Extending a source-to-source compiler with XML capabilities

Sergio Aldea
Dpto. de Informática
Univ. de Valladolid
saldeal@gmail.com

Diego R. Llanos
Dpto. de Informática
Univ. de Valladolid
diego@infor.uva.es

Arturo González-Escribano
Dpto. de Informática
Univ. de Valladolid
arturo@infor.uva.es

Abstract

This paper presents an extension that adds XML capabilities to Cetus, a source-to-source compiler developed by Purdue University. In this work, the Cetus Intermediate Representation is converted into an XML DOM tree that, in turn, enables XML capabilities, such as searching specific code features through XPath expressions. As an example, we write an XPath code to find private and shared variables for parallel execution in C source code.

Loopest is a Java program with embedded XPath expressions. While Cetus needs 2573 lines of internal JAVA code to locate private variables in an input code, Loopest needs a total of only 425 lines of code to determine the same private variables in the equivalent XML representation.

Using XPath as search method provides a second advantage over Cetus: extensibility. Changes in Cetus requires a deep knowledge of Java, Cetus internal structure, and its Intermediate Representation. Moreover, changes in Loopest are easier because it only depends on XPath to generate reports. Finally, we present Sirius, an XML DOM tree-to-C converter, that allows to generate the new output C code based on the annotations done in the XML tree.

1 Introduction

This work aims to extend Cetus, a source-to-source compiler developed by Purdue University [1, 2, 3], with XML capabilities. Cetus Intermediate Representation (IR) can be transformed in an XML DOM tree, which enables to use powerful XPath features and tools. XPath queries work in a similar way than recursive searches in a directory-based filesystem structure, allowing to select nodes or set of nodes in an XML document easily through path expressions.

The goal of this paper is twofold: To provide an extension to Cetus which incorporates XML capabilities, and to compare the advantages of using XPath instead Cetus internal mechanisms in terms of easy of programming, and extensibility. As an example, we show the benefits of XPath as an alternative to search private variables for parallel execution in a C source code.

This paper is structured as follows. Section 2 introduces Cetus, with a brief description of its structure and IR. Section 3 defines the System Architecture and also describes XMLCetus as extension of Cetus (Cetus IR-to-XML DOM tree converter). In this section we also describe Sirius, an XSLT program that transform the XML DOM tree in C source code. As a case of use, Section 4 describes Loopest, a program that uses XPath to classify variables as private or shared for parallel execution. Section 5 shows an example of C source code executed by this architecture and the differences between Cetus and XML-Cetus+Loopest in terms of number of lines and extensibility. Finally, Section 6 summarizes the results and provides the main conclusions of this work.

2 Understanding Cetus

The Cetus Project is developed by Purdue University (Indiana, USA), written in Java and distributed under a modified Artistic Li-

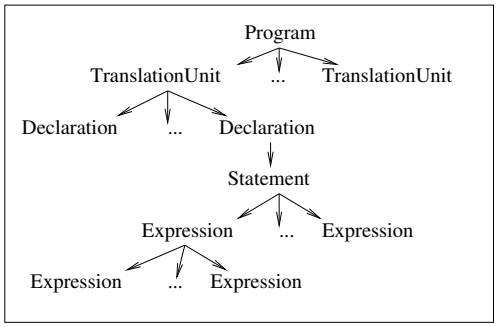

Figure 1: Cetus IR.

cence [7]. Cetus is a compiler infrastructure for source-to-source transformation of programs written in C. Cetus provides several functions, such as auto-parallelization of loops through private- and shared-variables analysis, and automatic insertion of OpenMP Directives.

Cetus works with an Intermediate Representation (IR)(see Figure 1), an abstract representation that holds the block structure of a C program. It is implemented in the form of a class hierarchy and accessed through the class member functions.

In Cetus, the concept of statements and expressions are closely related to the syntax of the C language, making the source-to-source translation easy. However, there are some disadvantages: an increasing complexity for pass writers (since they should think in terms of C syntax) and limited extensibility to process additional languages. Fortunately, this problem is mitigated by the provision of several abstract classes, which represent generic control constructs. Thus, generic passes can be written using the abstract interface, while more language-specific passes can use the derived classes.

Figure 2 shows an example of Cetus IR from a C source code. In Cetus terminology, a "TranslationUnit" is a file containing source code. The syntax tree shown in Fig. 1 and the class hierarchy are not completely equivalent. For example, in the syntax tree, the parent of a TranslationUnit is a Program, however neither

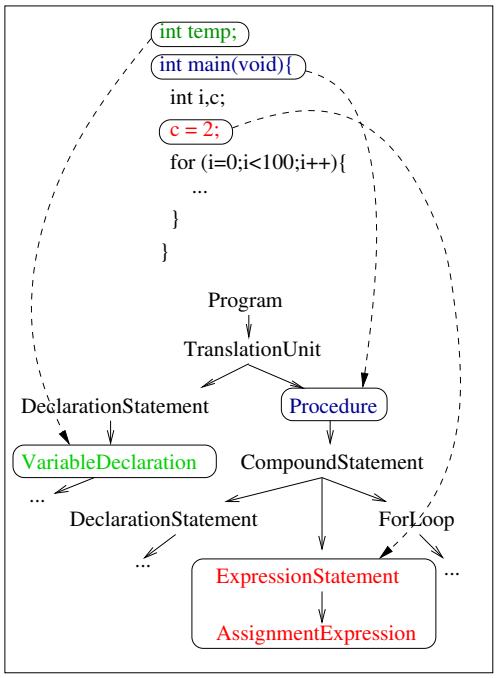

Figure 2: IR Tree Structure Example.

TranslationUnit nor Program have a parent in the class hierarchy.

3 System Architecture

Figure 3 shows the proposed system architecture. The input of this system is the original C file. XMLCetus receives this file and creates an XML file that represents the original source and contains all the information needed to rebuild the code. At this point, using tools for handling XML DOM trees, it is possible to generate a report with relevant information from the code. It is also possible to modify the XML DOM tree with automatic XML tools, in order to create a modified C source file (not shown in the figure). Once the transformation is done, the modified XML file is the input of Sirius, which rebuilds the C source code with the XML information provided. For legibility, we use the GNU tool `indent` to format the

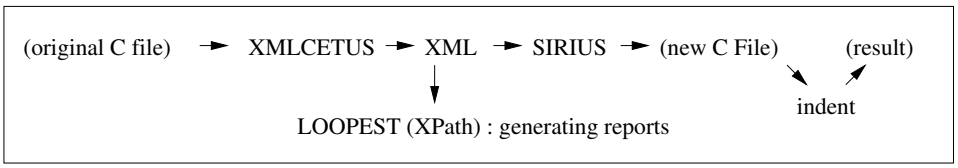

Figure 3: System Architecture.

result. The following sections explain XMLCetus and Sirius in more detail.

3.1 XMLCetus

XMLCetus is a modification of Cetus that also generates an XML DOM tree based on Cetus IR. Since Cetus is written in Java, it is easy to add functionalities to the code. XMLCetus is composed by a new class `Xml` included in a new package `cetus.xml`. The package created is imported by the `Driver` class, which is the entry point of the system. The main changes to Cetus are made just after Cetus has finished the analysis of the C source and has generated the IR. At that point, the *cetus.xml*'s function called `createDomTree()` is invoked with the first node of the tree (*Program*) as input parameter. This function creates a new DOM document with the XML tree, that can be printed by the `printDomTree()` function.

Every node of the IR has a corresponding representation in the XML DOM tree, preserving the original structure of the Cetus IR. The following steps explains the transformation procedure from Cetus IR to XML DOM tree:

1. Beginning with the first node, *Program*, we recursively descend all its children: TranslationUnit nodes which represent the source code files passed to Cetus. The tree is traversed with a preorder, depth-first search. The `getChildren()` function gets the children of each node as a list of nodes. Through a casting operation, the list is transformed into a *Traversable*-type objects list. *Traversable* is the type defined by Cetus as the generic class, representing any kind of node.

2. Next, the type of the children nodes are checked. This is done with the help of a loop that traverse each node and find, with subsequent checks, the class this node is an instance of. (using the `instanceof` Java operator). If one node is an instance of a given class and its parent class, two DOM elements that represents both nodes are created to reflect the original structure. A different situation arises when a node is an instance of a superclass and not of any of its subclasses. These facts, among others, are taken into account when creating DOM nodes and its relationships.

3. When the checking of the instance is positive with a particular class, an object instance of this class is created, through the node and a casting operation. Now, it is possible to obtain relevant information from the node and create a new DOM element with attributes that reflects this information.

4. Finally, after the creation of the element with its attributes, the DOM element is appended to its corresponding parent.

This procedure generates an XML file which represents the DOM tree. Using this file as input of Sirius, it is possible to rebuild the original C source transformed by Cetus.

3.2 Sirius

Once XMLCetus creates the XML output file with the DOM tree, it may be necessary to transform this tree to a different format. The best way to perform this process is using

XSLT. We built an XSLT program with template rules to translate the XML output file again to C. These rules guide the transformation from the DOM tree to a compilable source code that should be equivalent to the original source code after Cetus processing. To execute the XSLT program, we chose the *Saxon* [4] tool, due to its open-source nature and because it implements XPath and XSLT 2.0.

The structure of the XSLT program developed consists in a set of template rules, one for each element of the DOM tree that should be transform to C language constructions. This template rules generate the C code that corresponds to the DOM element identified, and indicate the application of another rule where necessary. For example, in the case of a *BinaryExpression* element, the corresponding template rule generates the correct binary operator, according to the value of the corresponding attribute, and decide the application of the appropriate template rules to the left and right side expressions of the binary operator.

As an example of use the XML DOM tree, together with the information obtained by its analysis, can be used to generate modified versions of the original C code, for example to parallelize loops with additional directives. We have developed a tool, called Loopest, that explores this possibility. This tool is described in the following section.

4 Loopest

Loopest is an experimental tool to automatically analyze the XML output file generated by XMLCetus, in order to classify the use of variables inside `for` loops. Loopest is written in Java and executes a set of XPath queries which determine the variables being read, being written, being read-and-written, and some other queries useful to find private variables in a loop. Once the queries finish, the sets of variables obtained are used to generate reports, thanks to the use of the package *ListUtils*, provided by Apache Commons [5].

Loopest was developed using XPath because it provides simplicity and extensibility. XPath syntax is easy to learn and provides enough functionality. It is possible to build complex queries with few words or lines. The result of these queries can be new node-sets that can be easily combined to search for new results. One of the main advantages of using such a tool is that its functionality can be modified easily, detecting new language constructions by adding or modifying queries.

```
int main() {
   int i, a, b=10;
   for (i=0; i<b; i++) {
      a=b+1;
   }
}
```

Figure 4: Example of C code.

The following example shows the entire process, analysing the XML representation of an original C code and generating a report.

5 An Example of the XML DOM Tree and XPath Capabilities

The main differences between Cetus and XMLCetus+Loopest are simplicity and extensibility. Detection of private variables is developed in Cetus too, but the code required to implement this functionality is much longer and complex than Loopest's code. Also, performing changes to Cetus' functionality requires a deep knowledge about Cetus software and its intermediate representation. Changes in Loopest software are much easier, because it is developed with XPath, not even requiring a widespread knowledge about Java or XML.

Figure 4 shows a simple C source code, with only a loop and three variables involved. The XML representation of this code is shown in Figure 5. Only the main elements which help to better understand Loopest functionality are shown in the example. The name of these nodes are the same as the nodes of Cetus Intermediate Representation, and the value of their attributes are also extracted from Cetus IR. The entire code is represented with a simple

```
<Program>
  <TranslationUnit filename="example1.c">
    <Declaration>
      <Procedure name="main" numParameters="1">
        ...
          <CompoundStatement>
            <Statement>
               ...
                 <VariableDeclaration numDeclarators="3">
                   <Declarator>
                      ...
                           <Identifier array="" name="i" opUnary="" type="int"/>
                   </Declarator>
                   <Declarator>
                      ...
                           <Identifier array="" name="a" opUnary="" type="int"/>
                   </Declarator>
                      ...
            <Statement lineNumber="3">
              <ForLoop condition="i&lt;b" initial="i=0;" lineNumber="3" step="i ++ ">
                <Statement>
                  ...
                      <BinaryExpression operator="=">
                          <Expression>
                            ...
                              <Identifier array="" name="i" opUnary="" type="int"/>
                          </Expression>
                          <Expression>
                            ...
                              <IntegerLiteral value="0"/>
                </Statement>
                <Expression>
                  <BinaryExpression operator="&lt;">
                    <Expression>
                      ...
                        <Identifier array="" name="i" opUnary="" type="int"/>
                    <Expression>
                      ...
                        <Identifier array="" name="b" opUnary="" type="int"/>
                </Expression>
                <Expression>
                  <UnaryExpression expression="i" operator="post ++">
                    ...
                        <Identifier array="" name="i" opUnary="" type="int"/>
                </Expression>
                <Statement lineNumber="3">
                  <CompoundStatement>
                    ...
                          <BinaryExpression operator="=">
                              <Expression>
                                  <Identifier array="" name="a" opUnary="" type="int"/>
                              <Expression>
                                <BinaryExpression operator="+">
                                  <Expression>
                                      <Identifier array="" name="b" opUnary="" type="int"/>
                                  <Expression>
                                      <IntegerLiteral value="1"/>
                                  ...
```

Figure 5: XML example.

```
Number of ForLoop:  1
----------------------
3:  Line number:  3
3:  Loop Control Variable:(int) i.
3:  Variable involved in the Loop
Control:(int) b.
3:  Variables only read:(int) b.
3:  Variables only written:(int) a.
3:  Private Variables:(int) a,(int) i.
3:  Shared Variables:(int) b.
```

Figure 7: Report generated by XMLCetus+Loopest for the program shown in Fig. 4.

XML tree structure, enabling to use XPath capabilities to generate reports. XPath queries work in a similar way than recursive searches in a directory-based filesystem structure, allowing to select nodes or set of nodes in an XML document easily. Figure 6 shows some queries used in Loopest.

The `for` loop constructors have their own nodes in the tree. A `for` loop node has always the same structure and children nodes:

- A *Statement* node which represents the initial statement.
- A *Expression* node which represents the loop condition.
- A second *Expression* node, which is the loop step.
- A *Statement* node which contains the loop body.

Returning to Figure 4 code, if `-privatize` option of Cetus is activated, Cetus detects `a` and `i` as private variables, and adds the following line into the code:

```
#pragma cetus private(a, i)
```

Loopest identifies the same variables as private and produces the report shown in Figure 7:

As can be seen, both Cetus and XMLCetus+Loopest produce the same results with respect to private variables identification. However, the difference is the effort needed to carry out this task. We may use the number of code lines needed as an effort indicator. In Cetus, at least eight java classes take part directly to locate the private variables of a given loop, consuming 2573 lines of code (calculated by SLOCCount [6]). However, Loopest only needs 425 lines of a lower complexity to carry out the same task.

6 Conclusions

The aim of this paper is to explore the advantages of using XML representation of C programs to generate reports easily and to allow automatic modifications to its structure. To do so, we have developed XMLCetus, a modified version of Cetus that uses its Internal Representation to generate an XML document, and Loopest, an experimental application that process this document in order to classify the use of variables in `for` loops. Regarding this particular problem, we have shown that the combined use of XMLCetus+Loopest solves the problem easier than invoking Cetus internal methods, allowing a 85% reduction of the number of code lines needed. Another advantage of processing the XML document instead of working directly with Cetus is that the latter option requires a deep knowledge about Java, Cetus Intermediate Representation, and its associated data structures. On the other hand, new functionalities can be added in Loopest simply by adding new XPath queries, only requiring some basic knowledge about XPath and Java to combine the results into meaningful reports.

Finally, the XML document generated by XMLCetus can be easily processed to modify the C code or to add new directives. Again, these modifications can be done much easily than modifying Cetus internal structure directly. We have developed a third tool, called Sirius, that handles an XML document with XMLCetus format and produces a new C program based on it.

We believe that the use of XML tools, such as XPath, is an interesting alternative to perform code analysis and modification, and that

```
// Variables written.
// Variables as indexes in a ArrayAccess are not written.
XPathExpression varLeftLoop =
  xpath.compile("Statement[2]/CompoundStatement//
                  ( (AssignmentExpression/Expression[1]//Identifier[
                      not(ancestor::AccessExpression) and not(ancestor::ArrayAccess)
                     ])
                  | (VariableDeclarator[ descendant::Initializer ]/
                      Expression//Identifier[
                          not(ancestor::AccessExpression) and not(ancestor::ArrayAccess)
                     ])
                  | (FunctionCall/Expression[ position()!=1 ]//
                      UnaryExpression[ @operator='&' ]//Identifier[
                          not(ancestor::AccessExpression) and not(ancestor::ArrayAccess)
                     ])
                  )");

// ArrayAccess written.
XPathExpression varLeftArrayLoop =
  xpath.compile("Statement[2]/CompoundStatement//
                  ( (AssignmentExpression/Expression[1]//
                      ArrayAccess/Expression[1]//
                         Identifier[ not(ancestor::AccessExpression) ])
                  | (VariableDeclarator[ descendant::Initializer ]/
                      Expression//ArrayAccess/Expression[1]//
                         Identifier[ not(ancestor::AccessExpression) ])
                  | (FunctionCall/Expression[position()!=1]//
                      UnaryExpression[ @operator='&' ]//ArrayAccess/Expression[1]//
                         Identifier[ not(ancestor::AccessExpression) ])
                  )");

// AccessExpression Variables written.
// The option "VariableDeclarator + Initializer" is not considered because that
   construction is not possible.(example long date.r1 = 5).
XPathExpression varLeftAccessLoop =
  xpath.compile("Statement[2]/CompoundStatement//
                  ( (AssignmentExpression/Expression[1]//
                      AccessExpression[ not(ancestor::AccessExpression) ])
                  | (FunctionCall/Expression[position()!=1]//
                      UnaryExpression[@operator='&']//
                          AccessExpression[ not(ancestor::AccessExpression) ])
                  )");

// Variables read and written (from unary increments or decrements).
XPathExpression varUnary =
  xpath.compile("Statement[2]//ExpressionStatement/Expression//(UnaryExpression[
                      @operator='post ++' or @operator='post --'
                      or @operator='pre ++' or @operator='pre --'
                  ])//( (Identifier[ not(ancestor::AccessExpression) ])
                      | (AccessExpression[ not(ancestor::AccessExpression) ])
                      )");
```

Figure 6: XPath example: Search for variables written inside a loop. Code is indented for readability

it opens new possibilities to extend the original capabilities provided by Cetus.

Acknowledgements

This research is partly supported by the Ministerio de Educación y Ciencia, Spain (TIN2007-62302), Ministerio de Industria, Spain (FIT-350101-2007-27, FIT-350101-2006-46, TSI-

020302-2008-89, CENIT MARTA, CENIT OASIS), and Junta de Castilla y León, Spain (VA094A08).

References

[1] Hansang Bae and Leonardo Bachega and Chirag Dave and Sang-Ik Lee and Seyong Lee and Seung-Jai Mind and Rudolf Eigenmann and Samuel Midkiff, *Automatic Parallelization with Cetus*, HPCLAB, ECE, Purdue University, 2008.

[2] Chirag Dave and Hansang Bae and Seung-Jai Mind and Seyong Lee and Rudolf Eigenmann and Samuel Midkiff, *Cetus: A Source-to-Source Compiler Infrastructure for Multicores*, IEEE Computer, 42(12):36-42, December, 2009.

[3] Hansang Bae and Leonardo Bachega and Chirag Dave and Sang-Ik Lee and Seyong Lee and Seung-Jai Mind and Rudolf Eigenmann and Samuel Midkiff, *Cetus: A Source-to-Source Compiler Infrastructure for Multicores*, Proceedings of the 14th Int'l Workshop on Compilers for Parallel Computing, CPC, 2009.

[4] Saxonica Limited, *Saxonica. XSLT and XQUERY Processing*, `http://www.saxonica.com`.

[5] Apache Commons, `http://commons.apache.org/collections`.

[6] David A. Wheeler, *SLOCCount: Counting Source Lines of Code*, `http://www.dwheeler.com/sloccount`.

[7] Purdue University, Cetus Licence, `http://cetus.ecn.purdue.edu/download.html`.

[8] Michael Kay, *XPath 2.0 Programmer's Reference*, John Wiley & Sons, 2004.