Intermediate Format Standardization: Ambiguities, Deficiencies, Portability issues, Documentation and Improvements

Amir Masoud Gharehbaghi, Mohammad Hossein Reshadi, *Zainalabedin Navabi
Electrical and Computer Engineering Department
Faculty of Engineering / University of Tehran / Tehran, Iran
{amir, reshadi}@cad.ece.ut.ac.ir
*Northeastern University / Boston, MA 02115
Tel: 617-373-3034; Fax: 617-373-8970
navabi@ece.neu.edu

Abstract

The growth in VLSI technology demands more capabilities from CAD tools. This requires better-integrated environments and more design portability across platforms and tools. The move towards hardware description languages in recent years and the issue of intellectual property and interoperability puts a pressure on EDA vendors to come up with a well-defined standard intermediate format. The draft AIRE/CE intermediate format, as distributed publicly on the Web, is one such standard. Although this standard may be better defined and better documented than any other proposed standards for this purpose, it has shortcomings that must be resolved before it is adapted by the EDA vendors and designers. This paper focuses on the AIRE weaknesses and presents our solutions according to our experiences with the AIRE implementation[1].

Keywords: Object-Oriented Intermediate Format, AIRE/CE, IIR, FIR, HDL, CAD Tools.

1. Introduction

One of the most important problems for integrating CAD tools is sharing of the design information between various tools environment. Lack of a standard intermediate format makes interfacing between tools very difficult, time consuming, and expensive. Exchanging design information, design reuse, and intellectual property commerce are important motivations towards developing a standard intermediate format.

Over the past decade, many efforts have been done to develop a suitable IF for hardware description languages. Researches from governmental, academic and EDA vendors have been working on developing an IF. The main effort for standardization of an IF was by the IEEE DASC working group, which was unsuccesful in reaching its goals.

Other efforts led to the AIRE/CE (Advanced Intermediate Representation with Extensibility / Common Environment), representation which is an object-oriented IF designed to support VHDL, VHDL-AMS, Verilog, and other languages. Worked into AIRE is extensibility capability, which makes this standard adaptable to new applications. The memory representation (Internal Intermediate Representation, IIR) of AIRE has five layers of hierarchy. Its file counterpart (File Intermediate Representation, FIR) stores the compiled model in the FIR files.

Although designers of AIRE have made their best to develop a general IF, there are serious problems and ambiguities in the AIRE structure. We will address some of these issues here and present solutions we reached in our implementaion of AIRE.

Section 2 of this paper presents problems with AIRE in specifiaction, documentaion and implementation. In sectio 3, we present solutions to these problems evolved as we implemented this standard. In section 4 recommendations and conclusions will be metioned.

[1] The version of AIRE/CE being discussed in this paper is the version that is publicly circulated on the Web [1], and several companies commercially use versions if AIRE/CE which may differ from the publicly available draft and which may address issues raised in this paper. AIRE is a trademark of FTL Systems.

2. Problem Issues

A major problem in implementing AIRE is in canonical and non-canonical categorization of objects and handling them differently. Other problems with AIRE specification are ambiguities in language analysis and documentation. The following subsections discuss these problems in more details.

2.1. Ambiguities

The AIRE standard has two major ambiguous areas in its structure and in language mapping.

Although AIRE is claimed to cover Verilog, ANSI C and VHDL-AMS as well as VHDL, its structure is far more VHDL oriented than any other language that it claims to support. In other words before mapping any other language to AIRE, it should be translated to VHDL. Even for this goal, it is not very clear how to convert some language constructs of VHDL to AIRE. For example, there are two methods to define multidimensional arrays in VHDL but only one of them is supported in AIRE.

AIRE ambiguities are also in the area of language mapping. Processing a source code is much more difficult than processing a compiled binary format. Intermediate formats are introduced to ease this processing. To do this, there should be a clear mapping between a language grammar and intermediate constructs. In many cases, types of the member variables in AIRE classes help to identify the correct mapping. However, in some cases there is either no clear or more than one possible way to do this conversion. Consider, for example, the Guard implicit signal. Whether a new guard signal is assigned to every block statement guard expression or guard expressions of nested blocks are converged, is not clear from the standard.

Although both alternatives may work, this and other ambiguities affect design portability. Designs are not portable between different implementations of AIRE if they use different conversion techniques.

2.2. Deficiencies

A critical deficiency in the AIRE standard is the concept of canonical objects. In currently released standard of AIRE, objects are divided into two major groups of non-canonical and canonical. A canonical object has many references whereas non-canonical objects are referenced by only one other object. Non-canonical objects are created and destroyed using *new* and *delete* operators respectively. There may be many instances of similar objects of this type keeping track of all the references is an easy problem. On the other hand, Canonical objects are created and destroyed using *get* and *release* methods respectively. Every object of this type will only have one instance in the memory and a reference-count keeps track of the number of references to the object. Calling *get* creates the object in the first call and increases reference-count in next calls. Calling *release* will decrease the reference-count and will delete the object if its reference-count is zero.

Many classes in AIRE have member variables of *IIR** type. In fact, these variables can be of any type because *IIR* is the predecessor of all other classes in the AIRE hierarchy. When an object referencing another object of the *IIR** type is being destroyed, the kind of the referenced object should be exactly known. If an object is canonical, *release* should be used; otherwise, *delete* is the proper destructor. This information is not easily extractable from the *IIR** type, and requires much processing.

2.3. Portability

Portability is one of the most important features of a good intermediate format. To be portable, compiled models should be AIRE implementation independent. Although FIR can be very effective in making AIRE compliant model portable, it has several major problems in this area. The following lists some of these portability issues.

The structure of FIR files is not defined as part of the AIRE standard. FIR V.3 that dates back to 1996 is the only available FIR and that is not a suitable intermediate format for HDL based CAD tools.

Filing hierarchy of IIR objects need to be clearly defined in a standard IF. The grouping of IIR objects for a single FIR file is not known.

Another problem with FIR is lack of a method for cross-referencing between objects of different FIR files. The current mechanism of using *FIR_ProxyRef* and *FIR_ProxyIndicator* is inefficient for referencing in the same file and inadequate for referencing objects outside a file. In addition, there is no clear mechanism for addressing other files and satisfying portability of source files. Since every design refers to languages primitives in its lower levels of design hierarchy, a provision in FIR should be made to support this

Language primitives that are not expressed in AIRE and should be tagged as primitives are not classified as such with a standard scheme. This classification of objects is necessary in FIR and IIR classes. When porting a compiled design, the set of files to be ported is not clear. For example, every implementation of AIRE has the standard packages, and designs using such packages should not require porting them.

In addition to FIR problems, language mapping ambiguities discussed earlier also affect portability of compiled designs. For example, the standard does not clearly define the order of suffixes and prefixes in *IIR_Name* subclasses.

2.4. Documentation and Availability

Documentation plays a major role in success of a standard. Systematic upgrade and review as well as regular updates are essential for documentation of a standard such as AIRE. Unfortunately, AIRE Version 4.6 document of early 1998 is the most recent document on AIRE. In addition to weaknesses and ambiguities, it has many errors that are left uncorrected. Of course we believe that the lack of an implementation (or a good one) is the cause of many of these documentation (or perhaps design) problems.

The following is just a list of a few documentation deficiencies in AIRE.

- Several classes are missing from the document for example *IIR_ChoiceByOthers* and *IIR_ChoiceByExpression* that are required for expressing *case* choices.
- The parameter type of *IIR_Label:: set_statement* does not allow all possible cases to be covered. This includes labels for concurrent statements.
- The document does not discuss the *IIR_Signature* as part of the *IIR_Attribute*.
- The *suffix* type in successors of *IIR_Attribute* as specified in the document does not cover all possible types allowed in the grammar.
- Although the *IIR_ScalarTypeDefinition* class has a direction member variable, the document does not define a corresponding type for it.

Extension classes are another important part of AIRE that need to be simplified. The process of inserting extensions between AIRE classes is very hard as presently defined. With the present level of difficulty, it will be a time consuming process if a set of existing extensions is to be ported from one AIRE implementation to another. More specific style definition for extension classes such as allowing and defining macros are more effective from programmers' point of view.

2.5. Security and Intellectual Property

The present form of AIRE does not provide a means of securing the transfer of compiled designs from IP developers to their customers. Based on the requirements of customers, the provider should be able to control the access of details in a design. Currently there is no specific method in AIRE to achieve this important property. Although AIRE document claims some sort of IP protection, no specific details for achieving this is proposed. In fact, security issues are in direct contradiction with portability, and more intensive work is needed to cover both of these issues.

3. Improvements

The problems discussed so far are just a few of what we faced in the process of developing a VHDL analyzer for producing AIRE classes. In this section, we present our suggestions to improve the AIRE structure as a standard IF. Applicability of our suggested improvements is verified in our AIRE implementation.

3.1. Ambiguity

To solve the ambiguity issues, classes like *IIR_ChoiceByOthers* and *IIR_ChoiceByExpression* were added to AIRE. *IIR_LibraryDeclaration* was changed to overcome the incompatibility of AIRE classes with the VHDL grammar. In addition, a one-to-one mapping between VHDL grammar and AIRE classes has been clearly defined.

There are many instances of *IIR_TextLiteral** in AIRE classes that should be replaced with a proper *IIR** or *IIR_Declaration** variable. This requires the use of the *lookup* method of *IIR_Name* class. However, to make a valid lookup, one needs to know the visibility scope of identifiers. This information is not available after code generation. Alternative solutions are either to extract visibility scope information from the generated code or to provide links from identifiers to their declaration in the generated code. The former solution degrades performance of applications based on AIRE. However, the latter is preferred and it should be done in code-generation elaboration-phase. The results should be stored in predefined well-known variables in the AIRE classes that have an identifier to reference other classes. For example, *prefix* in *IIR_SimpleName* is a good candidate for storing the pointer to an *IIR_Declaration* corresponding to the identifier of *IIR_Name*.

3.2. Deficiencies

Canonical and non-canonical objects cause problem at destruction time because of their different mechanisms of creation and destruction. In an object-oriented design with facilities such as inheritance and virtual methods, using conventional inefficient methods such as detecting the destruction mechanism based on type of the object are not recommended.

To correct this problem, we developed a uniform mechanism for both types of objects. This way, there is no need to detect the object type at the destruction time. This uniform mechanism also provides an easy way of copying an object. A copy of an object is needed in many instances such as default value expression in signal and variable declaration, elaboration of generate statements and component instantiation statement, and guard signal handling in block statements.

This copy can be a duplication of the original object or it can just share the pointer of the original object. In our solution, the copy of an object is created based on its type, i.e. sharing pointers for canonical objects and duplication for non-canonical objects. Besides being an easier implementation, it does not present any overhead to the original AIRE.

3.3. Portability

Almost all the existing portability problems of AIRE are due to lack of a good FIR structure. We have completely revised FIR with a new structure that properly handles portability issues. In this new design, each directory is equivalent to a VHDL library. Library units are stored as separate FIR files in the directory of their corresponding library. Format and general concepts of our FIR design will be presented.

```
[Header]
basic_data_type=filename
version= version number
[1]
kind = IR_ENTITY_DECLARATION
...
identifier = [3]
...
architectures = [12]

[2]
kind = IR_IDENTIFIER
...
text = "Mux2x1"
length = 6

[12]
kind = IR_LIBRARY_UNIT_LIST
no_of_elements = 1
elements ={ARC_Behavioral_Mux2x1.FIR;1;[behavioral]}
```

Figure 1. General Layout of FIR Text

FIR files in this design can be in binary and text formats. Text format is useful for debugging purposes and academic use, while the binary format is more compact and secure. Figure 1 shows a general outline of our proposed FIR text format.

There is only one basic-data-type for all designs. The file corresponding to this holds information about the number of basic data types and their size and byte alignment. Figure 2 shows a partial outline of this file.

```
ByteAlignment,
FileSystem,
OSName,
NoOfTypeInfos,
TypeInfoSize,
TypeName,
TypeSize,
…
```

Figure 2. Basic Data Types

An FIR file contains three sections: header, body, and footer. The header contains information like reference to the basic-data-types file, version number, coding information, company id, etc. The FIR body includes objects denoting the design information. Objects employ two different mechanisms for referencing internal objects and external objects. FIR footer section consists of a CRC number over all bytes of a file.

Header: FIR type, header size, body size, basic_data_types filename, version, source language, reserved for security information
Body: Objects of single IRR_Entity_Declaration
Footer: checksum

Figure 3. FIR File Structure

There are two kinds of object referencing. Local Object Referencing (LOR) points to other objects in the same FIR file, while Global Object Referencing (GOR) points to objects in other FIR files. *FIR_ProxyRef* and *FIR_ProxyIndicator* of the present FIR definition do not present a slotion for GOR and are inefficient for LOR. We are introducing a new mechanism to support both LOR and GOR in our text and binary forms of FIR.

LOR is simply done by unique IDs given to objects of a FIR file. In a Library Unit, every object that points to another object uses its ID in the FIR to maintain its connectivity. The saving mechanism promises the uniqueness of IDs in a file. GOR is more complicated. It is done through filenames, version and a sequence of names in the file. The version confirms that the file is not compiled lately and that this part of the design is still valid. The names will provide an ideal way to address and object by its name. The sequence of names is needed to solve the problems of visibility and scope that is not available (and is not needed to be) after compilation. This very important property enables the designs to be ported easily even if they have many references to the standard packages. The loader should use every name in the sequence in the revised version of *Lookup* to achieve the exact pointer of the desired object. Every time a *Lookup* fails, the Library Unit addressed by the filename must be loaded.

This design has a good mechanism for implementing a standard package as a FIR file. Language predefined structures are also supported.

The mapping portability problem mentioned in section 2.3 is resolved by defining a clear correspondence between AIRE classes and the VHDL grammar.

File addressing needs to be location independent as much as possible. This guarantees the portability of the design. Besides, every implementation and also extension for AIRE may need to handle language primitives (specially operators) differently. Some primitives cannot be represented in AIRE. For example, implementation of operators is application and implementation dependent parts of design that have not been solved in FIR.

Implementing some of very primitive parts of standard packages in the AIRE implementation and loading them in the memory before any other object is the basis of our solution.

3.4. Documentation and Availability

Documentation of a standard must be clear, error free and publicly available. An organization should assume responsibility for updating and publicizing relevant documents. Over the last ten months, AIRE has been suffering the lack of such an organization. Our corrections and deviations from AIRE (V 4.6) are now documented and can be made available upon request.

3.5. Security and IP

To protect their intellectual property, designers prefer to represent their design in fully compiled formats with very limited access to details. AIRE developers claim to have solved this problem by providing some coding techniques, and that a designer can decide on the level of access to design information that he or she wants to make available.

The IIR and FIR structures in AIRE are public information. Although it is possible to employ techniques to fully secure the FIR files from unauthorized access, securing the IIR format is very difficult. On the other hand, when a design is loaded in the memory, hackers can eventually break any secure coding. Secure access to design information is in direct contradiction with portability. In our implementation, we have not worked on security measures.

4. Conclusion and future work

In spite of its shortcomings, AIRE is an intermediate with a detailed document that is publicly available. There is a limited informal support for this IF, which in many ways is better than that of other intermediate formats. The object-oriented design of AIRE is a feature of this IF that is compatible with modern programming styles. A standard intermediate format is essential for today’s design environments. AIRE has come a long way towards achieving this goal and a concentrated effort for taking the last few remaining steps is needed.

References

1. AIRE document Version 4.6 at http://www.eda.org/aire/
2. J.C. Willis, G.D. Peterson, S.L. Gregor, *“The Advanced Intermediate Representation with Extensibility / Common Environment (AIRE/CE)“*, IEEE Transaction on Computer, 1998.
3. IEEE standard VHDL Language Reference Manual, IEEE std. 1076, 1993, The Institute of Electrical and Electronic Engineers, New York, NY.
4. Draft IEEE standard Verilog Language Reference Manual, IEEE std. 1364, 1995, The Institute of Electrical and Electronic Engineers, New York, NY.
5. M. H. Reshadi, A. M. Gharehbaghi, “*VHDL Intermediate Format Representation*”, CAD Lab. Report 23, University of Tehran, August 1999.[2]

[2] Documentation will be made available electronically.