Case study: An infrastructure for C/ATLAS environments with object-oriented design and XML representation

Cheng-Wei Chen, Jenq Kuen Lee *

Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan

Abstract

In this paper, we present an ATLAS compiler environment used for automatic testing as a case study to demonstrate the design of the state of the art compiler environments with object-oriented designs and XML representations. ATLAS is a testing language which is applied on the automatic test equipments (ATE). Currently, the programming language is used in the fields of avionics, industry facilities, and precision transport system. In this work, we develop the ATLAS compiler aiming to provide the control ability of the PC-based ATEs. First, it comes with an objected-oriented representation of program trees. The object-oriented program graph allows the flexibility for the manipulations of program trees. Second, it employs the object serialization technology for storing and retrieving the syntax trees and program graphs. The employment of object serialization techniques significantly reduces the programming effort from traditional compiler work in retrieving binary representations of program graphs from secondary storages. In addition, we establish the connection between the object-oriented program graph and XML representations. With the support of DTD and XSL files of XML environments, we can perform machine validation on XML-based representations, transform XML representations into graph structures, and annotate the representations for human browsing. Our software infrastructure can be used for subsequent controls and specifications for ATEs.

Key words: ATLAS language, compiler design, object-oriented infrastructures, automatic test equipment, XML

* This paper is submitted to Journal of System and Software. This is the revised version.
Corresponding author. Tel.: +886-3-5715131 EXT. 3519; fax: +886-3-5723694.
Email addresses: cwchen@pllab.cs.nthu.edu.tw (Cheng-Wei Chen), jklee@cs.nthu.edu.tw (Jenq Kuen Lee).

1 Introduction

ATLAS is an IEEE Standard Test Language for All Systems (IEEE, 1998, 1995). This language is designed to describe tests in terms that are independent of any specific test system, and has been constrained to ensure that it can be implemented on ATE. Since the 1960's through different phases of evolutions, it has been in use for field maintenance of avionics for the military and for commercial airlines. Initially a test description language, it has been automated to provide for automatic test programs. It is a large language, and the development of compilers for it is not a simple task.

The ATE can be applied to overhaul the failure facilities on the manufacture and the maintenance of variant precision instruments. In the process, it substantially improves the reliability of precision instruments. By using the ATE, the failed points of the failure facilities can be precisely located. Furthermore, they can be fixed in module level instead of replacement of entire systems. Therefore, the costs of the maintenance can be substantially decreased. The ATLAS language is widely used on the testing process of the ATE. Figure 1 illustrates the key components in the ATLAS testing model. The ATLAS programs are written to provide the entire test steps, the requirements of the switches and connections, the requests of the resource, and the statistic information. The intermediate form generated in the compiler environment is then used to generate subsequent controls to be integrated with ATEs.

In this paper, we developed the ATLAS compiler to aim at providing the control ability of the PC-based ATEs. To achieve this goal, we developed the compiler with the full implementation of IEEE standard. Our work is also a joint work between our programming language laboratory (Chang et al., 2001; Hwang et al., 1998, 2001), Tsing-Hua Univ., Taiwan, and a local industrial research institute, aerospace division, III (Institute for Information Industry), in promoting academic compiler technologies for industrial applications. Once the intermediate form is generated by our compiler environment, research engineers at III then use this information to generate controls to work with ATEs as shown in Figure 1. Our work can also serve as a case study to demonstrate the design of the state of the art compiler environments with object-oriented designs and XML representations. Our compiler is implemented using object-oriented support. First, abstract syntax trees are treated as linked objects. Linguistic constructs are defined in a class hierarchy. Object-oriented frameworks (Chang et al., 1997; Gannon and Lee, 1992; Lee et al., 1997; Wilson et al., 1994; Wu and Lee, 2000) for compiler infrastructures have gotten momentum recently. Our framework employs object hierarchies to represent program graphs of ATLAS languages. It's also the first object-oriented design for ATLAS compiler frameworks, and we think our experiences in the ATLAS software environments should offer software experiences and interesting engineering aspects for this problem. Second, in our design, we employ the object serialization technology

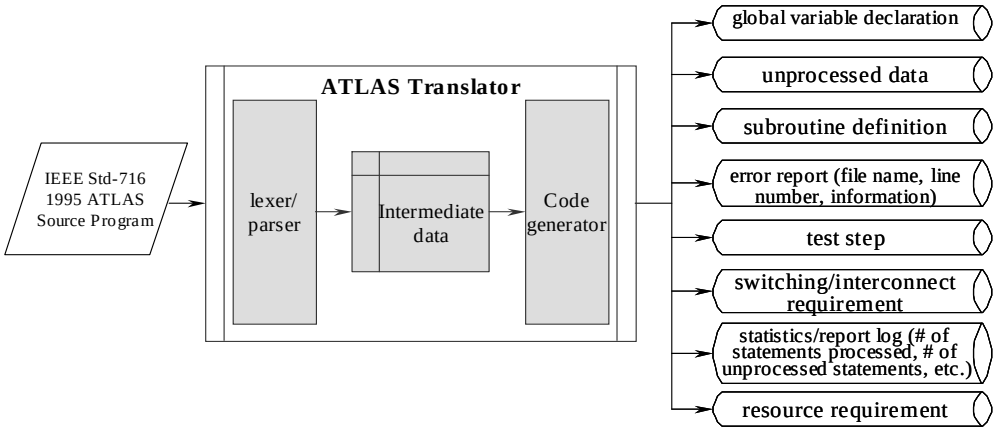

Fig. 1. Key components in the ATLAS testing model.

for storing and retrieving syntax trees and program graphs. The employment of object serialization techniques significantly reduces the programming effort from traditional compiler work. Traditionally, in retrieving binary representations of program graphs from secondary storages, one needs to carefully pack binary number and byte orders. This is normally an error-prone work. Our work is possible due to our employment of object-oriented representations of compiler intermediate forms, and our employment of emerging serialization methodologies in OO community.

Finally, we establish the connection between the object-oriented program graph and XML representations. With the support of DTD and XSL files of XML environments, we can perform machine validation on XML-based representations, transform XML representations into graph structures, and annotate the representations for human browsing. This connection provides great flexibilities for ATLAS program trees to be operated under XML tools. We also report experimental results for our system. Our work presents the first-hand software experiences for the state of the art design and implementation of a complex application domain language with object-oriented designs and XML representations.

The remainder of this paper is organized as follows. Section 2 gives a brief description of the ATLAS language. In Section 3, we present the development process of the ATLAS compiler. Section 4 then describes the XML representations for program graphs. Next, Section 5 gives our experimental results. Finally, Section 6 concludes this work.

2 The ATLAS language and environment

2.1 Overview of ATLAS language

The increasing complexity of systems being developed in the 1960s placed considerable demands on testing processes. A great deal of testing needs to be done, not only in development but also as part of subsequence maintenance. The ATLAS language arose from the perceived need to improve the precision and efficiency of the process.

To respond the requests of the commercial airlines, Aeronautical Radio Incorporated (ARINC) started the development of a standard testing language. There are a number of common test procedures to test and repair similar or identical avionics systems on the commercial aircraft, so developing test procedures in a standardized and unambiguous way enable their efficient reuse across airlines. The name of the language developed under the auspices of ARINC was the Abbreviated Test Language for Avionics Systems or ATLAS. There were a large number of commercial companies interested in avionics testing, and they participated in developing the ATLAS language. In 1976, standardization controls for ATLAS were transferred from ARINC to the IEEE. At this time the name of ATLAS became the Abbreviated Test Language for All Systems to reflect the wide-ranging nature of applications.

The ATLAS language standard was first published in 1968 and was titled ARINC 416-1. In 1988 the IEEE published ATLAS standard 716-1988, which represented a subset of the ATLAS 416. In 1984 the IEEE also published standard 771. This standard is a guide to the use of the ATLAS language. The up-to-date standard was IEEE Std 716-1995.

An example of the ATLAS test program is shown in Figure 2. ATLAS program begins with a BEGIN ATLAS PROGRAM statement and finishes with a TERMINATE statement. Note that each statement is started with a line-number, and the leading 'E' of the line-number E500100 means that the statement is the beginning of the main program. Similar to other programming languages, the ATLAS language provides the declaration statements for variables (e.g. line-number 000110), user-defined data types, function and procedure definitions, calculations of expressions (e.g. line-number 500112), program flow control statements (e.g. The IF-THEN-ELSE statement in line-number 500140), and input and output statements for transferring program data to and from an ATLAS program via consoles and/or files (e.g. line-number 500110), etc.

In addition to the common program statements, the ATLAS language provides additional statements performing the test procedure. It includes declarations of connections (e.g. line-number 000130), test resource allocation (REQUIRE statement) to establish and label test resource characteristics, the APPLY statement generating

```
 000100 BEGIN, ATLAS PROGRAM 'LAB1, PART 3' $
 000110 DECLARE, VARIABLE,  'RESULTS' IS DECIMAL$
 000120 DECLARE, VARIABLE, 'A' IS INTEGER              $
 000130 DECLARE, VARIABLE, 'HI-PIN' IS ARRAY (1 THRU 2) OF CONNECTION
          (J1-3, J1-4) INITIAL = CONNECTION J1-3, CONNECTION J1-4  $
E500100 APPLY, DC SIGNAL, VOLTAGE 5 V, CNX HI 'HI-PIN'(2) LO J1-11 $
 500110 OUTPUT, C'WAITING FOR 30 SEC FOR INPUT POWER STABILIZATION',C'\LF\' $
 500112 CALCULATE, 'A' = 30 $
 500115 WAIT FOR, 'A' SEC $
 500120 APPLY, AC SIGNAL, VOLTAGE 5 V, FREQ 60 HZ,
          CNX HI 'HI-PIN'(1) LO J1-10 $
 500130 VERIFY, (VOLTAGE INTO 'RESULTS'), AC SIGNAL,
          NOM 25.2 V UL 26.7 V LL 23.7 V,
          VOLTAGE RANGE 20 V TO 30 V,
          CNX HI J1-22 LO EARTH $
 500140 IF, GO, THEN $
 500150   OUTPUT, C'UUT PASSED' $
 500160 ELSE $
 500170   OUTPUT, C'UUT FAILED' $
 500180 END, IF $
 500190 WAIT FOR, 10 SEC $
 500200 VERIFY, (VOLTAGE INTO 'RESULTS'), AC SIGNAL,
          NOM 25.2 V UL 26.7 V LL 23.7 V,
          VOLTAGE RANGE 20 V TO 30 V,
          CNX HI J1-22 LO EARTH $
 500210 IF, GO, THEN $
 500220   OUTPUT, C'UUT PASSED' $
 500230 ELSE $
 500240   OUTPUT, C'UUT FAILED' $
 500250 END, IF $
 500260 REMOVE, ALL $
 999999 TERMINATE, ATLAS PROGRAM $
```

Fig. 2. An example of the ATLAS test program, note that the leading 'E' of the line-number E500100 means that the statement is the beginning of the main program.

or receiving a signal and defining the signal path between the UUT and the device (e.g. line-number 500120), the VERIFY statement to measure and compare with the values of UUT (e.g. line-number 500130), the REMOVE statement to remove particular established connections (e.g. line-number 500260 removing all established connections), etc.

2.2 Objectives of our ATLAS environment

In the following, we give the key objectives in our design of ATLAS compiler environments. We also give brief descriptions of our solutions to meet these objectives.

(1) **To support ATLAS compiler with IEEE Std 716-1995 specification:**
In this work, we develop the ATLAS compiler to aim at providing the control ability of the PC-based ATEs. To achieve this goal, we develop the compiler with the full implementation of IEEE standard.

(2) **To provide an ATLAS environment for ATE environments:**
ATLAS program graphs are needed to generate subsequent controls to be integrated with ATEs. We need a program graph designed with good method-

ologies to provide a systematic way for the manipulations of program trees. In our design, we employ object hierarchies to represent program graphs of ATLAS languages.

(3) **To establish connections between our program graph representations and XML tools:**

Due to large number of keywords and command constructs in ATLAS languages, we also need viewing tools for developers to manipulate the ATLAS program graph. In our design, we establish the connection between the object-oriented program graph and XML representations. With the support of DTD and XSL files of XML environments, we can perform machine validation on XML-based representations, map XML representations into graph structures, and annotate the program representations for human browsing. This connection allows ATLAS program trees to be operated under XML tools.

(4) **To support optimizations of program graph storage:**

To reduce the size of storage of program graphs in stores, we will hope to store the program graphs in binary representations. In our design, we employ the object serialization technology to achieve our goal for storing and retrieving the syntax trees and program graphs with binary representations.

3 The development of the C/ATLAS compiler

We now give the design steps for ATLAS compiler environments. These steps include transforming EBNF grammars into Yacc-format grammars, class hierarchies of program graphs, and object serializations of program graphs.

3.1 Lexical analyzer and grammars

In the process of compiler developments, we first get started with the lexical analyzer. The component for lexical analysis takes the input program and abstracts the tokens for the ATLAS parser. We retrieve the lexical definition from standard ATLAS syntax and develop the scanner with Flex (Mason and Brown, 1990; Paxson, 1995). Due to that the ATLAS language provides extreme large number of keywords for the nouns and modifiers to deal with all test cases, there are 900 keywords to be extracted approximately in our implementations for the lexical analysis. Figure 3 shows a typical set of syntactic structures extracted from the ATLAS lexical analyzer.

Next, we work on the grammar for the ATLAS language. We got the ATLAS language definition from the appendix of the ATLAS language specification. However, the language definition is in the Extended-BNF format, we need additional effort to automatically transform the EBNF into Yacc-like (BNF) grammars (Donnoly and

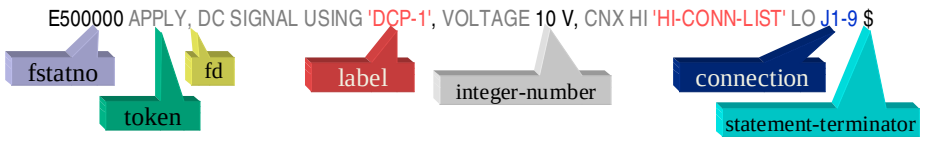

Fig. 3. The syntactic structures extracted from our ATLAS lexical analyzer.

Table 1
The operators and their meanings in the Extended-BNF syntax

Operator	Meaning
SYNTAX-RULES	Beginning of description
FINIS	End of description
:	Definition
*	Repetition 0 or more times
%	Repetition 1 or more times
|	Alternation
#	Separation
+	Optional chaining
{}	Grouping braces
[]	Optionality brackets
;	End of rule
!	Comment-indicator

Stallman, 1995; Mason and Brown, 1990). We need the transformation to be done automatically, as the ATLAS grammar is quite big and a transformation by hand will be hard. The EBNF grammar is listed 2687 lines in the language specification. The grammar in Yacc format after transformations from EBNF is around 6100 lines in our implementation. Table 1 shows the operators and their meanings used in the Extended-BNF syntax. The algorithm to convert an EBNF syntax grammar to the Yacc grammar is given in Figure 4 as well.

An example of the grammar in an EBNF form is listed below. It gives the declaration for record structures in ATLAS languages.

```
record-structure:
   "RECORD OF"
   "[" { { { record-field-identifier # fd }
   "IS" type } # ";" } "]" ;
```

The example grammar in an EBNF format shown earlier for record declarations are now translated into the form in Figure 5 according to our transformation algorithm. In the example, the algorithm described above recognizes the original EBNF "`record-field-identifier # fd`" as a sequence of `record-field-identifiers` separated by `fd`s when there are two or more `record-field-identifiers`, and it transforms the grammar into "`sep_006: sep_006 fd record_field_identifier | record_field_identifier;`". Then it finds that "`{sep_006 "IS" type}`" is grouped by braces, so it creates a new nonterminal and outputs the new grammar "`group_012: sep_006 T_IS type;`" to represent the group. There is another separation statement, "`group_012 # ";"`", so it can be transformed into "`sep_007: sep_007 ';' group_012 | group_012;`" as well. Finally, the top-level grammar can be outputed: "`record_structure: T_RECORD_sp_OF '[' sep_007 ']';`". We can also notice that the meta-parser converts the keywords into tokens, and change

Algorithm. Converting an EBNF grammar to a Yacc-like grammar (BNF)

Input: An EBNF syntax grammar with the operators specified in Table 1.

Output: The corresponding Yacc grammar.

Method: The parser reads the input string in rhs from left to right, and constructs the corresponding syntax tree from bottom to top. If the syntax tree has been built, output the syntax tree in Yacc grammar format, otherwise, output an error indication. The transformation rules for EBNF grammars are as follows:

repeat
  case rhs0
    : rhs1 rhs2:
      We build a node rhs0 with the children nodes as rhs1 rhs2.
    : rhs1 '*':
      We make a new nonterminal new_nt to represent rhs0,and set new_nt to rhs0,
      The output grammar for rhs0 is as follows:
      "new_nt : new_nt rhs1 | ;" It indicates zero or more times of rhs1.
    : rhs1 '%':
      We make a new nonterminal new_nt to represent rhs0, and set new_nt to rhs0.
      The output grammar for rhs0 is as follows:
      "new_nt : new_nt rhs1 | rhs1 ;". It indicates the repeat of one or more times of rhs1.
    : rhs1 '|' rhs2:
      build a node rhs0 of alternation with both the children nodes of rhs1 rhs2.
    : rhs1 '#' rhs2:
      We make a new nonterminal new_nt to represent rhs0, and set new_nt to rhs0.
      The output grammar is as follows:
      "new_nt : new_nt rhs2 rhs1 | rhs1 ;" It represents the repeating of rhs1, but separated by rhs2 among different appearances of rhs1.
    : '[' rhs1 ']':
      We make a new nonterminal new_nt to represent rhs0,and set new_nt to rhs0.
      The output grammar for rhs0 is as follows:
      "new_nt : rhs1 | ;". It indicates the optional appearance of rhs1.
    : '{' rhs1 '}':
      We make a new nonterminal new_nt to represent rhs0, and set new_nt to rhs0.
      The output grammar for rhs0 is as follows:
      "new_nt : rhs1 ;". It indicates the symbols in rhs1 are grouped (parenthesized).
    : terminal:
      set terminal to rhs0.
    : **else**:
      call error.
  endcase
until there is no more input symbol in rhs

Fig. 4. Algorithm of converting an EBNF syntax grammar to the Yacc grammar.

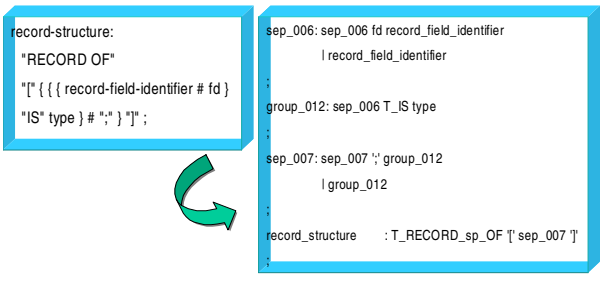

Fig. 5. An example for the transformation from EBNF to a Yacc-like format.

the hyphen in nonterminals to underscore to prevent Yacc from confusion.

As the original ATLAS language specification was written mainly for human-readable, ambiguous grammars and conflicts exist. After the transformation into Yacc-like

grammars, we further fine-tune the grammars to solve the grammar conflicts. The usual conflict, for example, is from the leading commas in front of different language structures, and ambiguous grammars in generating null elements. Once the grammar conflicts are solved, a parser is implemented to generate programs into an object-oriented intermediate forms. We will describe the object-oriented representations in the next sub-section.

3.2 The ATLAS abstract syntax tree

As the test programs were parsed by the parser, they should be translated to an intermediate form. Hence a completely class-library for the abstract syntax tree (AST) was designed for the intermediate form of parsed ATLAS programs. The AST was built in the object-oriented technology; all the entities within the AST were represented by the C++ classes. The program information is constructed in tree and table structures and described as follows.

- **Module and ModuleTable Classes**
  The module is the top-level element of the AST. It is also the top-level for program structures. A module is corresponding to a program module in ATLAS language. There is a module table in our AST to track module information in programs. Each module contains additional information for *StmtTable*, *NodeTable*, *SymbolTable*, and *TypeTable*. They represent the statement, expression, symbol, and type information, respectively. Figure 6 illustrates the concept of a module table.
- **Statement and StmtTable Classes**
  The statement class describes various statements in source programs. There are a variety of statements including variable declaration, resource requirement, connection requirement, subroutine definition, test steps, etc. The *StmtTable* consists of statements in the same module by incorporating the control flow information of statements.
- **Node and NodeTable Classes**
  The node class represents expressions in ATLAS programs. It gives the detailed information for each statement. The root of a tree is maintained in the statement class. In addition, there is a *NodeTable* to maintain the references to all nodes. The node class is linked by a form of binary trees. A variety of nodes are included. They are keywords, decimal numbers, character strings, connections, various arithmetic operations, etc. An example to represent expressions for node classes is illustrated in Figure 7. A statement node for *Calculate* statement is given. The detailed expressions is represented by the node class. Binary tree representations are used for linking nodes.
- **Symbol and SymbolTable Classes**
  The SymbolTable consists of both system-predefined symbols and user-defined symbols. The system-predefined symbols are predefined enumeration items such

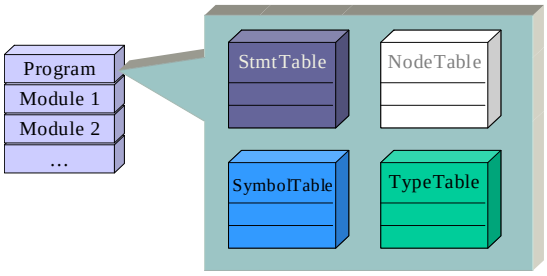

Fig. 6. The module table in the top level of the intermediate form.

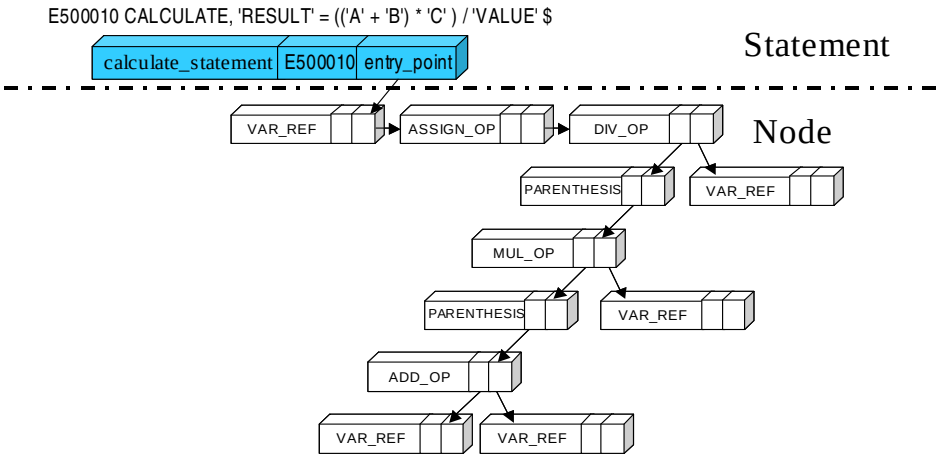

Fig. 7. An example of the expression nodes in AST of ATLAS programs.

as 'TRUE', 'FALSE', and 'BCD'. The user-defined symbols include program names, procedure names, variable names, constant names, connection items, enumeration items, etc. Each enumeration item has a link to the appropriate enumeration type entry in the *TypeTable*. Similarly, each variable symbol has a link to point to the appropriate type entry in the *TypeTable*. The *TypeTable* information is given next.

- **Type and TypeTable Classes**
  There are several types in *TypeTable*. First, it includes the primitive types, such as INTEGER, DECIMAL, CONNECTION, etc. Second, it includes the system-predefined enumeration types, e.g. the system-predefined symbol 'TRUE' is of BOOLEAN enumeration type. Finally, it includes the derived data types, such as ARRAY, RECORD, STRING, SUBRANGE, etc. Figure 8 illustrates the concepts of symbol and type nodes. In this example, we declare four variables of decimal type and a two-dimensional array of boolean type. Every variable reference in a particular expression has a link to point to the corresponding symbol in the *SymbolTable*, in which each symbol has its link to the declared type in *TypeTable*. If there is a composite type, e.g. array, the type entry will be further linked to the basic type of the composite type.

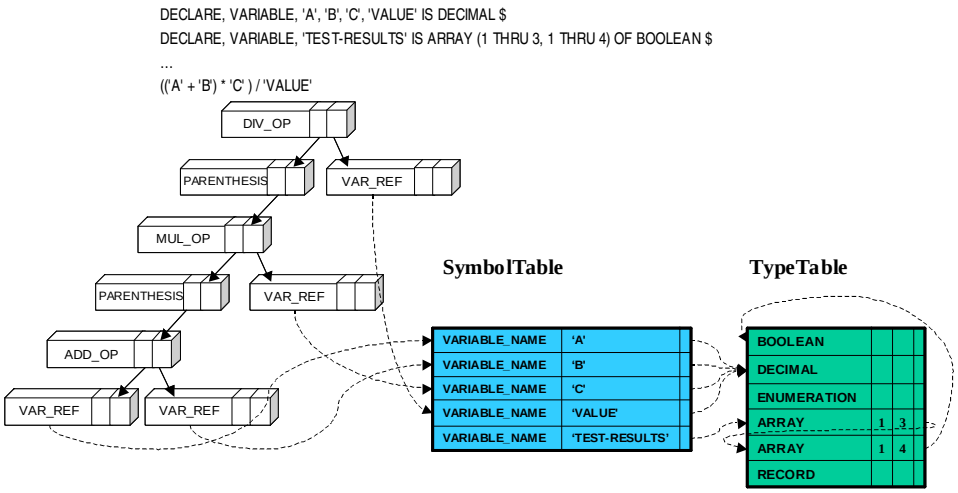

Fig. 8. An example of symbol and type information in AST.

3.3 The serialization mechanism for intermediate forms

The intermediate format with AST can be stored and read from stores. It can also be refined after the read from the store, and then be stored again. For continuous refinements, the AST needs to be with persistence. We employ the Microsoft Foundation Class Library (MFC) and object serialization technologies. The ASTs are saved to files in a compact binary format. All the classes describe in the previous sub-section override the Serialize() method, they store and retrieve their own data members within the methods. The table classes save all the entries in them. The links between instances and the tree structures are maintained by the MFC object serialization mechanism. The employment of object serialization techniques significantly reduce the programming effort from traditional compiler work in retrieving binary representations of program graphs from secondary storages. Traditionally, in retrieving binary representations of program graphs from secondary storages, one needs to carefully pack binary number order and bytes. This is normally an error-prone work.

To let our classes utilize the MFC mechanism for serialization, we make the classes inherit from the CObject class and make DECLARE_SERIAL macros. For example, the declaration of the statement class is done as follows.

```
class CStmt : public CObject, Dumpable {
public:
    DECLARE_SERIAL(CStmt);
    enum StmtType {
        apply_statement,
        define_complex_signal_statement,
        specify_component_signal_statement,
        ...
    };
// Construction/Destruction
    CStmt();
    CStmt(int lno, CString sno, StmtType var);
    virtual ~CStmt();
```

```
// Attributes
    CString statno;
    int lineno;
    StmtType variant;
    int id;
    CNode *entry_point;
    ...
// Implementation
    virtual void Serialize(CArchive& ar);
    virtual void AddNode(CNode *node);
    virtual char *GetStrStmtType();
    static char *GetStrStmtType(StmtType variant);
    virtual void PrintOut(FILE* fp, DumpType dumptype);
    ...
};
```

In the next step, the virtual function Serialize() must be implemented to provide the customized serialization functionality of the classes themselves: This is done as follows.

```
void CStmt::Serialize( CArchive& ar ) {
    CObject::Serialize( ar );
    if (ar.IsStoring()) {
        ar << (WORD) variant;
        ar << id;
        ar << lineno;
        ar << statno;
        ar << entry_point;
    }
    else {
        WORD wTemp;
        ar >> wTemp; variant = (StmtType)wTemp;
        ar >> id;
        ar >> lineno;
        ar >> statno;
        ar >> entry_point;
    }
}
```

The function is divided into two parts, to store data and to retrieve data, respectively. It will invoke the Serialize() function of its parent class, and then it will maintain the data members of itself. Note that the MFC mechanism will maintain the relationships of objects, such as the points to objects. For example, the item entry_point is a pointer to a *CNode* class in the source code listed above. When it is serialized, the MFC mechanism will maintain the graph of them. The other classes, such as *CModule* and *CNode*, also use the similar mechanism.

For the table classes, we have them inherit from the existing C++ template *CTypedPtrArray* in MFC. By using the *CTypedPtrArray* template, we can specify the container class and the element type for the table classes to employ. For example, the declaration of the *CStmtTable* class illustrates an example of such usages.

```
typedef CTypedPtrArray<CObArray, CStmt*> CStmtObjArray;

class CStmtTable : public CStmtObjArray, Dumpable
{
public:
    virtual int Add(CStmt *stmt);
    virtual void PrintOut(FILE* fp, DumpType dumptype);
```

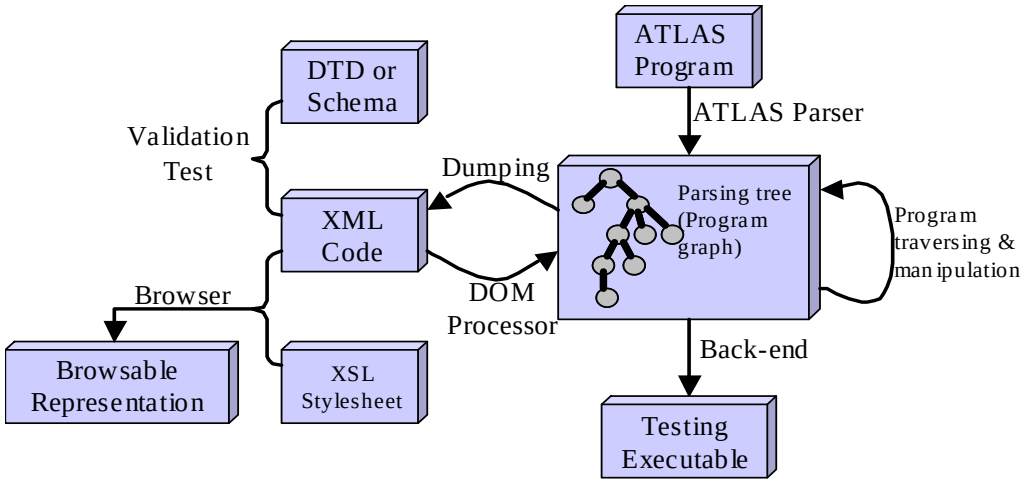

Fig. 9. The usage of XML representations in a compiler environment.

```
    ...
};
```

The first line of the code above defines the type *CStmtObjArray* as an instance type of *CTypedPtrArray* using *CObArray* template as the container and the pointer to *CStmt* as the element type. Class *CStmtTable* is then derived from *CStmtObjArray* and *CTypedPtrArray*. Note that there is no need to implement the Serialize() function here as the function in *CTypedPtrArray* template is equipped with such mechanisms. The other table classes, such as *CModuleTable* and *CNodeTable*, adopt this similar mechanism.

4 XML representations

We now explain how we establish the connections between our program graphs and XML toolkits. Figure 9 shows the major components of our system with XML representations. We will give detailed descriptions for each component in each subsection below. Among the components, with the Document Type Definition (DTD) (Bray et al., 2000) or XML Schema Language, we can validate the correctness and integrality of XML intermediate representations. With the Extensible Stylesheet Language (XSL) (Adler et al., 2000), we can transform XML intermediate files into table or graph structures which can be browsed in browsers or other XML toolkits.

4.1 XML Representations and Document Type Definition (DTD)

In this section, we try to establish the connections between our program graphs and XML representations. There are three dump formats in our design, the ASCII

format, HTML format, and XML format. The prior two formats are designed for the human-readable representations, besides, the XML format is a representation designed for both expression to human and validation to machine.

To implement the XML format for representing program trees, we first make an abstract class named *Dumpable*, and define a pure virtual function PrintOut() as an interface between the file writer and all kinds of elements in the program graphs:

```
class Dumpable {
public:
    enum DumpType {
        ASCII,
        HTML,
        XML,
    };
    virtual void PrintOut(FILE* fp, DumpType dumptype) = 0;
};
```

All the classes to dump their information then inherit from it. We have seen the examples in the previous subsection that the classes *CStmt* and *CStmtTable* both inherit from it. In addition, the classes must override the virtual function PrintOut() to provide their own dump function.

Next, we describe how we use DTD (Document Type Definition) for our program representations in XML format. A DTD defines a precise format for XML document structures. It describes the elements and attributes available, where and how many times they can occur, how elements can nest, and how elements and attributes can fit together. It can be used to validate the correctness and integrality of XML intermediate files. Figure 10 illustrates parts of the DTD file describing our XML intermediate form. In Figure 10, we can see the topmost element in intermediate form is *AtlasProgram*, which contains zero of more *Module* elements. Furthermore, there are four types of collections named *Stmts*, *Nodes*, *Symbols*, and *Types* in a *Module* element, each collection is composed of the same type of elements. We can see that there are several attributes of elements. For example, the *Symbol* element has the following attributes: “ID”, “VARIANT”, “ATTR”, “SCOPE”, and “TypeRef”, The type of “ID” is “ID #REQUIRED”. It means it’s an unique identifier of an element. Also the type of “TypeRef” is “IDREF #IMPLIED” means it’s an optional reference to a particular ID. When an XML intermediate file is loaded, the parser can check whether the destination of a reference exists.

For example, the following code segment is a XML skeleton; whose hierarchical structure and all elements and attributes are defined in the DTD file in Figure 10.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml:stylesheet type="text/xsl" href="atlas.xsl"?>
<!DOCTYPE AtlasProgram SYSTEM "atlas.dtd">

<AtlasProgram>
  <Module>
    <Stmts>
      <Stmt ID="A000" LINENO="001" STATNO=" 000100" EntryPoint="N000"
            VARIANT="begin_atlas_program_statement"/>
```

```
<?xml encoding="UTF-8"?>
<!ELEMENT AtlasProgram (Module)+>

<!ELEMENT Module (Stmts,Nodes,Symbols,Types)>

<!ELEMENT Stmts (Stmt)*>

<!ELEMENT Stmt EMPTY>
<!ATTLIST Stmt ID ID #REQUIRED
        LINENO CDATA #REQUIRED
        STATNO CDATA #REQUIRED
        EntryPoint IDREF #REQUIRED
        VARIANT CDATA #REQUIRED>

<!ELEMENT Nodes (Node)*>

<!ELEMENT Node (#PCDATA)>
<!ATTLIST Node ID ID #REQUIRED
        VARIANT CDATA #REQUIRED
        ChildNode IDREF #IMPLIED
        NextNode IDREF #IMPLIED
        SymbolRef IDREF #IMPLIED
        TypeRef IDREF #IMPLIED>

<!ELEMENT Symbols (Symbol)*>

<!ELEMENT Symbol (#PCDATA)>
<!ATTLIST Symbol ID ID #REQUIRED
        VARIANT CDATA #REQUIRED
        ATTR CDATA #IMPLIED
        SCOPE CDATA #IMPLIED
        TypeRef IDREF #IMPLIED>

<!ELEMENT Types (BasicType|RecordType|ArrayType|StringType|FileType|NamedType|SubrangeType)*>

<!ELEMENT BasicType (#PCDATA)>
<!ATTLIST BasicType ID ID #REQUIRED
        ATTR CDATA #IMPLIED
        SCOPE CDATA #IMPLIED
        VARIANT CDATA #REQUIRED>
...
```

Fig. 10. A Sample Fragment of the DTD File.

```
      ...
      <Stmt ID="A020" LINENO="052" STATNO=" 999999" EntryPoint="N159"
            VARIANT="terminate_atlas_program_statement"/>
    </Stmts>
    <Nodes>
      <Node ID="N000" VARIANT="LABEL">'LAB1,PPART2'</Node>
      ...
      <Node ID="N173" VARIANT="VAR_REF" SymbolRef="S047"></Node>
    </Nodes>
    <Symbols>
      <Symbol ID="S000" TypeRef="T007" VARIANT="ENUM_ITEM">FALSE</Symbol>
      ...
      <Symbol ID="S054" TypeRef="T002" VARIANT="VARIABLE_NAME">'VALUE'</Symbol>
    </Symbols>
    <Types>
      <BasicType ID="T000" VARIANT="INTEGER">INTEGER</BasicType>
      ...
      <SubrangeType ID="T022" VARIANT="SUBRANGE" NEXTTYPE="T021" LB="N172" UB="N173" />
    </Types>
  </Module>
</AtlasProgram>
```

In the XML document above, the file begins with headers specifying the stylesheet file and DTD file. The subsequent part defines the program graph in XML format. We can see that each element in a collection start with a tag name along with an attribute named ID, which is defined as a type "ID" in the DTD file. It means a unique identifier to the element. If there are references to a particular element, these references will be embedded in an attribute of type "IDREF" in the DTD file. For example, the attributes "SymbolRef" in a "Node" tag and "TypeRef" in a "Symbol" are of type "IDREF", and they express the references in AST.

4.2 The Extensible Stylesheet Language (XSL)

XSL, as a stylesheet language, supports separation of presentation and content, and it has considerable expressive power. By using XSL, we can transform XML documents into other formats of documents. We have developed an XSL file to transform the XML representations to HTML representations to demonstrate the usage. Therefore, when Internet Explorer opens a XML intermediate file along with the associated XSL file, the browser will annotate the XML file with HTML tags and display it. The abundant tools for XML can be used for manipulations of program graphs in XML representations.

In the usage of XSL to convert XML to HTML, The basic idea is converting a variety of elements in XML representation to particular tables respectively in a HTML file. Each row of a table contains an element along with its attributes. If the attribute is a IDREF, it fully utilizes the hyperlink to represent all connections between nodes. Once the format in HTML format, users can easily traverse among the statements and nodes and trace the links among symbols, types, and expressions. Furthermore, they can return to the original referencing states by just clicking the back button. This significantly reduces the effort to find cross-references among nodes of program trees from traditional compiler infrastructures.

Figure 11 and Figure 12 illustrate the transformed XML intermediate file with XSL for the source program originally shown in Figure 2. For example, statement number 500120 represents an apply_statement and statement number 500130 represents a verify_statement. These statement-level information can be looked up in Figure 11. The detailed expression information for those statement nodes can be further referenced by clicking N048 and N067 tags via html-based hyperlinks. In addition, statement number 00120 represents a declaration statement. The detailed expression information can be referenced by clicking the N005 tag, and it will automatically find N005 tag in the table below. It shows the beginning of the low-level expression chains. Similarly, Figure 12 gives an example for the variable symbol and type chains. There are mainly two tables in Figure 12. One is the symbol table and another is the type table. For example, variable "A" and "HI-PIN" can be found in the symbol table. Originally, variable "A" was declared in statement number

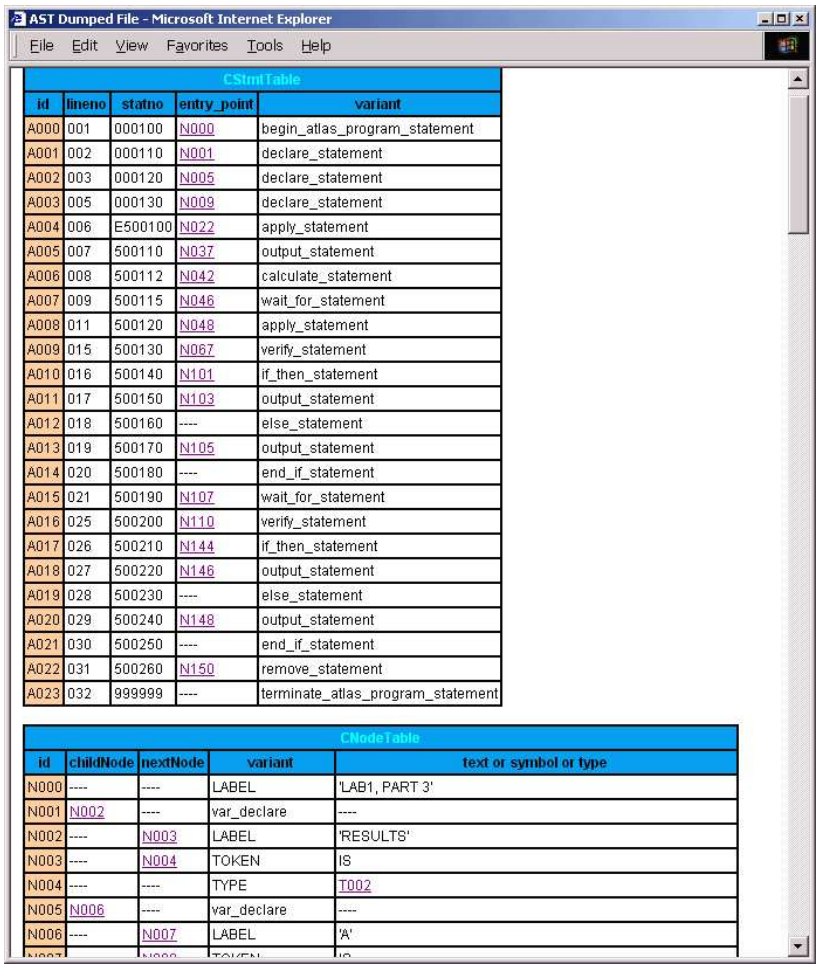

Fig. 11. The transformed XML intermediate file with XSL.

000120 and variable “HI-PIN” was declared in statement number 000130, respectively, in the code of Figure 2. “A” is hyperlinked to the type T000 in the type table which is an integer declaration. “HI-PIN” is hyperlinked to the type T010 in the type table which is an array declaration of base type T011, which is then a type of CONNECTION. Hyperlinks are used to help trace intermediate nodes.

In our design above, each basic element in the key nodes of AST (including statement node, expression node, symbol node, and type node) is first annotated with a corresponding tag in our HTML output. In the reference site, information related to the reference tag is again annotated the reference information. An example from the source above is listed as follows: “<A HREF=\"#N%03d\">N%03d</A>”. With such annotations, hence the cross-references can be traced by HTML hyperlinks.

Fig. 12. The symbol and type information of transformed XML representations into HTML representations.

4.3 The XML Parsers

There are two mainstreams of XML parser application programming interfaces (APIs) available:

- Document Object Model (DOM) (Hors et al., 2000), which is a tree-structure-based API issued as a W3C recommendation in 1998; and
- Simple API for XML (SAX) (Megginson, 2000), which is an event-driven API developed by the members of the XML-DEV mailing list.

We choose DOM as the API to re-construct object-oriented program representations from XML representations of program graphs. This enables a two-way transformation between XML and object-oriented program graphs. The DOM model represents an XML document as a tree whose nodes are elements, texts, and so on. It generates the tree and hands it to an application program. DOM provides a set of APIs to access and manipulate these nodes in the DOM tree. A DOM-based XML

parser creates the entire structure of an XML document in memory.

A DOM document object is returned after we invoke the parser to process an XML-based intermediate file. By using the method getElementById() introduced in DOM Level 2, we can select a particular element, whose ID is given by a string as a parameter, from a document entity. Therefore, the references to elements in the program graph can be traced in an XML-based intermediate file.

5 Experiments & Discussions

In this section, we performed experiments on compiling applications of ATLAS source programs to intermediate forms and stored them into files of XML and ASCII formats, respectively. The ATLAS compiler was evaluated using a set of ATLAS source files to test data structures, expressions, control flows, functions, modules, and various test steps in ATLAS programs. As this is a joint project between university and industrial research institute of Taiwan, the testing examples p2.atl, p3.atl, p4.atl, p5.atl, p8.atl, p9.atl, p10.atl, and p11.atl are laboratory examples from III (Institute for Information Industry) for testing and configurating sensor devices, stimulating UUT, and failure analysis. These samples are forwarded to us under the contract of Sekas Corp. and III, and the collaborations of III and our institute. They are used as a preliminary test for a variety of constructs for ATLAS programs. Among the testing samples, p2.atl gives tests for the declaration constructs of ATLAS programs. It includes the definition of digital configurations with source and sensor device, the storage of arrays for stimulus words, the connection variables and boolean logics, etc. p3.atl gives the test on a variety of constructs related to UUT. It includes the sense of UUT response, proving the UUT responses on UUT pins, stimulating UUT with the stimulus data, etc. Next, p4.atl gives essential test on the procedure routines in ATLAS programs. It includes the construct of routine declaration and the perform command. Next, p5.atl test programs on the construct of mixing control flows such as loops and procedure routines. p8.atl and p9.atl then give tests on command constructs of ATLAS programs such as require, apply, wait, measure, remove, etc. p10.atl and p11.atl give additional tests on command constructs of ATLAS programs such as identify, enable, disable, monitor, etc. It also tests timer, events, frequency adjustments, voltage conditions, etc. In addition, a large test case on industrial ATLAS source program (d37.atl) was used to evaluate our implementation. d37.atl was originally from NWA as a testing sample for erratic failures and UUT components. The application is also forwarded from III with their industrial partners. Note that d37.atl is more than 8500 LOC. It presents a real industrial application program as our test case.

Table 2 summarizes the experiments. The experiments are done on a 450 MHz Pentium II PC with 128 MB of RAM under Windows NT 4.0. The software is implemented by the Microsoft Visual C++ 6.0 compiler. Table 2 includes the compiler

Table 2
The experiment results

Filename	Source file			Serialization of AST			XML rep. of AST		HTML rep. of AST		ASCII rep. of AST	
	File size (bytes)	Number of lines	Number of statements	Number of nodes	Compile time (seconds)	File size (bytes)	Process time (seconds)	File size (bytes)	Process time (seconds)	File size (bytes)	Process time (seconds)	File size (bytes)
d37.atl	407K	8,520	4,477	21,288	1.047	566K	0.917	1.8M	1.253	3.1M	0.787	1.4M
p2.atl	2.1K	51	21	169	0.009	6.0K	0.009	19K	0.013	35K	0.01	15K
p3.atl	1.2K	32	24	155	0.007	4.4K	0.008	14K	0.011	26K	0.007	12K
p4.atl	2.2K	37	30	177	0.009	5.7K	0.008	17K	0.012	31K	0.029	13K
p5.atl	1.6K	33	26	148	0.007	4.9K	0.009	15K	0.012	27K	0.008	12K
p8.atl	1.5K	37	23	183	0.007	5.1K	0.009	16K	0.012	29K	0.008	13K
p9.atl	1.7K	28	19	119	0.006	3.5K	0.007	11K	0.008	21K	0.007	9.4K
p10.atl	2.4K	57	44	229	0.008	6.7K	0.009	20K	0.014	37K	0.009	17K
p11.atl	1.7K	34	30	151	0.007	4.9K	0.008	14K	0.010	26K	0.007	12K

time and the file sizes with respect to the source files, serialization of AST (abstract syntax tree) format, ASCII dump of AST files, HTML representation of AST files, and XML representation of AST files. There are totally five categories in Table 2. It includes the source file information, the information on the serialization of AST files, the information on the XML representation of AST forms, the information on the HTML representation of AST forms, and information on the ASCII representation of AST forms. The time listed in the second category is the compiler time to transform programs into object-oriented program graphs (AST) and then perform serialization of program graphs. The process time listed in category 3, 4, and 5 are only the process time to transform AST format (object-oriented program graphs) into the XML represntation, HTML representation, and ASCII representation of AST, respectively. A conventional representation for AST is the format with an ASCII representation. As we employ the serialization technique with object-oriented technologies to serialize objects into binary format automatically, the size of the representation can be reduced significantly. For example, with d37.atl application (one of the biggest application in our experiment), the ASCII representation is 2.47 times larger than our serialization representation. Our employment of serialization technologies helps reduce the size of intermediate representations. Next, we look at the size of the source files and the size of our AST representations. Our AST representation consists of statement, expression, symbol, and type information of the source programs. The serialization representation of our AST form grows slowly in proportions to the source files. Finally, with the XML representations of AST file, one can use XML toolkits to view the program graphs. Here we also give reference number for the size of program graphs in each format. The actual store of the program graph is in the serialization of object-oriented graphs, as it's the smallest one. All the other formats can be converted to each other in our system depending on the developer's requirements. For example, we can transform XML representations to HTML representations for browsing cross references.

Next, we discuss the benefit of our design in addition to the experiments shown above for the reduction of program graph in storages. As described earlier, our system is designed with object-oriented representations for program graphs. Object-oriented representation of program graph provides a systematic way for the manipulations of program trees. System software writers can use this object-oriented representation to generate subsequent controls to be integrated with ATEs. Next,

we establish the connection between the object-oriented program graph and XML representations. With the support of DTD and XSL files of XML environments, we can perform machine validation on XML-based representations, transform XML representations into graph structures, and annotate the representations for human browsing. This connection provides great flexibilities for ATLAS program trees to be operated under XML tools. As described earlier, due to large number of keywords and command constructs in ATLAS languages, we need good software design to make it easy to traverse program graphs and for consequent software developments with ATE environments. Our design with XML representations make it easy for developers dealing with program graphs to be able to view and connect program graphs and ATLAS command constructs with XML viewing tools. Our work presents the first-hand software experiences for the state of the art design and implementation of a complex application domain language with object-oriented designs and XML representations.

The ATLAS program environment is now shiped to III (Institute for Information Industry, a local industrial research institute), aerospace division, under university and industrial joint project contracts. Regarding to the constraints of our sysem, as we support IEEE standard compliant ATLAS-716-1995 parser, our system does not cover vendor extensions for ATLAS languages beyond IEEE standard compliant ATLAS-716-1995. The efforts for research and development in our site for the compiler environments include two researchers for working one year. The compiler infrastructure was then used by research engineers at III to generate controls to work with ATEs. The integrated platform was then used for test applications where long-term maintainability and reliability of UUTs (unit under test) was required. Our system system gives the essential components for ATLAS programming environments.

6 Conclusions

ATLAS has proven to be of substantial values in the automatic test specifications on ATEs. We think our implementation of IEEE standard compliant ATLAS-716-1995 parser can bring valuable experimental experiences for the ATE vendors. On top of current architecture and implementation, support for ATLAS-2000 (IEEE SCC-20, 1997) could be easily done. Our compiler is also done with the employment of advanced software techniques including object-oriented representations, serializations of objects, and the XML representation for AST information for employing abundant XML toolkits with compiler environments. Our software infrastructure can be used for subsequent controls and specifications for ATEs. We think our work gives the first-hand software experiences for modern compiler systems on complex application domain-specific languages.

Acknowledgements

The authors express their gratitude to the anonymous reviewers for their valuable suggestions and comments.

References

Adler, S., Berglund, A., Caruso, J., 2000. Extensible Stylesheet Language (XSL) Version 1.0, W3C Candidate Recommendation CR-xsl-20001121. (available at http://www.w3.org/TR/xsl).

Aho, A.V., Sethi, R., Ullman, J.D., 1988. Compilers-Principles, Techniques and Tools, Addison-Wesley.

Bray, T., Paoli, J., Maler, E., 2000. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation REC-xml-20001006. (available at http://www.w3.org/TR/2000/REC-xml-20001006).

Chang, R.G., Chen, C.W., Chuang, T.R., Lee, J.K., 1997. Toward Automatic Support of Parallel Sparse Computation in Java with Continuous Compilations, Concurrency: Practice and Experiences, Vol. 9(11), 1101-1111.

Chang, R.G., Li, J.S., Chuang, T.R., Lee, J.K., 2001. Probabilistic inference schemes for sparsity structures of Fortran 90 array intrinsics, International Conference on Parallel Processing, Spain.

Cortner, J.M., 1987. Digital Test Engineering, John Wiley & Sons, Inc.

Donnoly, C., Stallman, R., 1995. Bison, the YACC-compatible Parser Generator, Free Software Foundation, Cambridge, Massachusetts.

Gannon, D., Lee, J.K., 1992. SIGMA II: A Tool Kit for Building Parallelizing Compilers and Performance Analysis Systems, Proceedings of Programming Environments for Parallel Computing Conference, Edinburgh.

Holub, C., 1993. Compiler Design in C, Prentice-Hall Inc.

Hors, A.L., Hégaret, P.L., Wood, L., 2000. Document Object Model (DOM) Level 2 Core Specification Version 1.0, W3C Recommendation REC-DOM-Level-2-Core-20001113. (available at http://www.w3.org/TR/DOM-Level-2-Core).

Hulme, A.M.B., 1996. The ATLAS language-panacea or pariah? Does it only specify the task or does it really drive the tester?, IEEE Aerospace and Electronics Systems Magazine, 11(3):29–34.

Hutt, A.T., 1994. Object Analysis and Design, Object Management Group, John Wiley & Sons, Inc.

Hwang, G.H., Lee, J.K., Ju, D.Z., 1998. A Function-Composition Approach to Synthesize Fortran 90 Array Operations, Journal of Parallel and Distributed Computing, 54, 1-47.

Hwang, G.H., Lee, J.K., Ju, D.Z., 2001. Array operation synthesis to optimize HPF programs on distributed memory machines, Journal of Parallel and Distributed Computing, 61, 467-500.

IEEE SCC-20, 1997. ATLAS-2000 Test Language Requirements Document, IEEE SCC-20 Log No. 716-051.

IEEE Std 771-1998, 1998. IEEE guide to the use of the ATLAS specification, IEEE, Inc.

IEEE Std 716-1995, 1995. IEEE standard test language for all systems-Common/Abbreviated Test Language for All Systems (C/ATLAS), IEEE, Inc.

Lee, J.K., Tsaur, I.K., Hwang, S.Y., 1997. Parallel Array Object I/O Support on Distributed Environments, Journal of Parallel and Distributed Computing, 40, 227-241.

Mason, T., Brown, D., 1990. lex & yacc, O'Reilly.

Megginson, D., 2001. SAX 2.0: The Simple API for XML, XML-DEV mailing list. (available at http://www.saxproject.org/).

Parker, K.P., 1987. Integrating Design and Test: Using CAE Tools for ATE Programming, Computer Society Press of the IEEE.

Paxson, V., 1995. Flex, version 2.5, A fast scanner generator, The Regents of the University of California. (available at http://www.math.utah.edu/docs/info/flex_toc.html).

Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S.M., Hennesy, J.L., 1994. SUIF: An Infrastructure for research on parallelizing and optimizing compilers, ACM SIGPLAN Notices, 29(12): p. 31-37.

Wu, J.Z., Lee, J.K., 2000. A bytecode optimizer to engineer bytecodes for performances, in Languages and compilers for high-performance computing (LCPC '00), USA.

Cheng-Wei Chen received his B.S. degree in computer science and information engineering from Yuan Ze University and M.S. degree in computer science from National Tsing-Hua University in 1995 and 1997, respectively. He is currently a Ph.D. candidate, Department of Computer Science, National Tsing-Hua University, Taiwan. His research interests include object-oriented and parallel languages, the software for component-based environments, and compiler designs.

Jenq Kuen Lee received the B.S. degree in computer science from National Taiwan University in 1984. He received a Ph.D. in computer science from Indiana University in 1992, where he also received a M.S. (1991) in computer science. He was a key member of the team who developed the first version of the pC++ language and SIGMA system while at Indiana University. He joined the Department of Computer Science at National Tsing-Hua University, Taiwan, in 1992. Since then, he established a programming language research lab. there to develop advanced compiler toolkits for embedded systems and modern microprocessors. He is now a professor and vice-chairman with the computer science department, National Tsing-Hua Univ., Taiwan. He was a recipient of the most original paper award in ICPP '97 with the paper entitled "Data Distribution Analysis and Optimization for Pointer-Based Distributed Programs". The dissertation of his Ph.D. student, Gwan-Hwan Hwang, also received the distinguished dissertation award as honorable mention by

IICM of Taiwan, 1999. He was also a receipient for a Taiwan MOE award, 2001, on delivering compiler technologies for advancing the related industry sector in Taiwan.

He currently participated in the NSC/MOE research excellence project of Taiwan in developing advanced system software for pervasive computings, in the USA-NSF/Taiwan-NSC joint project as an international effort to develop advanced compiler toolkits for SoC designs, and in MOEA Project of Taiwan on Compiler Technologies for SoC designs. His current research interests include optimizing compilers, SDK toolkits and ISA specifications for SoC designs, Java software for pervasive computings, and parallel object-oriented languages and systems.