SAVANT Programmer's Manual

(Documentation for version 1.0)

Dale E. Martin and *Philip A. Wilsey*
Computer Architecture Design Laboratory
Dept of ECECS, PO Box 210030
Cincinnati, OH 45221–0030
savant@ececs.uc.edu

Published by the University of Cincinnati
Dept of ECECS, PO Box 210030
Cincinnati, OH 45221–0030 USA

Contents

Chapter 1

Introduction

SCRAM is a VHDL analyzer and extensible environment for CAD tool backend development. VHDL descriptions are analyzed and stored in the SAVANT intermediate form (IF), which is comptaible with the Advanced Intermediate Representation with Extensibility (AIRE). Provisions to support the insertion of backend CAD tools have been provided. More precisely, a technique for structuring and instantiating the nodes of the IF built by SCRAM has been developed that allows the CAD tool builder to easily insert additional capabilities into the IF structure. The specific technique is described in the following section. The effect is the construction of an extensible class structure for the SAVANT IF as outlined in the SAVANT Overview document.

The SCRAM parser is constructed as an LL(2) grammar using the Purdue Compiler Construction Tool Set (PCCTS) parser generator (many thanks to Terence Parr, Hank Dietz, Will Cohen, and to Tom Moog who has continued improving PCCTS 1.33). PCCTS is available at `http://www.polhode.com/pccts133mr.html`. SCRAM has been developed in C++ using GNU's EGCS 1.1.1 compiler. Effort has been made to maintain compatibility with GNU's g++ 2.7.2 for the 1.0 release of SAVANT - future versions of SAVANT most likely will use features found only the the newer versions of the the GNU EGCS compilers, such as exceptions and/or namespaces.

Chapter 2

Introduction

2.1 Introduction to SAVANT

SAVANT? Standard Analyzer of VHDL Applications for Next-generation Technology.

Who? MTL Systems and the University of Cincinnati. Funding by Wright Labs, SBIR Phase II.

What? Create a freely available, extensible VHDL Analyzer. Provided value added benefit and market.

Why? Stimulate integration of VHDL technology into the CAD research community.

2.2 The Software Components of SAVANT

SCRAM: a VHDL analyzer

Extensible classes: to build intermediate forms for

- IIR from the AIRE spec
- FIR from the AIRE spec [1]

Extension classes:

- transmute: eliminating redundant structures in the IIR
- publisher: outputting VHDL and C++[2]
- archiver: extensions to read/write FIR files to/from IIR trees

2.3 Software Availability

- Copyrighted by the University of Cincinnati
- Freely Redistributable under the terms of the GNU Public Library License. (LGPL)

[1] FIR support has been suspended and the existing code removed from SAVANT due to an incomplete specification at the time that the implementation was being done. FIR support could occur in the future if resources become available. Patches of the existing FIR implentation which are no longer being maintained could be made to interested parties on request.

[2] The development of the C++ output is sponsored by DARPA.

2.4 The Intermediate Forms

AIRE: Advanced Intermediate Representation with Extensibility

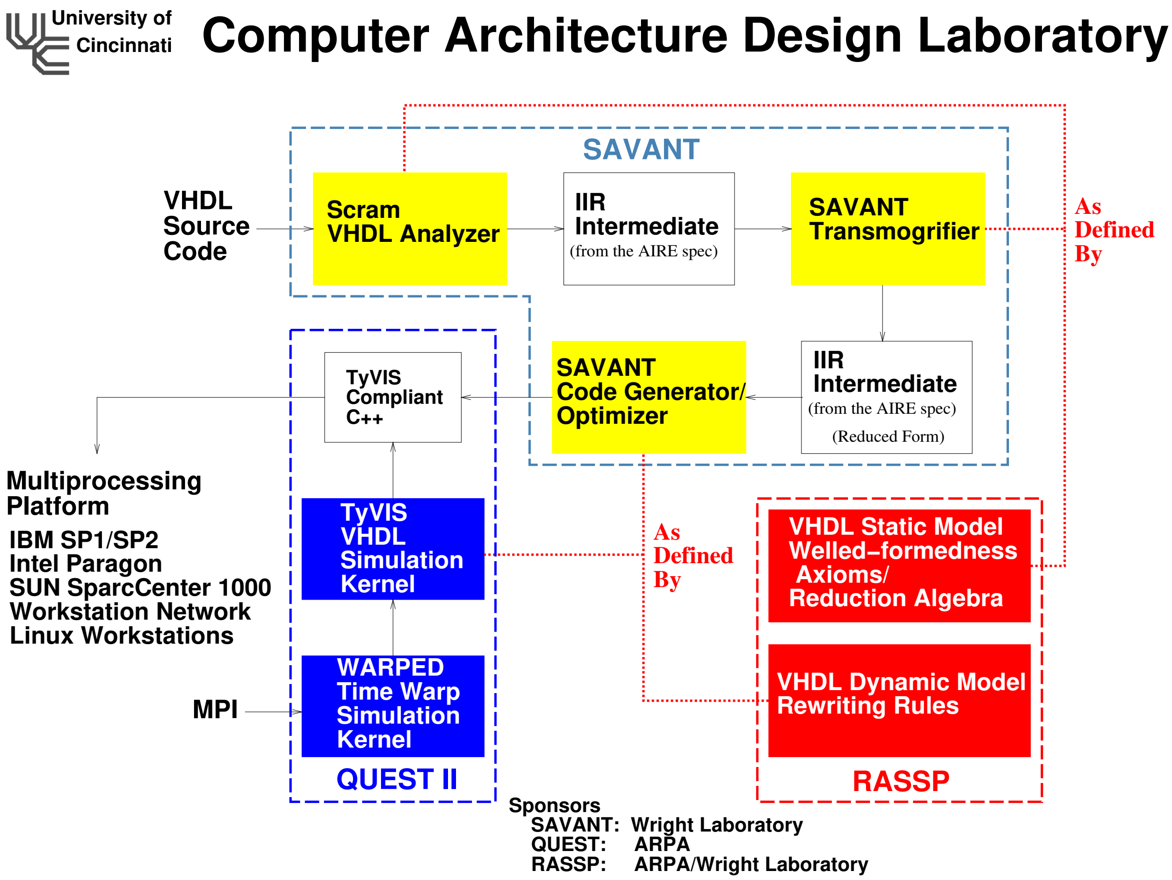
1. Formats:

   IIR: in-memory intermediate representation

   FIR: file intermediate representation

2. Jointly developed by:

   - University of Adelaide (Australia)
   - DA Solutions Ltd (UK)
   - FTL Systems, Inc. (USA)
   - MTL Systems, Inc. (USA)
   - University of Cincinnati (USA)



2.5 AIRE: Current Status

- IIR:
  - Draft design complete
  - Implementations by UC, FTL, and other parties
  - Extensible object-oriented design
- FIR:
  - Draft design in progress
  - Implementations under-development
  - Extensible object-oriented design
- EIA standardization in progress

2.6 SCRAM: Current Status

- LL(2) grammar
- Flex lexer, ANTLR (from PCCTS) parser generator

- Parsing: complete language coverage
- IIR node construction: complete
- Type checking: implemented, tested and validated
- Overloaded name resolution implemented

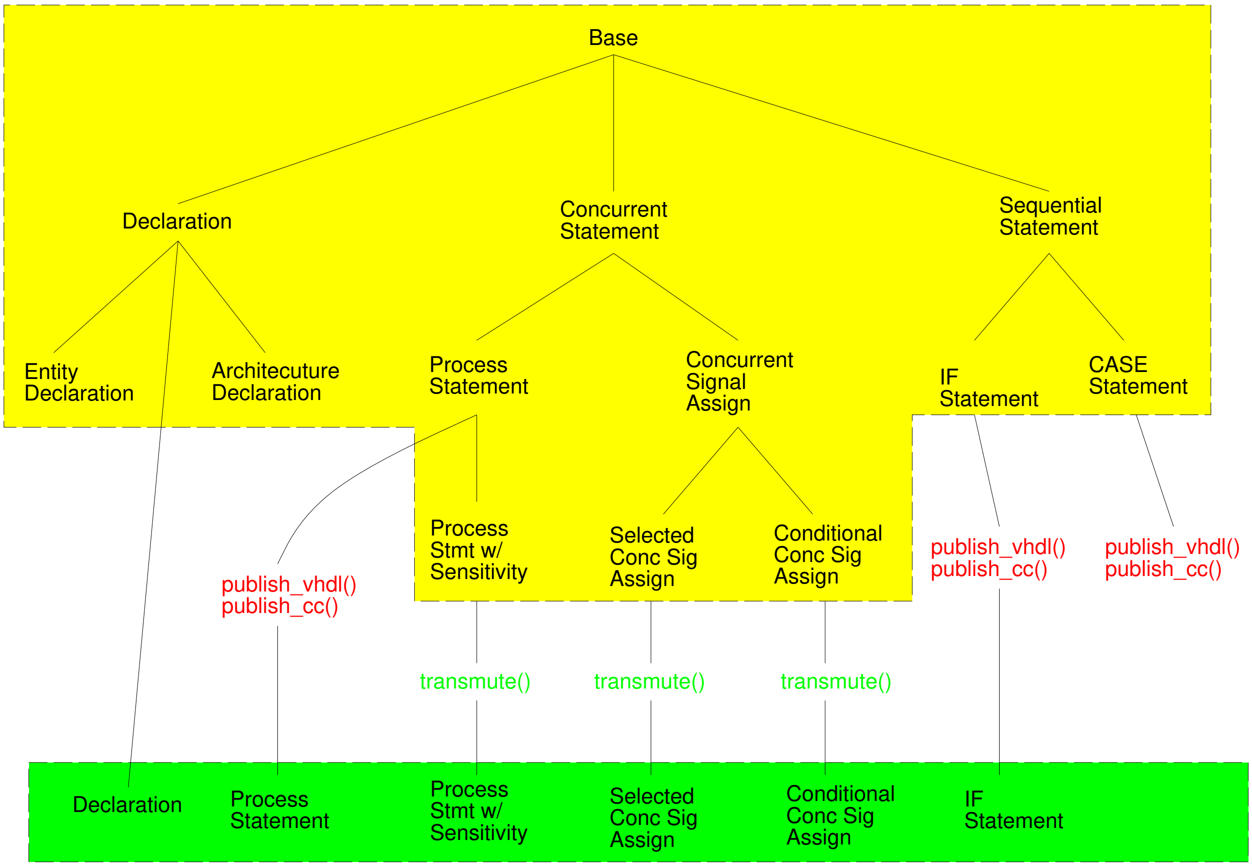
2.7 SAVANT Implementation of AIRE: Current Status

- IIR implementation complete
- FIR implementation suspended

2.8 Extensibility in SAVANT

- Achieved by user added derived classes to IIR/FIR
- Allow addition of data and methods to all AIRE classes
- Provide user full benefits of object-oriented programming

2.9 Class Relationships Supporting Extensibility



2.10 Transmute: Current Status

- Base rewriting are all implemented
- Operational on a node-by-node basis (integrated with publish)
- Available for use by extension classes

2.11 Record/Playback: Current Status

- Initial implementation of FIR class nodes completed in 1998
- Implementation suspended due to incomplete specification
- Specification is reported to be complete, and if resources become available it's possible that FIR support could be reintegrated
- Library support in SAVANT currently accomplished through VHDL publishing and reparsing

2.12 Publish: Current Status

- VHDL regeneration complete
- C++ for TyVIS operational and complete

2.13 Auxiliary Support Tools

- Flex: lexer generator
- PCCTS: parser-generator
- C++ compiler
- Assorted unix tools, e.g., make

2.14 Project Web Sites

- `http://www.ececs.uc.edu/~paw/savant/`

2.15 Related Web Sites

- `http://www.ececs.uc.edu/~paw/aire/`
- `http://www.ececs.uc.edu/~paw/quest/`
- `http://www.ececs.uc.edu/~paw/tyvis/`
- `http://www.ececs.uc.edu/~paw/warped/`
- `http://www.ececs.uc.edu/~paw/rassp/`

Chapter 3

SAVANT INSTALLATION

3.1 Installation

- Requirements
  - Resources
  - Tools
- Where to get it
- Installation Procedure
- Updating Your Distribution
  - Via patches
  - CVS import

3.2 Resource Requirements

Approximate Disk Space Requirements (for Linux and Solaris, varies with OS and compiler)

- 10M of Disk space for distributed source code or binaries
- 70M of Disk space for compilation
- 18M of Disk space for executable (with debug information)
- 3M of Disk space for executable (without debug information)

Virtual Memory

- At least 40M of virtual memory (for compilation).

3.3 Tool Requirements

tar Gnu tar version 1.2 is known to work

gunzip Gnu gunzip version 1.2.4 is known to work

Make Gnu make version 3.77 is known to work

Lexer Flex version 2.5.4 is known to work

Parser Generator PCCTS version 1.33MR12b is known to work

C++ compiler A full featured C++ compiler is required. Known to work:

- g++ version 2.7.2 (and later)
- EGCS g++ version 1.1.1
- Sunpro C++ version 4.1
- Digital Unix cxx C++ compiler

Operating System Environment Unix. Known to work:

- Linux versions 2.0 and 2.2
- Solaris 2.6
- Digital Unix

3.4 Where to get it

SAVANT:

- Source code, Debian Linux packages, and Solaris 2.6 packages `ftp://ftp.ececs.uc.edu/pub/users/dmartin/`
- Debian Linux packages - `ftp://ftp.debian.org`

GNU Tools: Make, Flex, g++, gnu tar, gunzip, and cvs

- `ftp://prep.ai.mit.edu/pub/gnu/`
- `ftp://gatekeeper.dec.com:/pub/GNU/`

PCCTS:

- `http://www.polhode.com/pccts133mr.html`

Linux:

- `http://sunsite.unc.edu/pub/Linux/distributions/` or
- `ftp://ftp.yggdrasil.com/mirrors/sunsite/INDEX.html` or
- `ftp://ftp.debian.org/pub/debian`

3.5 Downloading/unpacking Tips

- Always use **binary mode** when downloading files with ftp.
- SAVANT is distributed in a **gzip**'ed **tar**file.
- To uncompress the gzip'ed file `foo.gz`, type:

  ```
  gunzip foo.gz
  ```

- To unarchive a tar'ed file `foo.tar`, type:

  ```
  tar -xf foo.tar
  ```

- To unarchive a gzip'ed tar file `foo.tgz`, type:

  ```
  tar -xzf foo.tgz
  ```

- Suffixes can be cumulative: a gzip'ed tar file may be named: `foo.tar.gz`, which is unpacked by

  ```
  gunzip foo.tar.gz
  tar -xf foo.tar
  ```

3.6 Unix Environment Variables

- Environment variables exist in the shell to pass information to application programs.
- SAVANT uses two environment variables (`PCCTSROOT` and `SAVANTROOT`) to determine the location of header files and libraries.
- Depending on the shell you use, there are different methods for setting environment variables. See your shell documentation for details.

3.7 Setting Environment Variables (For the shells: `bash` and `sh`)

```
set FOO=/home/john-doe
export FOO
```

3.8 Setting Environment Variables (For the shells: `csh` and `tcsh`)

```
setenv FOO /home/john-doe
```

3.9 SAVANT Environment Variables

PCCTSROOT: This variable specifies the location of the header files included with `pccts`. Needed to compile SAVANT.

SAVANTROOT: This variable specifies the the top level directory of the SAVANT distribution. Needed to compile SAVANT. Need for the execution of SCRAM if not installed in a "standard" subdirectory, such as `/usr` or `/usr/local`.

3.10 Preparing SAVANT for compilation

1. Set the environment variables `PCCTSROOT` and `SAVANTROOT`.

2. The SAVANT distribution is shipped with a configuration script to generate a Makefile for your environment. This script is named `configure` and is in the directory `savant/src`. To configure the system, execute the following commands:

```
cd savant/src
./configure
```

3.11 The `configure` script

- The `configure` script will generate two Makefiles - `Makefile` in the `savant/src` directory and `Makefile.common` in the `savant/lib` directory, from the files `Makefile.in` and `Makefile.common.in` respectively.

- The script searches for a C++ compiler on your system. If the compiler you want to use is not in the list the script knows about, or if it locates a compiler other than the one you want to use, you can explicitly name a compiler in the environment variable `CXX`. This variable only needs to be defined while the script is running; it can be unset after the script has finished.

3.12 Building Dependencies

To build the SAVANT system, the `make` utility is used. `make` requires a list of dependencies to be generated into the file `.depend`. To generate these dependencies, issue the following commands:

```
cd savant/src
make depend
```

If only life was so simple. The generation of dependencies has been the main problem that users of SAVANT have asked the developers about. Initially SAVANT's Makefiles were written to call `makedepend` which is actually part of the X Window system. Eventually, the large number of files in the intermediate caused the "stock" `makedepend` to fail. One solution to this problem is to recompile `makedepend` with some `#define`s increased - see the file `INSTALL` for details. This isn't that simple to do since the program includes many files from the X Windows source - a full X source distribution is required. The developers responded to this problem by shipping binaries for the expanded `makedepend` for some common platforms with the source archive. This solution works if you're using a supported platform.

Another approach is to use `g++` to generate dependencies. The configure script looks for `g++` specifically for this purpose (even if you've specified another compiler in the `CXX` environment variable). However, the developers have experienced problems with this method - it appears that either `make` or `g++` has a bug in it.

The file `savant/lib/Makefile.common{.in}` defines a variable `MAKEDEPEND`. The configure script sets it one of two ways:

- `MAKEDEPEND=-rm -f .depend; g++ -MM -MG $(MDFLAGS) $(CPPFLAGS) $(MAKEDEP_SRCS) > .depend`

  or

- `MAKEDEPEND=-rm -f .depend; ../bin/makedepend.$host_os -f- -- $(MDFLAGS) $(CPPFLAGS) -- $(MAKEDEP_SRCS) > .depend`

In the second example shown above, `$host_os` will be replaced with the OS portion of the output from `config.guess`. For a most Linux machines, for example, the full binary name would be `makedepend.linux-gnu`. These examples are provided to allow the reader to determine the best solution for dependency generation available to them, and to provide them with the capability to insert their own solution into the Makefile.

3.13 Notes about Building Dependencies

1. it's a good idea to build dependencies any time you run the `configure` script for the first time.
2. anytime there is a change made to a source file in the SAVANT system involving a `#include`, this procedure needs to be performed.

3.14 Compiling the system

To compile the system, issue the commands:

```
cd savant/src
make
```

Chapter 4

CVS and SAVANT Updates

4.1 Source Code Control

Code control systems have many benefits for software developers. CVS is one example of such a system. Its features include:

- Support for Concurrent Software Development
- A common Central Repository of the Software
- Version Tags, Revision numbers, and Timestamps
- Support for off-site Importing and Exporting of revised versions
- Automatic Inclusion of Log Messages

The SAVANT software developers use cvs in a very concurrent and complex setting. It works very well for them, and they'd recommend it to anyone doing any type of software development, large or small.

4.2 Coordinating with SAVANT Distributions

- cvs supports the `import`ing of revised versions from a remote site
- **Recommendation:** users of SAVANT will benefit from using cvs to manage their software for integration (by cvs `import`) with revisions to the SAVANT software
- Alternatives to cvs do exist, but will not be explained here or in the SAVANT distributed software.

4.3 Creating a CVS Archive

If this is the first CVS archive you have ever set up, first you will need to set up an archive directory. The archive directory can be specified by the environment variable `CVSROOT`. Assume that `my_cvs_archive` is to be the name of the the archive directory. Then it's creation is achieved (in csh) by issuing the following commands:

```
cd
mkdir my_cvs_archive
mkdir my_cvs_archive/CVSROOT
setenv CVSROOT my_cvs_archive
```

4.4 Importing New Code into an Archive

Importing your code for the first time. For this example, our project name is `my_project` and the initial revision of the code is in the subdirectory `work/src`. We'll call our organization `savant_geeks` and make this revision `1_2`.

```
cd work/src
cvs import my_project savant_geeks 1_2
```

- cvs will invoke your default editor for entry of a log message
- edit the buffer with the desired log message
- save the buffer
- exit the editor and the log message will be applied

4.5 Explanation of the `import` Command

The syntax for `cvs import` is:

```
cvs import [repository] [vendor-tag] [release-tags]
```

where

repository This is the name you want to give the project. There may be more than one repository per CVS archive, so they must be uniquely named. Furthermore, when checking code out of the archive, it is placed in a directory with the same name as the repository.

vendor-tag This simply names the authoring "vendor".

release-tags This names any tags you would like to give the initial check-in. Tags can be used later to retrieve specific revision of files.

4.6 Checking out a CVS module

After checking in the initial revision of your application, it is important to check out a working copy from the archive. This is done by using the command

```
cvs checkout [module]
```

CVS will create a subdirectory with the same name as the module in the current directory. This is your working directory, and changes made to the files in this directory do not affect the archive until they are explicitly committed.

4.7 Committing Changes to the Archive

Changes are committed to the archive using the command:

```
cvs commit
```

The names of the files to commit may optionally follow the command. If no file names are given, CVS will recurse through all subdirectories, and commit all files that have been modified since they were checked out of the archive.
Notes:

1. Each time files are committed to the archive your editor will be invoked for a log message.
2. cvs commands operate implicitly on files in your current directory.

4.8 Adding Files to the Archive

New files to be added to the CVS archive must be explicitly added using the `cvs add` command. For example, addition of the file `foo` using the

```
cvs add foo
```

Notes:

1. `foo` can be a list of files.
2. Each time file(s) are added to the archive your editor will be invoked for a log message.
3. cvs commands operate implicitly on files in your current directory.

4.9 Removing Files from the Archive

To remove a file (*e.g.,* `foo`) from the archive:

1. remove the `foo` from the working directory
2. issue the command: `cvs remove foo`

Notes:

1. CVS keeps deleted files available in case requests to checkout old versions of the software are issued.
2. CVS will refuse to delete a file that still exists in the working directory.

4.10 Importing new SAVANT Releases into Your Archive

- If you are developing software using the SAVANT system, and a new SAVANT release becomes available, the following steps can be used to import the new release into your archive.

- For example to import a newly released version of savant (0.2.99d in this example), into the CVS archived named `testing` you would type:

  ```
  tar -xvf savant-v0.2.99d.tar
  cd savant
  cvs import testing dale v0_2_99d
  [ type in log message when prompted ]
  ```

- Syntax:
  `cvs import [repository] [vendor-tag] [release-tags]`

4.11 Output Generated from CVS Commands

CVS will report file modifications as they occur, in the following format:

```
U testing/src/aire/IIRBase/IIRBase_ContextClause.hh
U testing/src/aire/IIRBase/IIRBase_LibraryDeclaration.cc
M testing/src/aire/IIRBase/IIRBase_LibraryDeclaration.hh
C testing/src/aire/IIRBase/IIRBase_ComponentDeclaration.hh
```

4.12 Meaning of the Output Generated by CVS

U This file was updated by the import.

M This file was modified by you.

C Both you and the off-site developers modified this file, and CVS was unable to automatically resolve the conflict. (the differences will be saved in the file and surrounded by

```
<<<<<<< working-filename
=======
>>>>>>> version-number
```

Chapter 5

The AIRE Standard

5.1 Introduction to the AIRE specification

Who?
Initiated by Wright Labs, FTL Systems, and the University of Cincinnati.

What?
Create a specification for a freely available, highly portable, extensible VHDL intermediate form.

Why?
To allow easy tool interoperability.

5.2 AIRE Objectives

- Develop portable, memory-based and file-based representation of HDL designs
- Permit free redistribution tools complying with the specification
- Permit free implementation of tools complying with the specification
- Permit interoperability and exchange of
  - VHDL backend analysis tools
  - VHDL libraries access utilities
  - VHDL libraries

5.3 AIRE Fundamentals

Fundamental aspects of AIRE:

Object-Oriented
The AIRE specification is presented as a class hierarchy.
Each class has methods defined that implement its functionality.

Portable Across Languages
Allows implementation in variety of OO languages
Does not preclude implementations in non-OO language, *e.g.*, C or Ada

5.4 AIRE Overview

1. Formats:

   IIR: in-memory intermediate representation

   FIR: file intermediate representation

2. AIRE is a working group under EIA.

3. Scope: IEEE Std 1076–87 (VHDL 87), 1076-93 (VHDL 93) 1076.1 (Mixed Signal), 1364 (Verilog as VHDL).

4. Jointly developed by:

   - University of Adelaide (Australia)
   - DA Solutions Ltd (UK)
   - FTL Systems, Inc. (USA)
   - MTL Systems, Inc. (USA)
   - University of Cincinnati (USA)

5.5 AIRE Documents

- Available in both HTML and postscript forms.

- HTML version fully hyper-linked and X-referenced

- Primary viewing sites:

  - `http://www.vhdl.org/vi/aire/`
  - `http://www.ftlsystems.com/aire/Index.htm`
  - `http://www.ececs.uc.edu/~paw/aire/`
  - `http://www.cs.adelaide.edu.au/users/vide/aire/`

5.6 AIRE Fundamental Data Types

IR_Boolean: an enumeration type defining values `TRUE` and `FALSE`.

IR_Character: the domain of values covering the ISO 8859-1 standard.

IR_Int32: a domain of discrete values covering the range
-2,147,483,647 to +2,147,483,647.

IR_Int64: a domain of discrete values covering the range
-4,611,686,014,132,420,609 to +4,611,686,014,132,420,609.

IR_FP32: representing the 32-bit floating point type in IEEE Std. 754.

IR_FP64: representing the 64-bit floating point type in IEEE Std. 754.

IR_Kind: an enumeration uniquely qualifying each class.

5.7 IIR Overview

- Object-oriented, extensible data structure
- No public data in base definition
- Only public interface is by method invocation
- Approximately 227 class definitions
- Quasi-stable, current version: 4
- Two implementations under development
  - Auriga: FTL Systems, Inc
  - SAVANT: UC/MTL Systems, Inc
- Singlely rooted (at `class IIR`)

5.8 Chief Design Requirements for IIR

- Language Scope (coverage of 1976-87/93, 1976.1, 1364, Trial Implementation Draft of 1/24/97 Including Digital VHDL & VHDL-AMS support)
- Supportive of Multi-backend analysis tools
- Portable
- Extensibility
- Efficiency
- Support integration and product exchange
- Security (offer support for IP)
- Availability (non-proprietary, free or nearly free)

5.9 Real Object-Oriented Definition

- Other intermediates defined as OO, but in practice were not really OO
- IIR is defined as an OO structure
- IIR supports use as an OO structure

5.10 Example of the Derivation Hierarchy

- IIR
  - IIR_DesignFile
  - IIR_Comment
  - . . .
  - IIR_SequentialStatement
    - IIR_AssertionStatement
    - IIR_BreakStatement
    - . . .
  - IIR_ConcurrentStatement

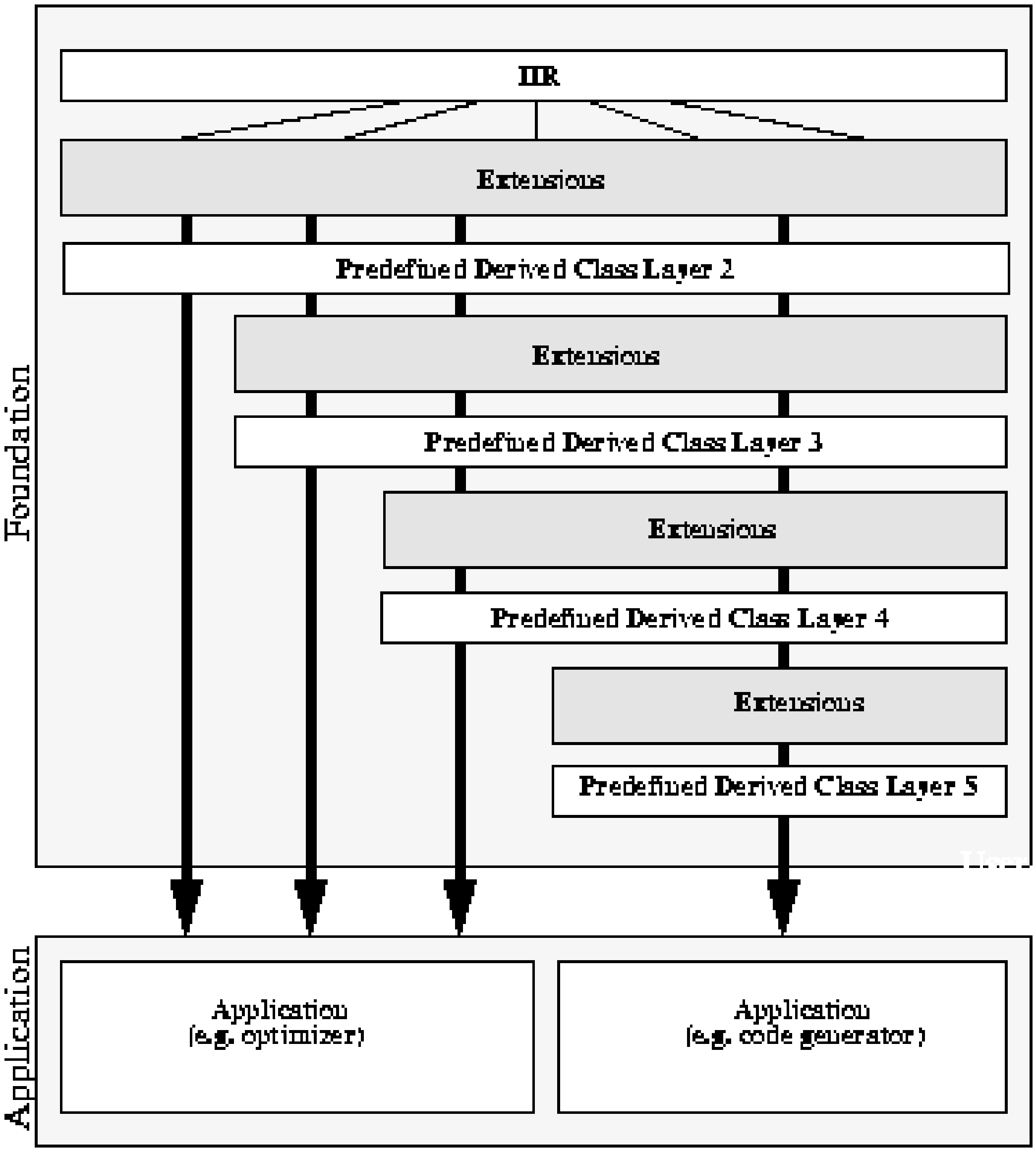
5.11 Constructors/Destructors

- All constructors must have no arguments
- No constraints on destructors

5.12 Interior Classes

- Some intermediate classes not directly instantiatable
- *e.g.*, IIR_SequentialStatement
- Interior classes are indirectly instantiated by instantiation of children

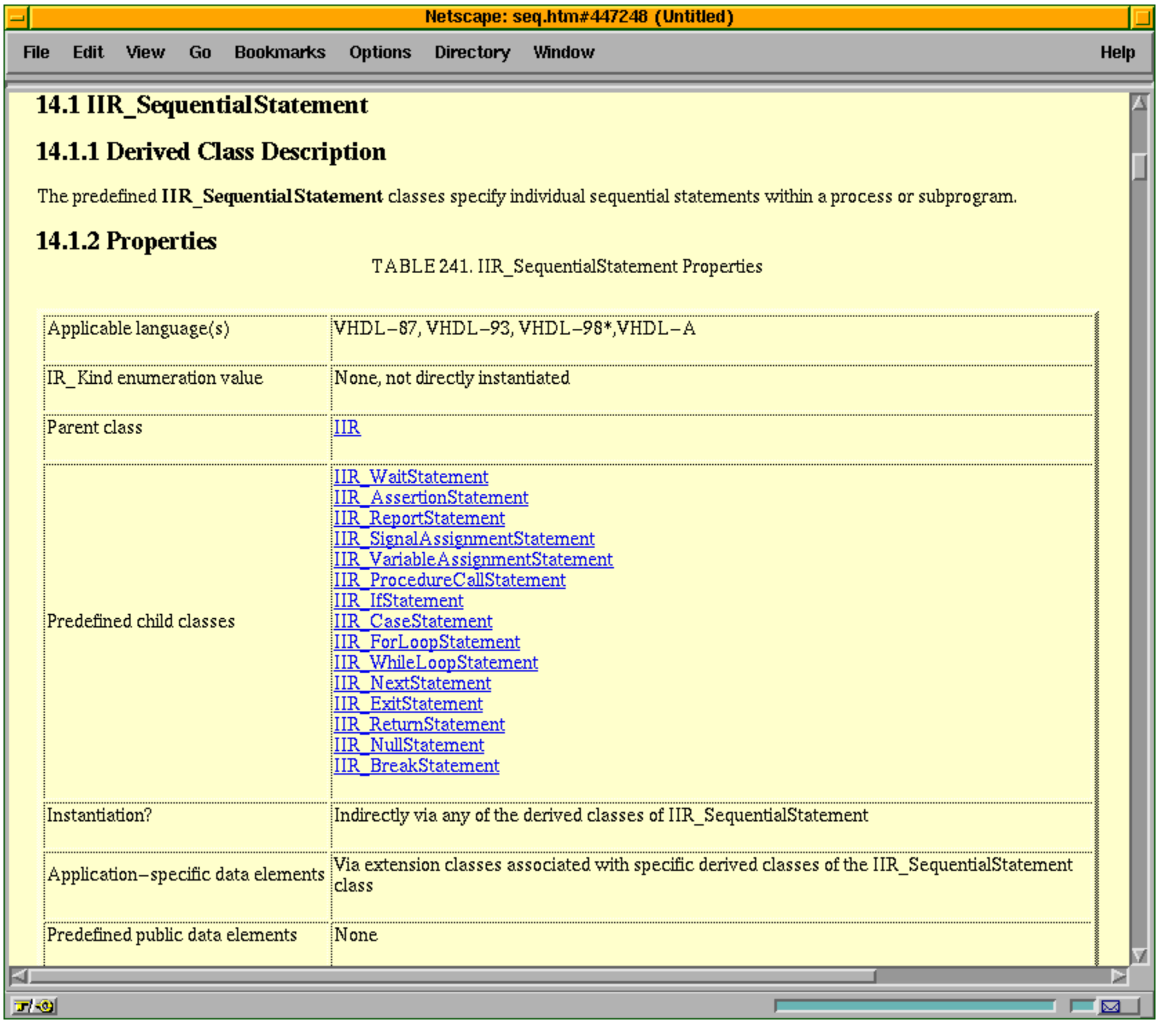
5.13 The IIR Class Hierarchy



5.14 Extensibility

- Each IIR class is really a collection of 2 or more classes
  - **IIRBase_X**: root class containing all predefined public methods
  - **IIRUser_X**: user class, void in base IIR definition
  - **IIR_X**: leaf class for this class (possible internal leaf) where **X** is a qualifier for a class in IIR
- Only leaf classes of non-interior classes are instantiatable
  (the derivation chain instantiates the respective parents)

5.15 Example from the HTML definition of IIR



14.1 IIR_SequentialStatement

14.1.1 Derived Class Description

The predefined **IIR_SequentialStatement** classes specify individual sequential statements within a process or subprogram.

14.1.2 Properties

TABLE 241. IIR_SequentialStatement Properties

Applicable language(s)	VHDL–87, VHDL–93, VHDL–98*,VHDL–A
IR_Kind enumeration value	None, not directly instantiated
Parent class	IIR
Predefined child classes	IIR_WaitStatement IIR_AssertionStatement IIR_ReportStatement IIR_SignalAssignmentStatement IIR_VariableAssignmentStatement IIR_ProcedureCallStatement IIR_IfStatement IIR_CaseStatement IIR_ForLoopStatement IIR_WhileLoopStatement IIR_NextStatement IIR_ExitStatement IIR_ReturnStatement IIR_NullStatement IIR_BreakStatement
Instantiation?	Indirectly via any of the derived classes of IIR_SequentialStatement
Application–specific data elements	Via extension classes associated with specific derived classes of the IIR_SequentialStatement class
Predefined public data elements	None

5.16 FIR Overview

- Object-oriented, extensible data structure
- Platform independent data formats to facilitate FIR library data exchange
- Design in progress, current version: 3.0
- Two implementations under development
  - Auriga: FTL Systems, Inc
  - SAVANT: University of Cincinnati
- Singlely rooted (at `class FIR`)

5.17 Chief Design Requirements for IIR

- Language Scope (coverage of 1976-87/93, 1976.1, 1364)

- Supportive of Multi-backend analysis tools
- Portable
- Extensibility
- Efficiency
- Support integration and product exchange
- Security (offer support for IP)
- Availability (non-proprietary, free or nearly free)

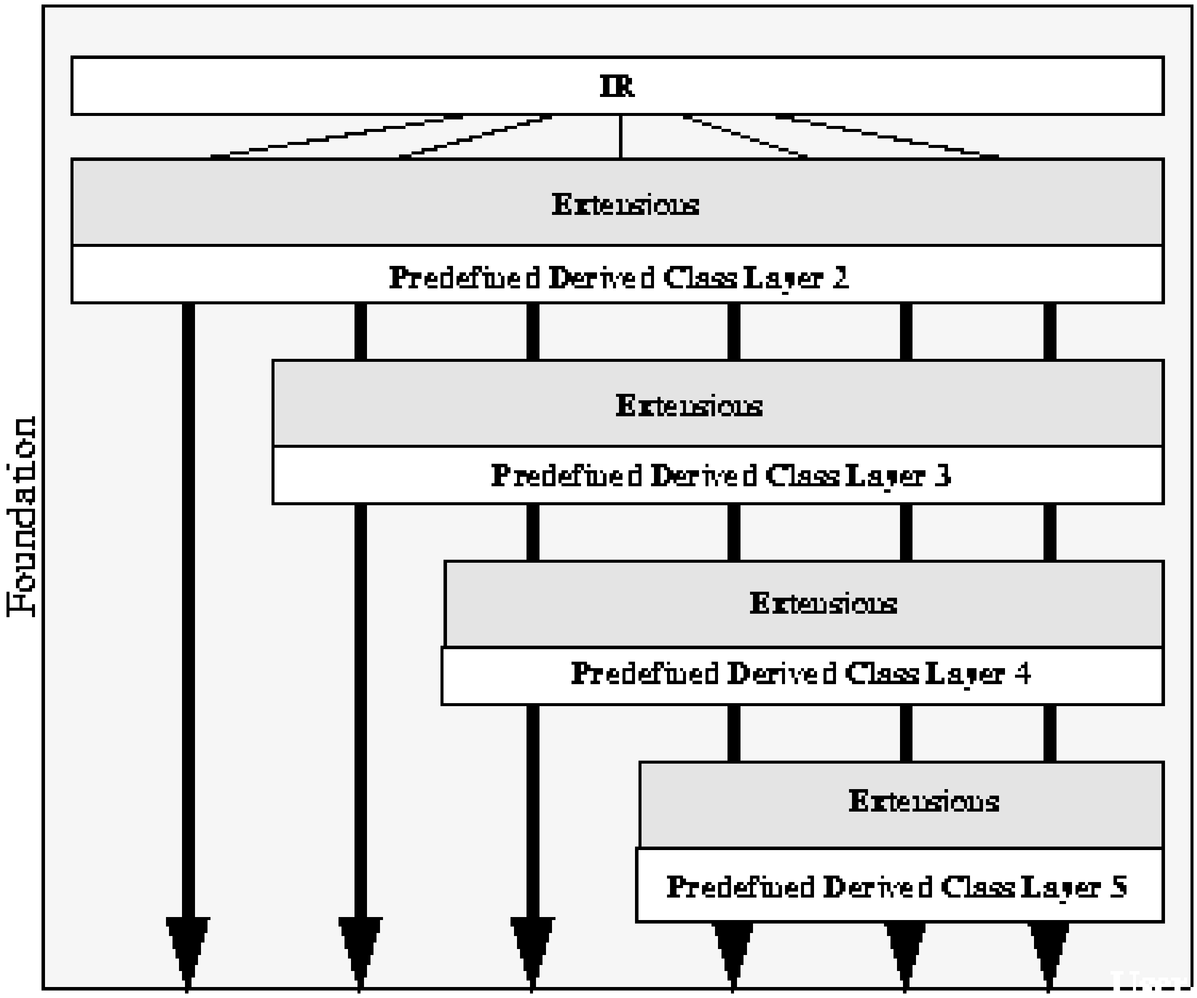
5.18 Real Object-Oriented Definition

- FIR is defined as an OO structure
- FIR supports use as an OO structure
- Design structure mirrors the class IIR definition
- Unlike IIR, FIR's public interface is a data structure definition

5.19 Example of the Derivation Hierarchy

- FIR
  - FIR_DesignFile
  - FIR_Comment
  - . . .
  - FIR_SequentialStatement
    - FIR_AssertionStatement
    - FIR_BreakStatement
    - . . .
  - FIR_ConcurrentStatement

5.20 The FIR Class Hierarchy



5.21 Example from the FIR HTML definition

Netscape: seq.htm#447248 (Untitled)

File Edit View Go Bookmarks Options Directory Window Help

15.1 FIR_SequentialStatement

15.1.1 Derived Class Description

The predefined **FIR_SequentialStatement** classes specify individual sequential statements within a process or subprogram.

15.1.2 Properties

TABLE 232. FIR_SequentialStatement Properties

Applicable language(s)	VHDL-87, VHDL-93, VHDL-98*,VHDL-A
IR_Kind enumeration value	None, not directly instantiated
Parent class	FIR
Predefined child classes	FIR_WaitStatement FIR_AssertionStatement FIR_ReportStatement FIR_SignalAssignmentStatement FIR_VariableAssignmentStatement FIR_ProcedureCallStatement FIR_IfStatement FIR_CaseStatement FIR_ForLoopStatement FIR_WhileLoopStatement FIR_NextStatement FIR_ExitStatement FIR_ReturnStatement FIR_NullStatement
Application-specific data elements	Via extension classes associated with specific derived classes of the FIR_SequentialStatement class

15.1.3 Predefined Data Elements

```
struct FIR_SequentialStatement {
        IR_Kind         kind;
        FIR_Source              source_locator;
        FIR_Ref         next; /* FIR_SequentialStatementList linkage */
        FIR_Ref         label; /* reference to an FIR_Label or FIR_Identifier */
};
```

Chapter 6

The SAVANT Implementation of AIRE

6.1 Goals for the SAVANT Implementation of AIRE

- Strictly adhere to the standard
- Attempt to enforce the intended usage
  - making non-final classes abstract to prevent their instantiation
  - using strong typing to enforce interfaces
- Encapsulate all core AIRE functionality
- Support user extensibility

6.2 Organization of the SAVANT Distribution

```
~/savant> ls
COMPONENTS             CVS/                   INSTALL
INSTALL-FOR-SIMULATION LGPL                   LICENSE
README                 RELEASENOTES           bin/
doc/                   lib/                   src/
~/savant>
```

6.3 savant/doc

```
~>ls savant/doc/
README            libraryManager/    programmers/   simulationManual/

~>
```

- overview.texi: overview of the `IIRScram` classes
- programmers.texi: programmer's guide

6.4 savant/src

```
~>ls savant/src/
total 945
 920 Makefile         1 TODO              2 savant.hh        2 version.hh
   6 Makefile.bak     1 aire/             1 scram/
   6 Makefile.in      5 main.cc           1 util/
~>
```

- location of main makefile
- `main.cc`: driver code for scram
- `version.hh`: specifies version/release date of distribution

6.5 savant/src/aire

```
~>ls savant/src/aire/
total 55
   9 IIR/                        7 IIRKind.hh
  18 IIRBase/                   18 IIRScram/
   2 IIRBasicDataTypes.hh
~>
```

- IIR/FIR common headers located here
- IIRBase: location of public AIRE/IIR methods
- IIRScram: location of scram specific data/methods
- IIR: leaf nodes for instantiation and derivation

6.6 IIRBase Contains

- Public interface described in the AIRE spec
- Protected/Private data/methods to support public interface

6.7 IIRScram Contains

- Methods for general use
  - `transmute()`
  - `publish_vhdl()`
  - `publish_cc()`
- Methods not for general use
  - type checking routines for SCRAM
  - symbol table management functions for SCRAM

6.8 IIR Contains

- empty constructor/destructors
- no other private/protected/public functions or data

6.9 savant/src/scram

```
~>ls savant/src/scram/
total 259
   6 FlexLexer.h            9 scram.cc
   2 README                 2 scram.hh
   2 RIGHTS                 2 scram.man
  22 VHDLLexer.flex         3 scram_func.hh
   3 VHDLLexer.hh           6 symbol_lookup.cc
   3 VHDLToken.hh           5 symbol_lookup.hh
   2 Xdefaults             10 symbol_table.cc
   2 bool.hh                6 symbol_table.hh
   2 config.hh            160 vhdl.gg
   3 declaration_chain.hh   9 vhdl.hh
~>
```

6.10 savant/src/util

```
~>ls savant/src/util/
total 71
   4 arg_parser.cc         4 error_func.cc          8 set.hh
   5 arg_parser.hh         4 error_func.hh          3 switch_file.cc
   2 container.hh          4 hash_table.hh          3 switch_file.hh
   7 dl_list.hh           20 resolution_func.cc
   2 elaborate_info.hh     5 resolution_func.hh
~>
```

- `arg_parser.{hh,cc}` parses command line arguments (discussed more fully tomorrow)

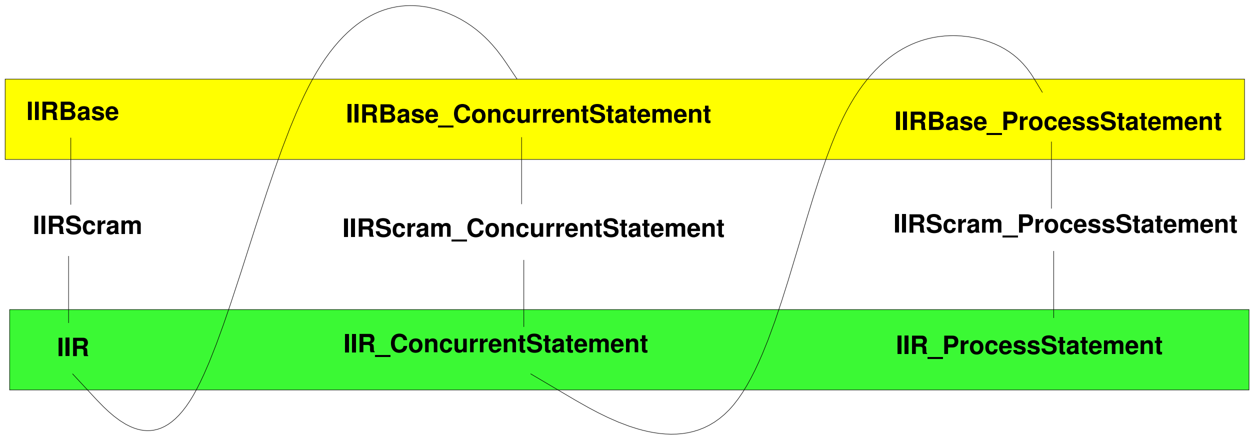
6.11 Programming Practices in SAVANT Development

- All SAVANT/AIRE class names directly correspond to the AIRE spec
- All AIRE specific data/methods reside in `IIRBase_`
- All SCRAM specific data/methods reside in `IIRScram_`
- Names of all extension methods have a leading underscore
- All `IIR_` classes are empty
- All C++ headers separate from bodies
- Fully descriptive names used whenever possible

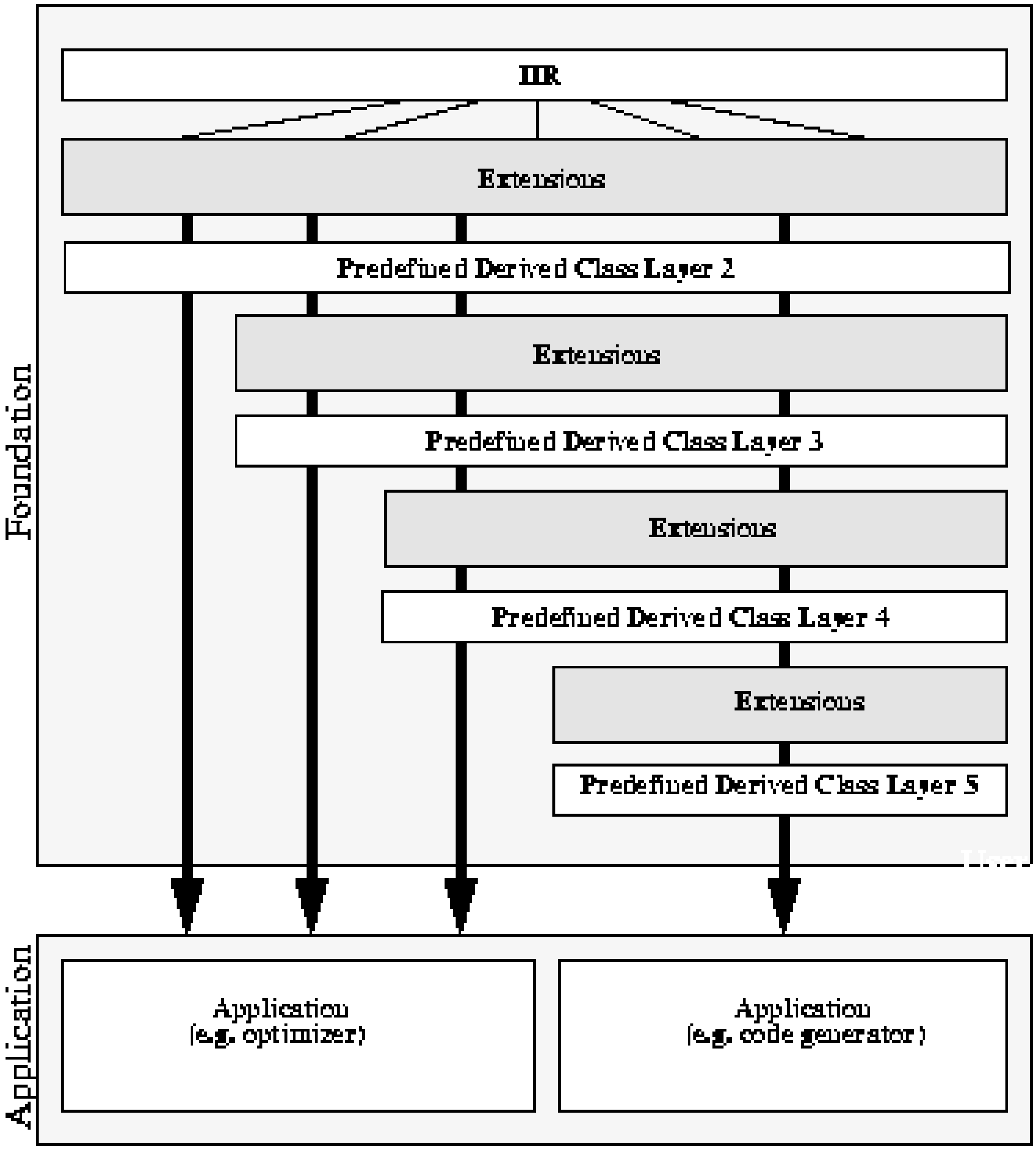
Chapter 7

Extensibility in SAVANT

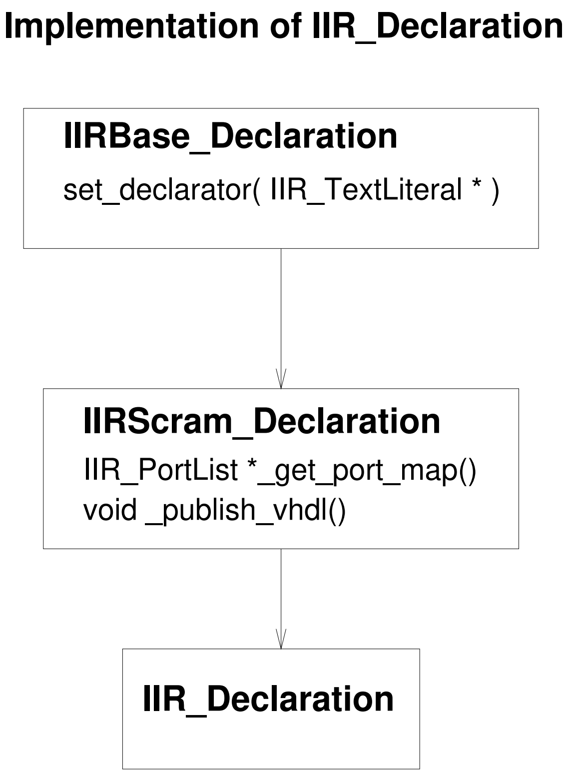
7.1 "Actual" Derivation of AIRE



7.2 Compare with the AIRE Spec



7.3 All AIRE Classes Occur as 3 SAVANT Class Definitions



7.4 Class Derivations Limited To....

- Abstract and concrete classes in `IIR`

7.5 Why this Structure?

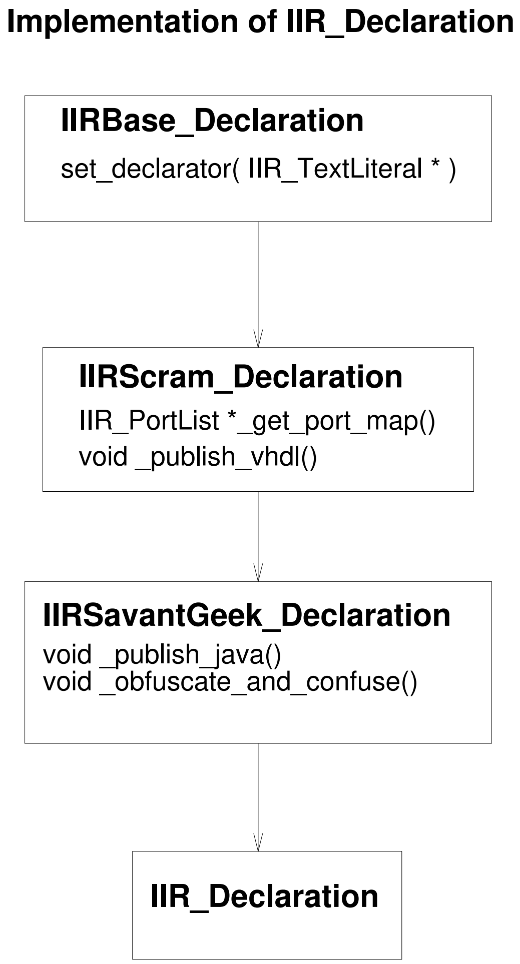
- SCRAM instantiates objects
- Instantiation causes invocation of parent constructors
- Virtual function tables are completed
- Enables extensibility w/o forcing modifications to code in the parser

7.6 Huh?

- SCRAM instantiates concrete classes in `IIR`
- The IIR tree is built by the SCRAM parser
- User defined extensions must then be inserted below the concrete, instantiatable nodes.

7.7 Extending `IIR_Declaration`

Classes can be added to the "micro" hierarchy to extend the functionality of the system without modifying the parser.



7.8 Code Changes for Extension

- build directory containing extension classes
- modify the makefile accordingly (tomorrow's lecture)
- change the derivation tree for the relevant classes in `IIR`
  - all the classes that are to be extended

Chapter 8

Review of SAVANT IIR Classes

8.1 Example Studied

In the remainder of this time period, we will trace the complete derivation tree from the root `class IIRBase` to `class IIR_SignalDeclaration`.

8.2 From IIRBase to IIR_SignalDedclaration: Exploring the SAVANT Class Hierarchy

8.2.1 IIRBase.hh

```
#ifndef IIRBASE_HH  // All header files are wrapped in this manner.
#define IIRBASE_HH  // Only shown here...

// standard copyright notice, warranty disclaimer, author identification, and

//---------------------------------------------------------------------------
//
// $Id: programmers.tex,v 1.6 1999/03/19 20:53:36 ramanan Exp $
//
//---------------------------------------------------------------------------

#include "IIRBasicDataTypes.hh"

class IIRBase {

public:

  virtual IIR_Kind get_kind() = 0;
  virtual IIR_Char *get_kind_text() = 0;

  void set_file_name(IIR_Identifier *file_name) { . . . }
  void set_sheet_name(IIR_Identifier *sheet_name) { . . . }
  void set_line_number(IIR_Int32 line_number);
  void set_column_offset(IIR_Int32 column_offset);
  void set_character_offset(IIR_Int32 character_offset);
  void set_x_coordinate(IIR_Int32 x_coordinate);
```

```
  void set_y_coordinate(IIR_Int32 y_coordinate);

  IIR_Identifier *get_file_name()  {return iir_file_name;}
  IIR_Identifier *get_sheet_name() {return iir_sheet_name;}

  IIR_Int32 get_line_number()      {return iir_line_number;}
  IIR_Int32 get_column_offset()    {return iir_column_offset;}
  IIR_Int32 get_character_offset() {return iir_character_offset;}
  IIR_Int32 get_x_coordinate()     {return iir_x_coordinate;}
  IIR_Int32 get_y_coordinate()     {return iir_y_coordinate;}

protected:

  IIRBase();
  virtual ~IIRBase() = 0;

private:

  IIR_Identifier *iir_file_name;
  IIR_Identifier *iir_sheet_name;

  IIR_Int32  iir_line_number;
  IIR_Int32  iir_column_offset;
  IIR_Int32  iir_character_offset;
  IIR_Int32  iir_x_coordinate;
  IIR_Int32  iir_y_coordinate;
};
```

8.2.2 IIRScram.hh

```
#ifndef DEVELOPER_ASSERTIONS
#include <assert.h>
#endif

extern switch_file _vhdl_out;  //file for vhdl output
extern switch_file _cc_out;    //file for c++ output

class IIRScram : public IIRBase {

public:

  virtual void _publish_vhdl();
  virtual void _publish_cc();
  virtual IIR_Boolean _is_declaration();
  virtual IIR_Boolean _is_ascending_range();
  virtual IIR_Boolean _is_name();
  virtual IIR_Boolean _is_signal();
  virtual IIR_Boolean _is_variable();
  virtual IIR_Boolean _is_text_literal();
  virtual IIR_Boolean _is_discrete_type();
```

```
  virtual IIR  *_transmute();

  //The following function is used for code generation that does runtime
  //elaboration
  virtual void _publish_cc_elaborate();

  void _report_undefined_scram_fn(char *);

  // These methods have to do with semantic checks and such.
  virtual set<IIR_Declaration> *_symbol_lookup();
  virtual set<IIR_Declaration> *_symbol_lookup( IIR_Declaration *);

  virtual set<IIR_TypeDefinition> *_get_lval_set();
  virtual set<IIR_TypeDefinition> *_get_rval_set();

  virtual void _type_check( set<IIR_TypeDefinition> * );
  virtual void _type_check( IIR_TypeDefinition * );

  virtual ostream &_print( ostream & );

  static void _set_symbol_table( symbol_table *s_t );
  static symbol_table *_get_symbol_table( );

protected:

  IIRScram();
  virtual ~IIRScram() = 0;
  static symbol_table *sym_tab;

private:

};

inline ostream &
operator<<(ostream &os, IIRScram &is ){
  return is._print( os );
}
```

8.2.3 IIR.hh

```
class IIR : public IIRScram {

public:

  virtual ~IIR() = 0;

protected:

  IIR();

private:
```

```
};
```

Note: this is an abstract class

8.2.4 IIRBase_Declaration.hh

```
class IIRBase_Declaration : public IIR {

public:

  IIR_Kind get_kind() {
    return IIR_DECLARATION;
  }

  IIR_Char *get_kind_text() {
    return "IIR_Declaration";
  }

  void set_declarator( IIR_TextLiteral *declarator );
  IIR_TextLiteral *get_declarator();

protected:

  IIRBase_Declaration();
  virtual ~IIRBase_Declaration() = 0;

private:

  IIR_TextLiteral *declarator;
};
```

8.2.5 IIRScram_Declaration.hh

```
class IIRScram_Declaration : public IIRBase_Declaration {

public:

  virtual void _publish_vhdl();

  // Methods for C++ code generation

  virtual IIR_Boolean _is_type() { return FALSE; }
  virtual IIR_Boolean _is_access_type() { return FALSE; }
  virtual IIR_Boolean _is_read_file() { return FALSE; }
  virtual IIR_Boolean _is_write_file() { return FALSE; }
  virtual IIR_Boolean _is_object_decl() { return FALSE; }
  virtual IIR_Boolean _is_signal() { return FALSE; }
  virtual IIR_PortList *_get_port_map(){ return NULL; }
  virtual IIR_GenericList *_get_generic_map(){ return NULL; }
```

```
  virtual IIR_Declaration *_find_symbol( IIR_Name *look_for ){
    _report_undefined_scram_fn("_find_symbol( IIR_Name *)");
    return NULL;
  }

  virtual char * _get_type_string(){
    return "(unknown)";
  }
  enum declaration_type { ERROR = 0, UNDEFINED, VARIABLE, SHARED_VARIABLE,
                          TYPE, SUBTYPE, SIGNAL, PROCEDURE, INTERFACE,
                          FUNCTION, S_FILE, ENTITY, CONSTANT, CONFIGURATION,
                          COMPONENT, ATTRIBUTE, ALIAS, ARCHITECTURE, PACKAGE,
                          PACKAGE_BODY, INTERFACE_VARIABLE,
                          INTERFACE_SIGNAL, INTERFACE_CONSTANT,
                          INTERFACE_FILE, LABEL, LITERAL, UNITS, GROUP,
                          LIBRARY, LAST_DECLARATION_TYPE };

  virtual declaration_type _get_type();

  void _set_scope( IIR_Int32 );
  IIR_Int32 _get_scope();
  virtual IIR_TypeDefinition *_get_lval();
  virtual IIR_TypeDefinition *_get_rval();

  virtual bool _check_param( IIR_TypeDefinition *decl, int arg_num );
  virtual bool _is_implicit_declaration();

  IIR_Attribute *_get_attribute_name();
  void _set_attribute_name( IIR_Attribute * );

  IIR_DeclarationList implicit_declarations;
  IIR_Declaration *_find_in_implicit_list( char * );

  virtual IIR_TypeDefinition *_get_type_of_element( int );
  virtual IIR_Int32 _get_num_indexes();

  virtual ostream & _print( ostream & );

protected:

  IIRScram_Declaration();
  virtual ~IIRScram_Declaration() = 0;

private:

  IIR_Int32 scope;
  IIR_Attribute *attribute_name;
};
```

8.2.6 IIRScram_Declaration.hh

```
class IIR_Declaration : public IIRScram_Declaration {

public:

protected:

  IIR_Declaration() {};
  virtual ~IIR_Declaration() = 0;

private:

};
```

8.2.7 IIRBase_ObjectDeclaration.hh

```
class IIRBase_ObjectDeclaration : public IIR_Declaration {

public:

  IIR_Kind get_kind() { return IIR_OBJECT_DECLARATION; }
  IIR_Char *get_kind_text() { return "IIR_ObjectDeclaration"; }

  void set_subtype(IIR_TypeDefinition* subtype);
  IIR_TypeDefinition* get_subtype();

  IIR_AttributeSpecificationList attributes;

protected:

  IIRBase_ObjectDeclaration();
  virtual ~IIRBase_ObjectDeclaration() = 0;

private:
  IIR_TypeDefinition* subtype;
};
```

8.2.8 IIRScram_ObjectDeclaration.hh

```
class IIRScram_ObjectDeclaration : public IIRBase_ObjectDeclaration {

public:

  IIR_TypeDefinition *_get_lval();
  IIR_TypeDefinition *_get_rval();
  set<IIR_TypeDefinition> *_get_rval_set();
```

```
  IIR_Boolean _is_object_decl();

  IIR_Int32 _get_num_indexes();
  IIR_TypeDefinition *_get_type_of_element( int index );

protected:

  IIRScram_ObjectDeclaration() {}
  virtual ~IIRScram_ObjectDeclaration() = 0;

private:
};
```

8.2.9 IIR_ObjectDeclaration.hh

```
class IIR_ObjectDeclaration : public IIRScram_ObjectDeclaration {

public:

protected:

  IIR_ObjectDeclaration() {};
  virtual ~IIR_ObjectDeclaration() = 0;

private:

};
```

8.2.10 IIRBase_SignalDeclaration.hh

```
class IIRBase_SignalDeclaration : public IIR_ObjectDeclaration {

public:

  IIR_Kind get_kind() {return IIR_SIGNAL_DECLARATION;}
  IIR_Char *get_kind_text() {return "IIR_SignalDeclaration";}

  void set_value(IIR* value);
  IIR* get_value();
  void set_signal_kind(IIR_SignalKind signal_kind);
  IIR_SignalKind get_signal_kind();

protected:

  IIRBase_SignalDeclaration();
  virtual ~IIRBase_SignalDeclaration() = 0;

private:
```

```
  IIR* value;
  IIR_SignalKind signal_kind;

};
```

8.2.11 IIRScram_SignalDeclaration.hh

```
class IIRScram_SignalDeclaration : public IIRBase_SignalDeclaration {

public:

  void _publish_cc();
  IIR_Boolean _is_signal(){ return TRUE; }

  declaration_type _get_type();

protected:

  IIRScram_SignalDeclaration() {}
  virtual ~IIRScram_SignalDeclaration() = 0;
};
```

8.2.12 IIR_SignalDeclaration.hh

```
class IIR_SignalDeclaration : public IIRScram_SignalDeclaration {

public:

  IIR_SignalDeclaration() {};
  virtual ~IIR_SignalDeclaration();

protected:

private:

};
```

Chapter 9

Programming Exercise 1

9.1 Problem Statement

1. Build a mechanism to profile the static structure of a VHDL model
2. Modify the of SAVANT classes in `IIRScram_`
   (a) Add the static variable `count` of type to the class
   (b) Modify the constructor to increment `count`
   (c) Modify the destructor to decrement `count`
3. Begin with `IIRScram_SignalDeclaration.hh`
4. Recompile scram
5. Run with supplied test file
6. Print the result
7. Compare the results to the original test file. Is the count correct?
8. Continue with additional VHDL constructs of your choice.

9.2 Detailed Programming Changes

9.2.1 Edit the file `IIRScram_SignalDeclaration.hh`

1. In the `private:` section, add:
   `static IIR_Int32 count;`
2. In the `public:` section, add:
   `static IIR_Int32 get_count(){ return count; };`
3. In the `protected:` section, change the line:
   `IIRScram_SignalDeclaration(){}`
   to
   `IIRScram_SignalDeclaration(){count++;}`
4. Save the file.

9.2.2 Edit the file `IIRScram_SignalDeclaration.cc`

1. Add the following line somewhere in the file:
   `IIR_Int32 IIRScram_SignalDeclaration::count;`

2. Change the line:
   `IIRScram_SignalDeclaration::~IIRScram_SignalDeclaration(){}`
   to
   `IIRScram_SignalDeclaration::~IIRScram_SignalDeclaration(){count--;}`

9.3 Recompile

1. move to the `savant/src/` directory

2. type: `make aire/IIRScram/IIRScram_SignalDeclaration.o`

3. Re-edit `IIRScram_SignalDeclaration.hh` and/or `IIRScram_SignalDeclaration.cc` until it compiles error free (ask an instructor for help if necessary).

9.4 Reporting the Results of the Static Counting

1. Edit the file `main.cc`

2. Just before the last }, add the lines:
   ```
   #include "IIR_SignalDeclaration.hh"
   cout << "Signal count:  " << IIR_SignalDeclaration::get_count() << endl;
   ```

3. Recompile

   (a) cd to the `savant/src/` directory

   (b) type `make`

   (c) If problems arise, contact an instructor for help.

9.5 Interesting results

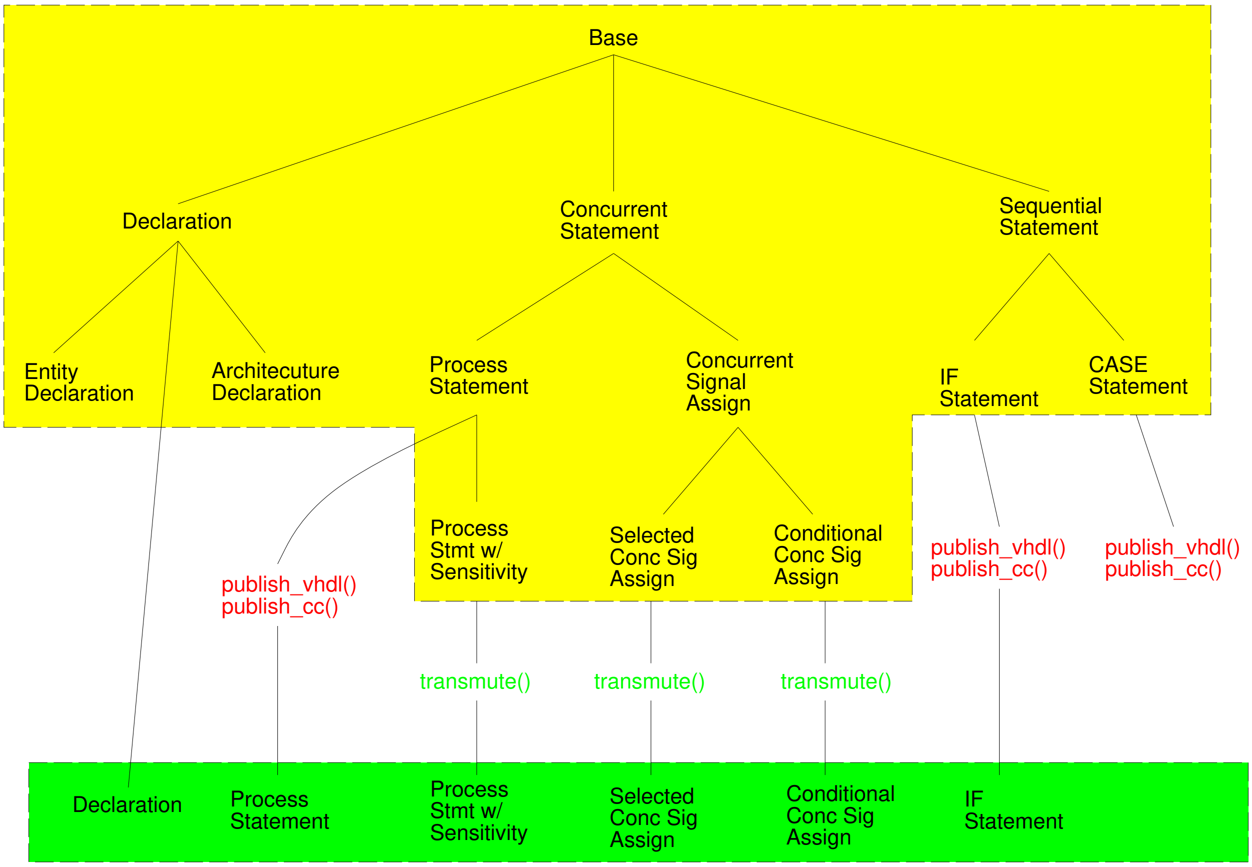
1. Now that we have a modified `scram` binary, we can test it on some vhdl files. After running the new `scram` on some simple files, count the number of signal declarations in the VHDL versus the number you program reports.

2. If the number the program reports is greater than the number you count, then count the number of signal valued attributes, and add that number to the number reported. As required by AIRE, `scram` internally creates declarations for various attributes to augment code generation in the back end.

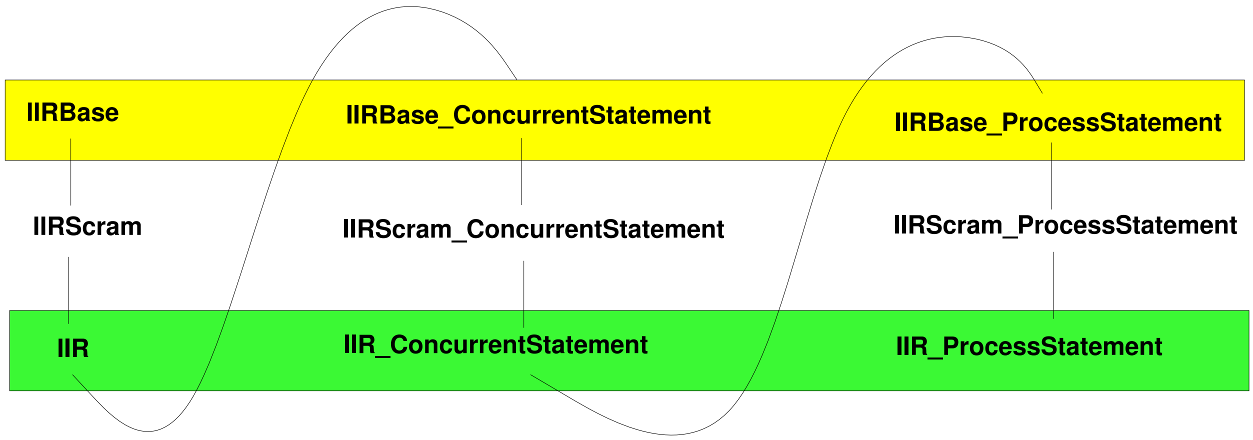
Chapter 10

Advanced Savant Programming

10.1 Recall the "Micro" Derivation of AIRE



10.2 Recall the "Actual" Derivation of AIRE



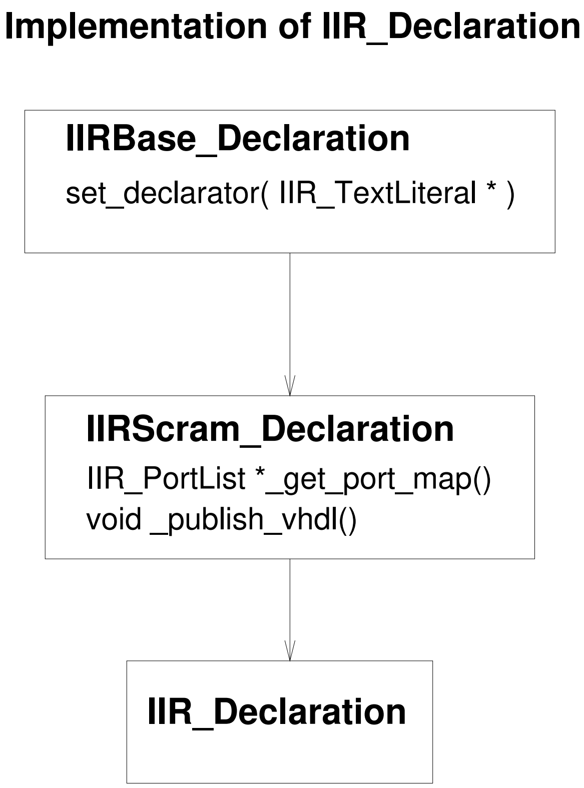
10.3 User Modifications

- Define a new class that shadows an existing method
- Add a new virtual function to the IIR root class (`IIR`) with a default action that is selectively shadow'ed on a class by class basis

- **Note:** code changes near the root of the IIR tree can trigger numerous dependency relations and create a situation where seemingly small changes cause recompilation of virtually the entire source tree.

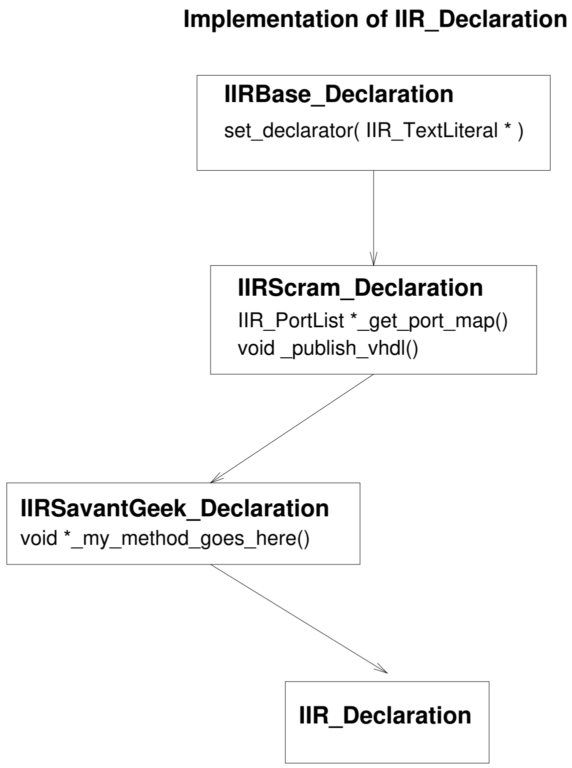
10.4 Inserting a User Defined Class into the Hierarchy (Part I)

As previously discussed, the default hierarchy of the SAVANT/AIRE nodes looks like:



10.5 Inserting a User Defined Class into the Hierarchy (Part II)

When inserting a user defined class into the hierarchy, the default derivation is “broken” to look like:



10.6 Inserting a User Defined Class into the Hierarchy (Part III)

Essentially, the user-defined class must be derived from the IIR level just “underneath” the IIR level, in this case, IIRScram. In addition, the derivation of the IIR level must be modified such that it inherits directly from the user defined class.

10.7 Preparations outside the AIRE Classes

- Updating the Makefile
- Modifying `main()` to invoke new methods
- Defining and Parsing new command line arguments

10.8 Editing `Makefile.in` for Inclusion of New Classes

When new classes are added to the SAVANT/AIRE hierarchy, the new classes must be compiled and included into the libraries. The file `Makefile.in` that comes with the distribution has a simple mechanism for adding classes to `libaire.a`. To include the classes in the subdirectory `IIRSavantGeeks` into `libaire.a`, find the following lines in `Makefile.in`:

```
# If you are extending the AIRE classes with your own, put your subdirectory
# in this list and it will automatically get compiled into libaire.
AIRE_SUBDIRS = $(AIRE) $(AIRE)/IIRBase $(AIRE)/IIRScram $(AIRE)/IIR
```

10.9 Editing `Makefile.in` (Part II)

Adding a subdirectory to this list has the following consequences:

- All ".cc" files are compiled into ".o" files.
- All of the ".o" files generated get written into `libaire.a`.
- The named directory is added to the list searched for C++ header files.
- `make clean` will remove the ".o" files in this directory.

Note: after editing `Makefile.in`, `configure` or `config.status` needs to be re-run (to generate `Makefile`), and dependencies need to be rebuilt with `make depend`.

10.10 Modifying "main"

The procedure `main` in `savant/src/main.cc` instantiates SAVANT's parser class, `scram`. `scram` essentially has one public method, `parse_files`. This method takes the following arguments:

int argc This variable tells the parser how many filenames are listed in the next argument.

char **argv This is an array of strings, holding the filenames to parse.

symbol_table * This is the symbol table you want to use for parsing.

The method `parse_files` returns a `dl_list<IIR_DesignFile> *`. Once this list is generated, the rest of the code in main my do whatever it likes with it. This is where methods in extension classes are normally invoked.

10.11 Adding Command Line Arguments to Scram

- When extending the SAVANT system with new capabilities, it will often be desirable to add command line options to scram to control them.
- In the `savant/src/util` subdirectory there is a class that implements an argument parser (in files `arg_parser.hh` and `arg_parser.cc`). This class is used by scram to process the command line, and can easily be extended to support additional options.

10.12 Argument Parser Initialization

In the file `savant/src/main.cc`, the following lines initialize the array of `arg_record`s passed to the arg_parser:

```
static arg_parser::arg_record arg_list[] = {
  {"-publish-vhdl","publish VHDL", &publish_vhdl, arg_parser::ARG_BOOLEAN},
  {"-publish-cc","publish c++", &publish_cc, arg_parser::ARG_BOOLEAN},
  {"-no-file-output", "send publish_cc output to stdout instead of compilable files",
    &no_file_output, arg_parser::ARG_BOOLEAN},
  {"-warranty-info", "print information about (lack of) warranty",
    &print_warranty, arg_parser::ARG_BOOLEAN },
  {NULL, NULL}
};
```

10.13 Syntax of arg_records (Part I)

Each arg_record consists of the following fields:

char *arg_text This is the text of the argument itself, like `--help`.

char *arg_help This is the description of the argument's function that is printed with usage messages.

void *storage This is a pointer to whatever type of storage this type of argument needs (as determined by the next parameter).

arg_type type This enumeration what type of argument this is - a boolean flag, or a string, for instance. See inline documentation for current options.

10.14 Syntax of arg_records (Part I)

To add a new argument to `scram`:

Modify arg_list Described above.

Declare new variable That the argument will effect. Don't forget to initialize the variable as well.

Chapter 11

Programming Exercise 2

11.1 Problem Statement

1. Build a mechanism to obfuscate identifiers in VHDL models
2. Make a `IIRTraining` directory in `savant/src/aire/iir`
3. Build the class:
   `IIRTraining_Identifier` derived from:
   `IIRScram_Identifier` that shadows the method:
   `void _publish_vhdl()` in `IIRScram_Identifier`
4. Obscure the printed identifier output by `_publish_vhdl()`
5. In the directory savant/src/aire/iir/IIR:
   (a) Modify the file `IIR_Identifier.hh` to include `IIRTraining_Identifier.hh` instead of `IIRScram_Identifier.hh`
   (b) Derive the class `IIR_Identifer` from `IIRTraining_Identifier` instead of `IIRScram_Identifier`
6. Recompile scram
7. Run against a VHDL file that contains an object of some type from package standard
8. What prints in the data type declarator?
9. How can this be fixed?
10. Continue with additional obfuscations of your choice.

11.2 Detailed Programming changes

1. Make a new directory for extension classes:

   `savant/src/aire/iir/IIRTraining`

2. Create new class:

   g `IIRTraining_Identifier`.

11.3 IIRTraining_Identifier.hh

```
#ifndef IIRTRAINING_IDENTIFIER_HH
#define IIRTRAINING_IDENTIFIER_HH

#include ``IIRScram_Identifier.hh''

class IIRTraining_Identifier : public IIRScram_Identifier {

public:
  _publish_vhdl();  // This is defined as virtual in base classes.
                     // This definition overrides.

protected:
  IIRTraining_Identifier() {};
  ~IIRTraining_Identifier() = 0;
}
#endif
```

11.4 IIRTraining_Identifier.cc

```
#include "IIRTraining_Identifier.hh"
#include "switch_file.hh"
#include <ctype.h>

IIRTraining_Identifier::~IIRTraining_Identifier() {}

void
IIRTraining_Identifier::_publish_vhdl(){
  int i;
  for( i = 0; i < length; i++ ){
    if( isalpha((text[i] + 1)) ){
      _vhdl_out << (char)(text[i] + 1);
    }
    else{
      _vhdl_out << (char)text[i];
    }
  }
}
```

11.5 Modify IIR_Identifier.hh in savant/src/aire/iir/IIR

```
#ifndef IIR_IDENTIFIER_HH
#define IIR_IDENTIFIER_HH

#include "IIRTraining_Identifier.hh"

class IIR_Identifier : public IIRTraining_Identifier {

public:

  IIR_Identifier() {};
```

```
  ~IIR_Identifier() {};

protected:  private:

};

#endif
```

11.6 Add IIRTraining to Makefile

1. Modify `Makefile.in` to include the files from `IIRTraining` into `libaire.a`. Look for the following lines in `Makefile.in`:

```
# If you are extending the AIRE classes with your own, put your subdirectory
# in this list and it will automatically get compiled into libaire.
AIRE_SUBDIRS = $(AIRE) $(IIR)/IIRBase $(IIR)/IIRScram $(IIR)/IIR
```

2. Change it to read:

```
# If you are extending the AIRE classes with your own, put your subdirectory
# in this list and it will automatically get compiled into libaire.
AIRE_SUBDIRS = $(AIRE) $(IIR)/IIRBase $(IIR)/IIRScram $(IIR)/IIR $(IIR)/IIRTraining
```

11.7 Build new Makefile and Compile

1. Build a new Makefile from Makefile.in by re-running the script `savant/configure/configure`.
2. Build dependencies in the new `Makefile` using the command `make depend`.
3. Build `scram` by issuing the command `make`.

NOTE: You can make sure that your new class compiles before rebuilding the entire system by:
`make aire/iir/IIRTraining/IIRTraining_Identifier.o`
Additionally, you can build `libaire.a` to see if you modified the `Makefile` correctly:
`make ../lib/libaire.a`

Acknowledgments

This research has been conducted with the participation of many investigators. While not an complete list, the following individuals have made notable direct and/or indirect contributions to this effort (in alphabetical order): Jeff Carter, Harold Carter, Praveen Chawla, John Hines, Herb Hirsch, Timothy J. McBrayer, Greg Peterson, Al Scarpelli, Mike Shellhause and John Willis. We wish to express our sincerest gratitude for the time that you spent reviewing and commenting on our designs.

This research was supported in part by MTL Systems, Inc and the Wright Laboratory at Wright-Patterson AFB. Thank you for your support.