_USERTYPE Class,
20   const char *Name,
21   SIR_type *Type = NULL,
22   SIR_members *Members = NULL,
23   SIR_symbols *Scope = NULL,
24   SIR_notes *Notes = NULL,
25   SIR_symbol *TypeDef = NULL);
26
27  SIR_UserType( /* constructor #3 (duplicator) */
28   SIR_usertype *Original);
29
30  SIR_UserType( /* destructor */
31     SIR_UserType (void);
32
33  BOOL IsNamed( void); /* checks if Name is defined */
34
35  BOOL IsDefinition (void); /* checks if Members are defined */
36
37  BOOL ContainsNested ( /* checks for nested struct/union */
38   SIR_usertype *UserType); /* (struct/union definitions only) */
39
40
41
```

135
sir_userotypes *GetTable(void); /* determines the table where this is in */
/* (returns NULL if not in a table) */
sir_symbols *GetScope(void); /* determines the scope of this usertype */
/* (returns NULL if links not available) */

ERROR DFS_ForAllNodes(
    /* iterator over all nodes (depth first) */
    sir_node_mptr MemberFct,
    sir_node_marg MemberFctArg);

ERROR DFS_ForAllSymbols(
    /* iterator over all symbols (depth first) */
    sir_symbol_mptr MemberFct,
    sir_symbol_marg MemberFctArg);

ERROR DFS_ForAllUserTypes(
    /* iterator over all usertypes (depth first) */
    sir_usertp_mptr MemberFct,
    sir_usertp_marg MemberFctArg);

ERROR DFS_ForAllNotes(
    /* iterator over all notes (depth first) */
    sir_note_mptr MemberFct,
    sir_note_marg MemberFctArg);

ERROR SetImported(
    /* sets the Imported pointer (if unset) */
    sir_usertp_marg ImportPtr); /* or resets it if NULL */

ERROR TakeOverImport(
    /* takes over an imported usertype */
    sir_usertp_marg ImportPtr);

ERROR MarkImportEntry(
    /* marks its import entry being used */
    sir_usertp_marg /* Unused */);

ERROR MarkUsedTypes(
    /* marks the type entries used here */
    sir_usertp_marg /* Unused */);

ERROR DeleteYourType(
    /* deletes its associated type */
    sir_usertp_marg /* Unused */);

void SetAlias(
    /* sets all type, usertype, symbol alias */
    sir_usertype *Alias); /* (iterates over symbols and usertypes) */

void UnAlias(
    /* unalias all type, usertype, symbol links */
    sir_symbols *GlobalSymbols);

const char *ClassName(void); /* returns its Class as string (eg. "struct") */

static const char *ClassName(
    /* returns Class as string (eg. "struct") */
    SIR_USERTYPE Class);
95  const char *NameOrUnnamed(void); /* returns Name or "<unnamed>" */
96  const char *PrettyString (  /* creates a SpecC string */
97         string *Buffer, /* (one of the ones below) */
98         BOOL IncludeNotes,  
99         BOOL CplusplusMode = FALSE,  
100       BOOL UserTypeDefReq = FALSE);  
101
102  const char *PrettyStringN (  /* PrettyString with name */
103         string *Buffer); /* e.g. "struct Name" */
104
105  const char *PrettyStringT (  /* PrettyString with typedef */
106         string *Buffer); /* e.g. "TypeDefName" */
107
108  const char *PrettyStringD (  /* PrettyString with definition */
109         string *Buffer, /* e.g. "struct [Name] { ... }" */
110         BOOL IncludeNotes,  
111       BOOL CplusplusMode);  
112
113  BOOL LocalUTypeNeedsDef(  /* check whether a user-type member */
114         sir_member *Member); /* needs a local definition */
115
116  ERROR WriteSC (  /* (re-) generates SpecC source code */
117         FILE *File, /* for usertype _definitions_ */
118       BOOL WriteNotes,  
119       BOOL CplusplusMode);  
120
121
122
123  ERROR Annotate(  /* attach a note to the usertype */
124         sir_note *NewNote); /* (consumes NewNote) */
125
126 );

4.46.2 Purpose:

An object of class SIR_UserType represents a user-defined type, in other words, a struct, union, or enum declaration or definition.

4.46.3 Notes:

- The member Class of this user-defined type determines the class. It is one of the following:  
  SIR_USERTYPE_STRUCT, SIR_USERTYPE_UNION, SIR_USERTYPE_ENUM.

- The member Name points to the name of this user-defined type. It is NULL for an anonymous type.

- The link Type points to the type table entry of this user-defined type.
• The member **Members** contains the list of the members of this user-defined type. It is **NULL**, if the user-defined type is only a declaration (not a definition).

• The member **Scope** contains a local symbol table for **struct** and union types, otherwise it is **NULL**.

• The member **Notes** contains the annotations attached to this user-defined type, or is **NULL** if there are none.

• The link **TypeDef** points to the symbol of a type definition for this user-defined type if such a **typedef** exists, otherwise it is **NULL**.

• The link **Imported** points to the design name from which this user-defined type was imported, or is **NULL** otherwise.

• The link **Alias** is used internally only.

• The method **IsNamed** determines if this user-defined type is named.

• The method **IsDefinition** determines if this user-defined type is a definition (not only a declaration).

• The method **GetTable** returns a handle to the user-type table.

• The method **GetScope** returns a handle to the symbol table.

• The method **DFS_ForAllNodes** performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, **MemberFct** is called for each node with the argument **MemberFctArg**.

• The method **DFS_ForAllSymbols** performs a traversal over all symbols in this subtree in depth-first search order. During the traversal, **MemberFct** is called for each symbol with the argument **MemberFctArg**.

• The method **DFS_ForAllUserTypes** performs a traversal over all user-defined types in this subtree in depth-first search order. During the traversal, **MemberFct** is called for each user-defined type with the argument **MemberFctArg**.

• The method **DFS_ForAllNotes** performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, **MemberFct** is called for each annotation with the argument **MemberFctArg**.

• The method **ClassName** returns the type class as a string.
• The method `PrettyString` returns a SpecC (or C++) source string of this user-defined type.

• The method `WriteSC` generates SpecC (or C++) source code of this user-defined type definition.

• The method `Annotate` annotates the label with `NewNote`. 
4.47 Class SIR UserTypes

The class SIR_UserTypes is declared in file IntRep/UserType.h.

4.47.1 Declaration:

1 class SIR_UserTypes : public SIR_List <SIR_UserType> {
2     /* user-defined type "table" */
3     /* is simply a list of user types */
4     /* with additional member(function)s */
5     public:
6     sir_userTypes *Parent; /* link to parent UserType table (or NULL) */
7     sir_symbols *Symbols; /* link back to symbol table on same level */
8
9     //+++++++++++++++++++++++++++++++API Layer 1+++++++++++++++++++++++++++++++/
10
11     SIR_UserTypes( ) /* constructor #1 */
12     sir_userTypes *Parent,
13     sir_symbols *Symbols);
14
15     SIR_UserTypes( ) /* constructor #3 (duplicator) */
16     sir_userTypes *Original);
17
18     ~SIR_UserTypes( void ); /* destructor */
19
20     ERROR DFS_ForAllNodes( ) /* iterator over all nodes (depth first) */
21     sir_node_mptr MemberFct,
22     sir_node_marg MemberFctArg);
23
24     ERROR DFS_ForAllSymbols( ) /* iterator over all symbols (depth first) */
25     sir_symbol_mptr MemberFct,
26     sir_symbol_marg MemberFctArg);
27
28     ERROR DFS_ForAllUserTypes( ) /* iterator over all userTypes (depth first) */
29     sir_userp_mptr MemberFct,
30     sir_userp_marg MemberFctArg);
31
32     ERROR DFS_ForAllNotes( ) /* iterator over all notes (depth first) */
33     sir_note_mptr MemberFct,
34     sir_note_marg MemberFctArg);
35
36     void SetAlias( sir_userTypes *Alias ); /* iterates over symbols and userTypes */
37
38     void UnAlias( sir_symbols *GlobalSymbols );
39
40
41
4.47.2 Purpose:

An object of class SIR\_UserTypes represents a list head. The list contains objects of type SIR\_UserType.

4.47.3 Notes:

- The list of user-defined types is sorted in alphabetical order (with anonymous entries at the end) and can be traversed with the standard list operations.
• The link `Parent` points to the parent user-type table in the hierarchy of symbol and user-defined type tables. This link is `NULL` only for the global user-type table.

• The link `Symbols` points to the symbol table defined at the same hierarchy level as this user-type table.

• The method `DFS_ForAllNodes` performs a traversal over all nodes in this subtree in depth-first search order. During the traversal, `MemberFct` is called for each node with the argument `MemberFctArg`.

• The method `DFS_ForAllSymbols` performs a traversal over all symbols in this subtree in depth-first search order. During the traversal, `MemberFct` is called for each symbol with the argument `MemberFctArg`.

• The method `DFS_ForAllUserTypes` performs a traversal over all user-defined types in this subtree in depth-first search order. During the traversal, `MemberFct` is called for each user-defined type with the argument `MemberFctArg`.

• The method `DFS_ForAllNotes` performs a traversal over all annotations in this subtree in depth-first search order. During the traversal, `MemberFct` is called for each annotation with the argument `MemberFctArg`.

• The method `Find` searches for a user-defined type by `Name` globally, that is, in this table and parent tables. `NULL` (but no error) is returned if no entry is found.

• The method `FindLocally` searches for a user-defined type in this list by `Name`. `NULL` (but no error) is returned if no entry is found.

• The methods `Declare` and `Define` are used by the SpecC parser to build the user-defined type table.

• The method `WriteSC` generates SpecC (or C++) source code for all the user-defined types in this table.
5 The SIR module

From a programmers point of view, the SpecC Internal Representation is basically a program module consisting of a linker library libIntRep.a and a set of C++ header files IntRep.h and IntRep/*.h.

5.1 The SIR header files

The main header file of the SIR data structure is the file IntRep.h. It can be viewed as the only header file for the SIR module because it automatically includes all other SIR header files (IntRep/*.h), so that the complete SIR data structure is declared after a \#include <IntRep.h> statement.

Each of the SIR header files declares one, or a few related classes, and the associated constants, types, and other declarations. For easy reference, the filenames of the header files are named after their classes (e.g. the classes SIR_Statement and SIR_Statements are declared in file IntRep/Statement.h). Please note that the header files contain a lot of useful comments and should also be used as documentation of the SIR data structure, in addition to this document.

Figure 4 lists the complete set of SIR header files that are part of the SCRC, and also shows the inclusion dependencies among them. As stated above, the file IntRep.h automatically includes all other headers, in other words, it depends on them. To be on the safe side, one can always just include IntRep.h. But for example, if some code only needs SIR constants, the inclusion of IntRep/Constant.h is sufficient and much more efficient.

The files IntRep/Declarator.h and IntRep/BasicType.h, drawn with dashed boxes in Figure 4, contain special support for the SpecC parser. The declarations in these files are used exclusively by the parser for storing temporary information. Furthermore, the file IntRep/Extern.h contains a set of definitions with interface information to the bitvector and simulation libraries. All these header files can be ignored for standard SpecC tools (and therefore these are not included in Section 4).

5.2 The SIR library

The file libIntRep.a represents the library of the SIR data structure. It contains all object code of the SIR implementation. Therefore, every program using the SIR must be linked with it.

Furthermore, the SIR library itself depends on the libraries libGlobal.a (containing global declarations and error handling), libbit.a (the SpecC bitvector library), and liblonglong.a (the SpecC longlong library). So when linking a SpecC tool, these libraries are required as well.

For example, in order to compile the CreateHelloWorld program example discussed in Section 7, the following command can be used:

```
g++ CreateHelloWorld.cc -o CreateHelloWorld \
  -I$(SPECC)/inc -L$(SPECC)/lib \
  -lIntRep -lGlobal -lbit -llonglong
```

Here, g++ is the C++ compiler being used and the
Figure 4: SIR header files and their dependencies
environment variable $(SPECC)$ stands for the installation directory of the SpecC system.

6 Error handling

An important issue in program design is error handling. Errors during program execution must be detected and handled in a well-defined way. It is not acceptable to just ignore error conditions (in which case the program most probably crashes with a segmentation or memory fault, or, even worse, produces wrong output leading to serious error conditions later), nor to just abort the program when an error is detected.

This is particularly true for libraries that are to be linked with a larger program. Errors occurring in any library function must be detected and reported to the main program, which solely can decide whether to report the error to the user (in a suitable way), and whether to handle and go on with the error, or to abort the program execution. Also, it is important that even in error conditions data structures are being kept in a clean and well-defined state.

In general, error conditions can be classified into several categories. For example, there are warnings, recoverable errors, and unrecoverable errors. Also, errors can be detected and handled locally in a program module, can be reported to the caller, or can be taken care of globally. As an example for the latter, an out-of-memory condition is best handled globally, so that standard program modules can just assume to always have enough memory available.

In the SpecC system, error handling is based on the conventions defined by the global module libGlobal.a (header file GL_Global.h). This module not only specifies the conventions for error handling across all other modules, it also takes care of out-of-memory conditions. Every allocation and deallocation of dynamic memory is handled here (by overloaded new and delete operators). If no memory is available any more (not even after several retries), the program is aborted with an error message (out-of-memory is an unrecoverable error condition).

For recoverable errors, a set of error codes (integer numbers) is reserved for every program module to report errors. For example, the SIR module can use error codes in the range from SIR_ERROR_BASE up to SIR_ERROR_BASE + SIR_ERROR_RANGE (currently 2000 to 2199). With this scheme, each error condition in a SpecC program can be uniquely identified and handled in the right way.

As a special case, the no-error condition NO_ERROR is defined as 0. This conveniently can be used in C/C++ programs when testing for error conditions in the following fashion:

```c
1 if ( Error )
2  { /* ... handle error */ }
3 /* ... go on */
```

Two programming styles can be used for reporting errors. First, a library function may return an error code directly as its return value. In this case, the return value is either NO_ERROR or one error code from the set of numbers assigned to the library. The caller of the function then can easily detect
and handle an error condition with a code fragment as shown above.

For library functions returning pointers, the second method can be used. In case of an error, the function returns NULL indicating an error condition. The actual error code then can be obtained from a global variable exported by the library. For example, the SIR module exports the variable SIR_Error, which contains the error code in such cases.

It should be noted, that there are functions which don't have any error conditions at all. For such functions, error handling obviously does not need to be supported. The semantics of a function, documented in comments or any other documentation, should contain information about this.

In order for a main program to report errors to the user in a suitable manner, error codes must be combined with a descriptive error message. In most cases, such a message can only be constructed by the module which detects the error condition. Therefore, every module in the SpecC system is required to export a global function XX_ErrorText (where XX is replaced with the module's prefix), which takes an error code as argument and returns a character string describing the error.

For more details and an example for error handling in the SpecC system, please refer to the SpecC source code, in particular the files Global/Global.h, Global/Global.cc, IntRep/Error.cc, IntRep/IntRep/Error.h, or, the example described in Section 7.
7 HelloWorld example program

This section lists and describes a small example program, CreateHelloWorld, which uses the SIR API. The example program creates the famous HelloWorld design from scratch by use of the classes described earlier. Thus, this example demonstrates the use of the SIR API layer 1.

After introducing the example, the low-level SIR data structure of the HelloWorld design is explained. Then, the C++ program CreateHelloWorld.cc is listed and explained. This program builds the described SIR data structure from scratch and generates SpecC source code from it.

7.1 The HelloWorld example

The HelloWorld example is a small design model included in the SpecC distribution (file examples/simple/HelloWorld.sc).

This is the listing HelloWorld.sc:

```
1  note FileName = "HelloWorld.sc";
2
3  extern void printf (const char *, ...);
4
5  behavior Main (void)
6  {
7      int main (void)
8          {
9              char *Name;
10
11             label1: Name = "World";
12             printf ("Hello %s!\n", Name);
13             return (0);
14          }
15  };
```

The purpose of the example is just to give a first impression about the SpecC language. When executed, it just prints Hello World! and exits.

7.2 The low-level HelloWorld data structure

When looking at the SIR data structure, the very small HelloWorld example is already quite complex. Figure 5\(^1\) shows the top part of the HelloWorld design tree, as defined in Section 3.1. Please compare this

\(^1\)The Figures 5, 6 and 7 are created with ddd, the Data Display Debugger. As a special debugging tool, ddd lets the user interactively examine the graphically displayed data structure of a running program. As such, it is quite useful for a programmer
example with the generic tree shown before in Figure 2.

In Figure 5, box 1 represents the root of the design tree. It contains pointers to the list of source files (box 2), the list of imported designs (box 5), the type table (box 6), the global symbol table (box 10), and the list of global annotations (box 17).

For the HelloWorld design, the list of source files only contains one entry, the file named HelloWorld.sc (box 3). The list of imported designs is empty (box 5).

There are 12 type entries in the list of types (box 12). The first three type nodes are shown: type char (box 7), type const char (box 8), and type int (box 9).

The global symbol table (box 10) consists of two entries: the behavior Main (box 11) declared in line 5 (box 13), and the declaration of the external function printf (box 14) in line 3 (box 16).

The list of global annotations (box 17) contains one note (box 18). The note is labeled FileName and contains the string constant HelloWorld.sc (box 20).

In Figure 6, the behavior Main (box 1) is shown in more detail. Its port declaration list (box 3) contains one entry, the type void (box 4 with link to box 13). The symbol table for the behavior body (box 5) contains the method main (box 6), which consists of a list of labels with one entry (boxes 8, 9) and a function body (box 11).

It should be emphasized, that in Figure 6, the arrows to the type nodes (arrows to boxes 12, 13, 14, 15) denote links (not pointers). In the design tree, all types (boxes 12 to 15) are part of the type list and are referred to by links.

Figure 7 shows the function body of the main method as a compound statement (box 1) with its own scope (box 2). In this local symbol table, the symbol Name is declared (box 3). The compound statement also includes a list of (sub-) statements (box 5) that consists of a labeled statement (box 6), an expression statement (box 7), and a return statement (box 8).

Due to the limited space, the expression trees contained in the statements are only shown partially. The labeled statement contains an assignment expression (box 10), the expression statement contains a function call (box 16), and the return statement contains a parenthesized constant expression (boxes 17, 18, 19, 20).

---

to understand a complex data structure. The author of this report recommends using ddd with some SpecC examples in order to fully understand the SIR.

150
7.3 The HelloWorld creator program

The SIR data structure of the HelloWorld example discussed before can be built from scratch by using the methods offered by the SIR classes. The following C++ program starts with creating a new design and then successively adds objects to the data structure. After the HelloWorld data structure is completely built, it is used to generate the SpecC source code shown earlier.

This is the commented listing CreateHelloWorld.cc:

```
1 // ---------------------------------------------------------------
2 // CreateHelloWorld.cc : Example for use of SIR API, layer 1
3 // ---------------------------------------------------------------

4 #include "Global.h"        // SpecC system global declarations
5 #include "IntRep.h"       // SIR data structure declarations

6 int main(void)
7 {
8   sir_design *MyDesign;     // local variables
9   sir_fileinfo *MyFile;
10  sir_note *MyNote;
11  sir_type *MyType,
12     *MyArg1Type,
13     *MyArg2Type,
14     *MyPortType;
15  sir_type_ptrs *MyArgTypes;
16  sir_parameter *MyPort,
17     *MyParameter;
18  sir_symbol *MyBehavior,
19     *MyFunction,
20     *MySymbol;
21  sir_symbols *MyScope,
22      *MyClassScope,
23      *MyLocalScope;
24  sir_statement *MyStatement;
25  sir_labels *MyLabelList;
26  sir_statements *MyStmtList;
27  sir_constant *MyConst;
28  sir_expression *MyExpression;
29  sir_expressions *MyArgs;
30  sir_label *MyLabel;
31  FILE *MyFileHandle;
32 ERROR MyError;
33
34    /* (1) start with an empty design */
35
36    MyDesign = new SIR_Design("HelloWorld");
37
38    /* (2) create a file entry */
39
40    MyFile = new SIR_Filesystem("HelloWorld.txt");
41
42    MyNote = new SIR_Notes("HelloWorld Notes");
43
44    /* (3) add classes */
45
45    MyClassScope = new SIR_Scope("");
46
47    MyScope = new SIR_Scope("HelloWorld");
48
49    /* (4) add ports */
50
51    MyPort = new SIR_Ports("");
52
53    MyParameter = new SIR_Parameters("HelloWorld");
54
55    /* (5) add symbols */
56
56    MySymbol = new SIR_Symbols("HelloWorld");
57`
MyFile = MyDesign->FindOrInsert ("HelloWorld.sc");
/* (3) create the note in line 1 */
MyNote = new SIR_Note("FileName",
    new SIR_Constant (    
        SIRCONST_CHARSTRING,    
        "HelloWorld.sc",    
        1, MyFile));
if ((MyError = MyDesign->Annotate(MyNote)))
    { GL_PrintError (MyError, SIR_ErrorText (MyError));
      delete MyNote;
      delete MyDesign;
      return RESULT_ERROR;
    } /* fi */
/* (4) create the function declaration in line 3 */
MyArg1Type = MyDesign->Types->FindOrInsert (    
    SIR_TYPE_POINTER,    
    MyDesign->Types->FindOrInsert (    
        SIR_TYPE_CHAR, true));
MyArg2Type = MyDesign->Types->FindOrInsert (    
    SIR_TYPE_ANY_TYPE);
MyArgTypes = new sir_type_ptrs (    
    new sir_type_ptr (MyArg1Type));
MyArgTypes->Append (new sir_type_ptr (MyArg2Type));
MyType = MyDesign->Types->FindOrPtr (    
    SIR_TYPE_FUNCTION,
    MyDesign->Types->FindOrPtr (    
        SIR_TYPE_VOID),
    MyArgTypes);
MySymbol = new SIR_Symbol (    
    SIR_SYMBOL_IDENTIFIER, "printf",    
    MyType, SIR_STORAGE_EXTERN, 0,    
    NULL, /* no initializer */    
    3, MyFile);
MyDesign->Symbols->Insert (MySymbol); // insert it in symbol table
/* (5) create the behavior starting in line 5 */
MyPortType = MyDesign->Types->FindOrInsert (SIR_TYPE_VOID);
MyPort = new SIR_Parameter (MyPortType);
MyClassScope = new SIR_Symbols (    
    MyDesign->Symbols, /* (global scope is parent) */    
    SIR_SCOPE_CLASS);
MyBehavior = new SIR_Symbol (    
    SIR_SYMBOL_BEHAVIOR, "Main",    
    NULL, /* type not created yet (symbol is needed first) */    
    new SIR_Parameters (MyPort),
MyClassScope,
    new SIR_SymbolPtrs (), /* empty interface list */
5. MyFile);
MyArg1Type = MyPortType; // build the behavior type
MyArgTypes = new sir_type_ptrs (  
    new sir_type_ptr (MyArg1Type));
MyType = MyDesign->Types->FindOrInsertType(  
    SIR_TYPE_BEHAVIOR, MyBehavior->Name.chars (),
    MyArgTypes, MyBehavior);
MyBehavior->Type = MyType; // fix the type of the behavior
MyDesign->Symbols->Insert (MyBehavior); // insert it in symbol table
MyLocalScope = MyBehavior->ClassScope; // "enter" the behavior scope

    /* (6) create the 'main' method starting in line 7 */

MyArg1Type = MyDesign->Types->FindOrInsert (  
    SIR_TYPE_VOID);
MyArgTypes = new sir_type_ptrs (  
    new sir_type_ptr (MyArg1Type));
MyType = MyDesign->Types->FindOrInsert (  
    SIR_TYPE_FUNCTION,  
    MyDesign->Types->FindOrInsert ( // return type 'int'
    SIR_TYPE_INT),
    MyArgTypes);
MyScope = new SIR_Symbols(  
    MyLocalScope, // (parent is behavior scope)
    SIR_SCOPE_PARAMETER);
MyParameter = new SIR_Parameter (  
    MyArg1Type,
    NULL); // no symbol (type is void) */
MyLabelList = new SIR_Labels (); // build an empty label list
MyFunction = new SIR_Symbol(  
    "main",  
    MyType, SIR_STORAGE_NONE,
    new SIR_Parameters (MyParameter),
    MyScope, MyLabelList,
    NULL, /* no function body yet */
    7, MyFile);
MyStmtntList = new SIR_Statements (); // build an empty statement list
MyStatement = new SIR_Statement ( // function body is a compound stmtnt.
    SIR_STMT_COMPOUND, MyFunction,
    new SIR_Symbols ( // compound statement scope
    MyScope, // (parent is function scope)
    SIR_SCOPE_STATEMENT,
    MyFunction),
    MyStmtntList,
    8, MyFile);
MyFunction->FctBody = MyStatement; // insert compound statement as body
MyLocalScope->Insert (MyFunction); // insert func. in beh. symbol table
MyLocalScope = MyStatement->Scope; // "enter" the statement scope
143     /* (7) create the local variable definition in line 9 */
144
145     MyType = MyDesign->Types->FindOrInsert( // build type 'char'
146         SIR_TYPE_POINTER,
147         MyDesign->Types->FindOrInsert (SIR_TYPE_CHAR));
148     MySymbol = new SIR_Symbol( // build the symbol 'Name'
149         SIR_SYMBOLIDENTIFIER,
150         "Name", MyType,
151         SIR_STORAGE_NONE, 0,
152         NULL, /* no initializer */
153         9, MyFile);
154     MyLocalScope->Insert (MySymbol);          // insert in current symbol table
155     /* (8) create the labeled assignment statement in line 11 */
156
157     MyConst = new SIR_Constant( // build the string constant
158         SIR_CONSTANTCHARSTRING, "World",
159         11, MyFile);
160     MyExpression = new SIR_Expression( // build the assignment expression
161         SIR_EXPRASSIGNMENT,
162         (MyLocalScope->Find("Name"))->Type, /* result type */
163         new SIR_Expression( /* left argument */
164             SIR_EXPRIDENTIFIER,
165             MyLocalScope->Find("Name"),
166             11, MyFile),
167         new SIR_Expression( /* right argument */
168             SIR_EXPRCONSTANT, MyConst,
169             MyDesign->Types->FindOrInsert (SIR_CONSTANTCHARSTRING),
170             11, MyFile),
171         NULL, /* no third argument */
172         11, MyFile);
173     MyStatement = new SIR_Statement( // build the assignment statement
174         SIR_STMT_EXPRESSION,
175         MyFunction, MyExpression,
176         NULL, NULL, /* no sub-statements */
177         11, MyFile);
178     MyLabel = new SIR_Label( // create a label
179         "label1",
180         NULL); /* labeled statement not known yet */
181     MyLabelList->Insert (MyLabel);          // insert the label in the list
182     MyStatement = new SIR_Statement( // create the labeled statement
183         SIR_STMTLABELLED,
184         MyFunction, MyLabel,
185         MyStatement, /* sub-statement */
186         11, MyFile);
187     MyStmtList->Append (MyStatement);        // append to the list of statements
188     MyStmtList->Append (MyStatement);        // fix the statement pointer
189     MyStmtList->Append (MyStatement);        // append to the list of statements
190     MyStmtList->Append (MyStatement);
191     /* (9) create the function call in line 12 */
192
193     MyArgs = new SIR_Expressions ( // create the argument list
194 }
new SIR_Expression(
    SIR_EXPR_CONSTANT,
    new SIR_Constant(
        SIR_CONST_CHARSTRING, "Hello %s!\n"),
    MyDesign->Types->FindOrInsert (SIR_CONST_CHARSTRING),
    12, MyFile));
MyArgs->Append(new SIR_Expression(
    SIR_EXPR_IDENTIFIER,
    MyLocalScope->Find(" Name"),
    12, MyFile));
MyExpression = new SIR_Expression(  // create the function call
    SIR_EXPR_FUNCTION_CALL,
    (MyLocalScope->Find(" printf"))  // result type is return type
    ->Type->SubType,  // of the declared function
    new SIR_Expression(
        SIR_EXPR_IDENTIFIER,
        MyLocalScope->Find(" printf"),
        12, MyFile),
    MyArgs,
    12, MyFile);
MyStatement = new SIR_Statement(  // build the function call statement
    SIR_STMTN_EXPRESSION,
    MyFunction, MyExpression,
    NULL, NULL,
    12, MyFile);
MyStmtList->Append(MyStatement);  // append to the list of statements

/* (10) create the return statement in line 13 */
MyExpression = new SIR_Expression(  // create the return value
    SIR_EXPR_CONSTANT,
    new SIR_Constant (SIR_CONST_INT, 0),
    MyDesign->Types->FindOrInsert (SIR_CONST_INT),
    13, MyFile);
MyStatement = new SIR_Statement(  // create the return statement
    SIR_STMTN_RETURN, MyFunction,
    new SIR_Expression(
        SIR_EXPR_PARENTHESES, /* (optional bracketed )* /
        MyExpression->Type,
        MyExpression,
        (sir_expression * )NULL, ( sir_expression * )NULL,
        13, MyFile),
    NULL, NULL,
    13, MyFile);
MyStmtList->Append(MyStatement);  // append to the list of statements
/* (11) dump the created data structure into a file */
if (! (MyFileHandle = GL_OpenFileForWriting("HelloWorld.sc")))
    { GL_PrintError (GL_Error, GL_ErrorText (GL_Error));
        GL_CloseFile (MyFileHandle);
delete MyDesign;
return (RESULT_ERROR);
} /* fi */
if (((MyError = MyDesign->WriteSC (MyFileHandle, "HelloWorld.sc")))
{ GL_PrintError (MyError, SIR_ErrorText (MyError));
GL_CloseFile (MyFileHandle);
delete MyDesign;
return (RESULT_ERROR);
} /* fi */
if (((MyError = GL_CloseFileWithCheck (MyFileHandle)))
{ GL_PrintError (MyError, GL_ErrorText (MyError));
delete MyDesign;
return (RESULT_ERROR);
} /* fi */
/* (18) clean up and exit */
delete MyDesign; // free the memory
return (RESULT_OK);
} /* end of main */
/* EOF */

When the program listed above is compiled and executed, it creates a SpecC source file HelloWorld.sc.
For comparison with the intended code listed on page 149, the generated SpecC program is shown here:

```c
1 /*---------------------------------------------*/
2 // SpecC source code generated by SpecC V1.1
3 // Design: HelloWorld
4 // File: HelloWorld.sc
5 // Time: Fri Sep 14 11:08:47 2001
6 /*---------------------------------------------*/
7
8 #line 1  "HelloWorld.sc"
9 note  FileName = "HelloWorld.sc";
10
11 #line 13  "HelloWorld.sc"
12 behavior  Main ( void ) ;
13
14 #line 3  "HelloWorld.sc"
15 extern  void  printf ( const char * , ... ) ;
16
17 behavior  Main ( void )
18 {
19
20 #line 22  "HelloWorld.sc"
21     int  main ( void ) ;
22
23 #line 7  "HelloWorld.sc"
24     int  main ( void )
25         {
26             char  * Name;
27
28             label1 : Name = "World";
29             printf ( "Hello %s! \n" , Name);
30             return ( 0 );}
31
32       };
33
34 /*---------------------------------------------*/
35 // End of file HelloWorld.sc
36 /*---------------------------------------------*/
```
8 Conclusion

This section concludes this report with a summary and a brief description of open issues and future work.

8.1 Summary

The SCRC [SCRC] is an Open Source implementation of a compiler and simulator for the SpecC language [BOOK00, BOOK01].

In this report, the software architecture and implementation of the SCRC has been described. In particular, the data structure and the application procedural interface (API) of the SpecC Internal Representation (SIR) has been addressed in detail in this document. The SIR data structure is used by the SpecC Reference Compiler to build, store, and output designs specified in the SpecC language.

The SIR can be used in other SpecC tools as well. It is not limited to be used inside a compiler only. The use of the SIR as a separate module to be used as a basis for new SpecC tools has been described in Section 5, including details such as error handling.

In addition, an example program was shown that demonstrates the use of the API by generating a simple SpecC design example, HelloWorld, from scratch. The main portions of the SIR data structure of this example were shown graphically and explained in detail.

In conclusion, this report showed detailed information about the SCRC, its SIR data structure including its API, so that any C++ programmer can use this to develop and implement new tools for the SpecC language.

8.2 Future work

Future work includes maintenance (e.g. bug fixes) of the current implementation as well as extensions of the implementation if this becomes necessary. Furthermore, it is the goal of the SCRC project to provide a free reference implementation of the latest SpecC language definition. Thus, whenever the SpecC language is refined and a new SpecC Language Reference Manual is approved by the SpecC Technology Open Consortium, the SpecC Reference Compiler will be updated accordingly.
9 References


[CECS] Center for Embedded Computer Systems web pages:
http://www.cecs.uci.edu/.

[SCRC] SpecC Reference Compiler web pages:

[STOC] SpecC Technology Open Consortium web pages:
http://www.stoc.org/.
# Index

<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>API, 6</td>
<td></td>
</tr>
<tr>
<td>API layer, 6</td>
<td></td>
</tr>
<tr>
<td>Base class, 4</td>
<td></td>
</tr>
<tr>
<td>C++ Compiler, 2</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>BitSlice, 9</td>
<td></td>
</tr>
<tr>
<td>BitSlices, 11</td>
<td></td>
</tr>
<tr>
<td>Constant, 13</td>
<td></td>
</tr>
<tr>
<td>Constraint, 17</td>
<td></td>
</tr>
<tr>
<td>Constraints, 19</td>
<td></td>
</tr>
<tr>
<td>Design, 21</td>
<td></td>
</tr>
<tr>
<td>Event, 25</td>
<td></td>
</tr>
<tr>
<td>Events, 27</td>
<td></td>
</tr>
<tr>
<td>Exception, 29</td>
<td></td>
</tr>
<tr>
<td>Exceptions, 33</td>
<td></td>
</tr>
<tr>
<td>Expression, 35</td>
<td></td>
</tr>
<tr>
<td>Expressions, 41</td>
<td></td>
</tr>
<tr>
<td>FileInfo, 43</td>
<td></td>
</tr>
<tr>
<td>FileList, 45</td>
<td></td>
</tr>
<tr>
<td>Import, 47</td>
<td></td>
</tr>
<tr>
<td>ImportList, 49</td>
<td></td>
</tr>
<tr>
<td>Initializer, 51</td>
<td></td>
</tr>
<tr>
<td>Initials, 53</td>
<td></td>
</tr>
<tr>
<td>Label, 55</td>
<td></td>
</tr>
<tr>
<td>Labels, 57</td>
<td></td>
</tr>
<tr>
<td>LineInfo, 59</td>
<td></td>
</tr>
<tr>
<td>List, 61</td>
<td></td>
</tr>
<tr>
<td>ListElem, 65</td>
<td></td>
</tr>
<tr>
<td>Member, 67</td>
<td></td>
</tr>
<tr>
<td>Members, 69</td>
<td></td>
</tr>
<tr>
<td>Node, 71</td>
<td></td>
</tr>
<tr>
<td>Note, 73</td>
<td></td>
</tr>
<tr>
<td>Notes, 77</td>
<td></td>
</tr>
<tr>
<td>Parameter, 79</td>
<td></td>
</tr>
<tr>
<td>Parameters, 81</td>
<td></td>
</tr>
<tr>
<td>PortMap, 83</td>
<td></td>
</tr>
<tr>
<td>PortMaps, 85</td>
<td></td>
</tr>
<tr>
<td>Statement, 87</td>
<td></td>
</tr>
<tr>
<td>Statements, 93</td>
<td></td>
</tr>
<tr>
<td>Symbol, 97</td>
<td></td>
</tr>
<tr>
<td>SymbolPtr, 113</td>
<td></td>
</tr>
<tr>
<td>SymbolPtrs, 115</td>
<td></td>
</tr>
<tr>
<td>Symbols, 107</td>
<td></td>
</tr>
<tr>
<td>Transition, 117</td>
<td></td>
</tr>
<tr>
<td>Transitions, 119</td>
<td></td>
</tr>
<tr>
<td>Type, 121</td>
<td></td>
</tr>
<tr>
<td>TypePtr, 129</td>
<td></td>
</tr>
<tr>
<td>TypePtrs, 131</td>
<td></td>
</tr>
<tr>
<td>Types, 125</td>
<td></td>
</tr>
<tr>
<td>Unit, 133</td>
<td></td>
</tr>
<tr>
<td>UserType, 135</td>
<td></td>
</tr>
<tr>
<td>UserTypes, 141</td>
<td></td>
</tr>
<tr>
<td>CLI, 6</td>
<td></td>
</tr>
<tr>
<td>Delparser, 2</td>
<td></td>
</tr>
<tr>
<td>Design representation</td>
<td></td>
</tr>
<tr>
<td>Internal, 2</td>
<td></td>
</tr>
<tr>
<td>Design tree, 4</td>
<td></td>
</tr>
<tr>
<td>Error handling, 147</td>
<td></td>
</tr>
<tr>
<td>Example, 149</td>
<td></td>
</tr>
<tr>
<td>HelloWorld, 154</td>
<td></td>
</tr>
<tr>
<td>HelloWorld 1, 149</td>
<td></td>
</tr>
</tbody>
</table>
GUI, 6

Header files, 145
Hierarchy layer, 6

Kernel, 6

Library, 145
Linker, 2

Module, 145

Object classes, 9

Parser, 2
Preprocessor, 2

SIR, 2
  API layer, 6
  Base class, 4
  Classes, 9
  Design tree, 4
  Error Handling, 147
  example, 149
  Header files, 145
  Library, 145
  Link, 4
  Module, 145
  Pointer, 4
SpecC compiler, 2
  Program flow, 2

Translator, 2