Xevolver XML Framework

A Framework for XML-Based AST Transformations

Introductory Tutorial

(AN UNFINISHED DRAFT)

Hiroyuki Takizawa
Graduate School of Information Sciences
Tohoku University
Sendai 980-8579 Japan
+81-22-795-7010 (office) +81-22-795-7011 (fax)
takizawa@cc.tohoku.ac.jp

June 22, 2015

Preface

Welcome to the Xevolver XML introductory tutorial. Xevolver XML (XevXML) is one of software products developed by the Xevolver project. The purpose of this project is to help migration of legacy HPC applications to new systems by improving their performance portabilities across system generations. Since a high priority is given to performance, an HPC application is often optimized and specialized for a particular HPC system. As a result, the performance is not portable to other systems. To make matters worse, such system-specific code optimizations are likely to be scattered over the application code. This is one main reason why HPC application migration is so painful. It is not affordable to reoptimize the whole code whenever a new system becomes available.

XevXML is developed for XML-based AST transformations to provide an easy way for user-defined code transformations. The current implementation of XevXML is built on top of the ROSE compiler framework. XevXML converts ROSE's AST to an XML document, and exposes it to programmers. So the programmers can use any XML-related technologies and tools to transform the AST. Then, the transformed AST is given back to the ROSE compiler framework so that the AST is unparsed to generate a transformed application code. Instead of directly modifying an application code, programmers can define their own code transformations to optimize the code for each system. System-specific optimizations are represented as XML translation rules, which can be defined separately from an application code. This leads to separation between application requirements and system requirements, expecting a lower migration cost of HPC applications to new systems.

Acknowledgments

The Xevolver Project is supported by JST CREST Research Area "Development of System Software Technologies for post-Peta Scale High Performance Computing" led by Dr. Yonezawa at RIKEN AICS and later by Dr. Sato at RIKEN AICS.

XevXML is under active development, and many people are being involved in the design and development.

Shoichi Hirasawa is a research associate employed by Tohoku University for the Xevolver Project. He has been actively working on writing translation rules and testing XevXML tools. That is, he is always the first user of the tools whenever new versions are committed to the code repository. His feedbacks are always helpful to improve the tools. Moreover, he is the main author of most rules in the transformation rule library. He designed the template of XSLT rules for AST transformation.

There are many people I would like to thank for their contributions:

- Contributing Collaborators:
  Reiji Suda (The University of Tokyo),
  Yasuharu Hayashi (NEC Corporation),
  Ryusuke Egawa (Tohoku University),
  Daisuke Takahashi (University of Tsukuba),
  Fumihiko Ino (Osaka University), and
  Kazuhiko Komatsu (Tohoku University)

- Students:
  Chunyan Wang (Tohoku University),
  Xiong Xiao (Tohoku University), and
  Daichi Sato (Tohoku University)

- Advisory Boards:
  Hiroaki Kobayashi (Tohoku University),
  Michael M. Resch (HLRS),
  Wenmei W. Hwu (UIUC), and
  Chisachi Kato (The University of Tokyo)

Contents

Chapter 1

Introduction

High-performance computing (HPC) system architectures are getting more complicated and diversified. Due to the system complexity, performance optimizations specific to processor architectures, system configurations, compilers, and/or libraries, called *system-specific optimizations*, are mandatory and becoming more important to exploit the potential of a particular system; an application code must be thoroughly optimized and specialized for one platform to achieve high performance. As a result, one HPC application often needs to have multiple versions, each of which is optimized in a different way for adapting to a particular platform. The diversity of system architectures increases the number of optimized versions required for performance portability across major platforms. To make matters worse, popular platforms can change drastically, and thus an application might need to be optimized not only for current platforms but also for future ones. Accordingly, an increase in system complexity and diversity would force a programmer to further invest enormous time and effort for HPC application development and maintenance.

XevXML [10] [11] is a code transformation framework that allows users to define their own code transformations, called *user-defined code transformations*. It exposes an abstract syntax tree (AST) to users so that they can apply any transformations to the AST. Transformation rules written in external files can be defined for individual systems, compilers, libraries, and so on. That is, code transformations representing system-aware optimizations can be separated from an application code. Hence, to achieve high performance, the users no longer need to specialize an application code itself for a particular platform.

XevXML assumes that an application code is annotated with a special mark, using directives and/or comments, and transformations are applied to the marked parts of the code. Note that the mark indicates "where to transform," but does not indicate "how to transform." The transformation rules that indicate how to transform the code are defined in external files. If system-aware code modifications are expressed as code transformations, users can express system-awareness separately from an application code.

As the name implies, XevXML employs eXtensible Markup Language (XML) [1] to represent an AST, i.e., an internal representation of code structures used by compilers. XML is a widely-used data format, and various XML-related technologies and tools have already been standardized and matured. Therefore, the users implement special code transformations by using only those standard tools. This chapter briefly describes an overview of code transformation with XevXML.

1.1 An overview of XevXML

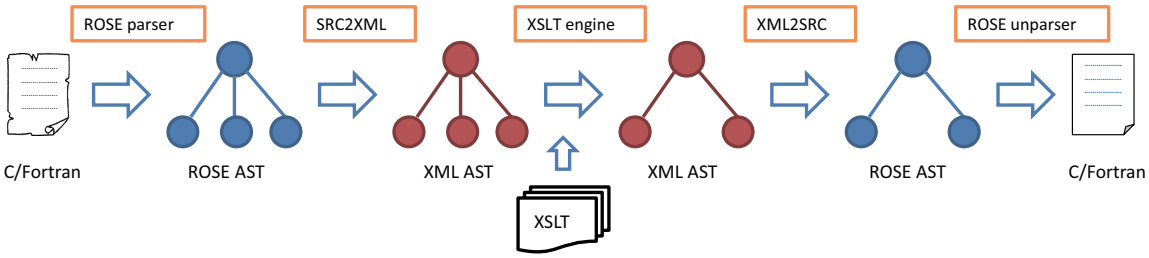

Figure 1.1: An overview of interconversion between ROSE ASTs and XML ASTs.

XevXML has so far been developed on top of the ROSE compiler infrastructure [3] [9]. XevXML provides the interconversion between an ROSE AST and its XML representation. Figure 1.1 shows an overview of the interconversion. XevXML converts a ROSE AST to an XML representation of the AST, called an *XML AST*. In XevXML, an XML AST is exposed to users. After user-defined transformations, the transformed XML AST is again converted back to a ROSE AST so that ROSE can unparse it to a C or Fortran code.

The interconversion is achieved by combining the following two commands, `src2xml` and `xml2src`.

NAME

`src2xml` – source-to-xml translator

SYNOPSIS

```
src2xml [OPTIONS] INPUT-FILE
```

DESCRIPTION

`src2xml` converts a C or Fortran code into an XML document. `src2xml` reads a code from the input file given by the command-line argument, and prints an output XML document to the standard output.

`-F, --check_fortran_pragma=`<**true/false**>
: enable the conversion of each Fortran pragma, e.g. `!$xev loog_tag`, to an <`SgPragmaDeclaration`> element. This conversion is enabled by default, and needs to be disabled to make the behavior identical to that of ROSE.

`-h, --help`
: print the usage.

`src2xml` also accepts some of ROSE command-line options such as `-rose:verbose`. A ROSE command-line option `-rose:skip_syntax_check` is automatically appended to the command-line options because it is required for some Fortran90 codes.

EXAMPLES

```
src2xml hello.c > hello.xml
```

This command will read `hello.c` and output its AST as an XML document to `hello.xml`.

NAME

`xml2src` – xml-to-source translator

SYNOPSIS

`xml2src [OPTIONS]`

DESCRIPTION

`xml2src` converts an XML AST to a C or Fortran code. Since the original language, C or Fortran, is recorded in an XML AST, `xml2src` generates a code written in the original language. `xml2xml` reads an XML AST from the standard input, and prints the generated code to the standard output. At present, command-line options for `xml2src` are simply ignored.

EXAMPLES

`xml2src < hello.xml > hello-again.c`
This command will read `hello.xml` and output its code to `hello-again.c`.

An XML AST is exposed to users. Thus, the users can apply any transformation to the AST. As an AST is represented as an XML document, any XML-related technologies and tools are available for the transformations. XevXML employs XML Stylesheet Language Transformation (XSLT) [6] [12] as the low-level interface to express the transformation rules of an XML AST. AST transformation is what compilers do internally for code transformation. Therefore, XevXML is capable of implementing various code transformations usually done by compilers.

XevXML can easily collaborate with ROSE. ROSE already has various features of code analyses and transformations to implement custom code transformation programs in C++. It must be painful if a user is required to reimplement those features from scratch for XevXML. So XevXML provides not only the above basic commands but also some C++ classes and functions, which are helpful to read and write XML ASTs, so that code transformation programs developed with ROSE can handle XML ASTs. Those classes and functions will be further described in Chapter 4.

If a code transformation is general enough and hence reusable in many applications, it could be implemented using either ROSE or XevXML. However, code transformations in practice could be application-specific, system-specific, domain-specific, and even programmer-specific. If such a code transformation program is implemented with ROSE, the user needs to maintain the program in addition to his/her application code. In XevXML, only transformation rules are defined declaratively, and code transformations are performed using standard XML tools. So the user does not need to develop his/her own program for applying the rules to application codes.

1.2 XML elements and attributes

Let's get started with a simple example, "Hello, World!" in C.

```
#include <stdio.h>
```

```
int main()
{
  printf("Hello, World!\n");
  return 0;
}
```

Using the `src2xml` command, the above code is converted to an AST of the following XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<SgSourceFile filename="hello.c" language="2" format="2">
  <SgGlobal>
    <SgFunctionDeclaration name="main"  end_name="0" >
        <SgTypeInt/>
      <SgFunctionParameterList/>
      <SgFunctionDefinition>
        <SgBasicBlock>
          <SgExprStatement>
            <SgFunctionCallExp>
              <SgFunctionRefExp symbol="printf" />
              <SgExprListExp>
                <SgCastExp mode="0" >
                    <SgPointerType base_type="SgModifierType" >
                      <SgModifierType modifier="const" >
                        <SgTypeChar/>
                      </SgModifierType>
                      <SgTypeChar/>
                    </SgPointerType>
                  <SgStringVal value="Hello, World!\n" paren="1"/>
                </SgCastExp>
              </SgExprListExp>
            </SgFunctionCallExp>
          </SgExprStatement>
          <SgReturnStmt>
            <SgIntVal value="0"  string="0" />
          </SgReturnStmt>
        </SgBasicBlock>
      </SgFunctionDefinition>
<PreprocessingInfo pos="2"  type="6" >
#include &lt;stdio.h&gt;

</PreprocessingInfo>
```

```
    </SgFunctionDeclaration>
  </SgGlobal>
</SgSourceFile>
```

The data format of XML ASTs is designed so that the interconversion between ROSE ASTs and XML ASTs becomes simple. In general, an XML document consists of XML elements and their attributes. In an XML AST, each XML element corresponds to a ROSE AST node. XML attributes of an XML element are used to keep the necessary information to restore the ROSE AST node. An XML AST looks like a text representation of a ROSE AST. In other words, XevXML provides another interface, XML, to handle ROSE AST nodes.

In the above XML document, the first line just indicates that the file is written in XML. The root node of an AST is the <SgSourceFile> element in the second line. The <SgSourceFile> element represents the whole C code. The <SgGlobal> element in the third line is a child node of the root node, and indicates the global scope of the C code. In the global scope, the `main` function is declared and defined. In the function body, the first statement is an expression statement, and the second statement is a `return` statement. Comments and preprocessor information such as `#include <stdio.h>` are written as strings within the `PreprocessingInfo` element.

XML attributes are used to restore ROSE AST nodes. For example, the <SgStringVal> element corresponds to a ROSE AST node of a `SgStringVal` class object representing a string. Thus, a string of `"Hello, World!\n"` is written as its `value` attribute. Similarly, the <SgFunctionRefExp> element is a reference to the name of a function, and the function name is given as the `name` attribute. Let's change "printf" is to "puts" by a text editor. Then, when the modified XML AST is converted to a C code (by using the `xml2src` command), the function call of `printf` will be changed to that of `puts`. This is a simple example to show that, in XevXML, XML data transformation results in AST transformation and thereby code transformation.

See the ROSE reference manual [4] to learn more about the definition of each AST node.

1.3 XML data transformation

XML data are texts, and various tools are thus available to modify an XML AST. As shown in Section 1.2, even a text editor can modify an XML AST. One may consider that, in the case of using a text editor, modifying the original code written in C/Fortran is much easier than modifying its XML AST. So why don't we directly modify the code? The answer is to avoid specializing the code for a particular platform.

In many cases, system-aware code optimizations assuming a particular target platform are necessary to exploit the system performance. A problem is that those optimizations are often harmful to the performance of another platform. A pragmatic approach is to maintain multiple versions of a code, each of which is optimized for a different platform. However, this results in degrading the maintainability and making legacy application migration more painful.

XevXML has been developed to replace code modifications with "mechanical transformations" of an XML AST. There are several benefits of the replacement. One important benefit is that the original code is not necessarily specialized for a particular platform. In other words, system-awareness is separated from an application code. This will be helpful to avoid maintaining multiple versions of an application code.

Another benefit is that expert knowledge about performance optimizations can be expressed in a machine-usable way. Basically, performance optimizations are very intellectual tasks that are often done on a case-by-case basis. However, focusing on a particular case, there are repetitive patterns in code modifications for performance optimizations. Thus, the code modifications can be replaced with a smaller number of mechanical code transformations.

The mechanical code transformations required instead of code modifications could be application-specific, system-specific, domain-specific, and even programmer-specific. Thus custom code transformations are often needed for special demands of individual cases. Therefore, XevXML has been developed for users to define their own code transformations in an easy way.

In XevXML, XSLT is currently employed to describe custom transformation rules of XML ASTs at the lowest abstraction level[1] In XSLT, XML data transformations are themselves written in XML. XSLT uses XPath expressions [5] [6], to define a pattern within a tree of XML elements and attributes. During the transformation process of XSLT, every XML element is visited in a depth-first manner. When a pattern is found at an XML element, the XML element is altered based on the rule associated with the pattern.

A simple XPath expression looks like a UNIX file path. In a UNIX file system, files and directories organize a tree structure. A file path is a text string to specify a location in the directory tree. There are two ways to point to the location of a file or a diretory. One is an absolute path, and the other is a relative path. If the string of a path starts with a slash, `/`, the path is an absolute path, otherwise it is a relative path. An absolute path is the path to a file or a directory from the root diretory. For example, the root directory is expressed by `/`, its sub-directory named "sub" is expressed by `/sub`, and a file named "xfile" that is located in the "sub" directory is expressed by `/sub/xfile`. Note that `/` represents the root directory and is also used as a delimiting character. On the other hand, a relative path indicates the path from the working directory where a user or an application is located. When the working diretory is `/sub`, a relative path to a `xfile` can be represented as `xfile`, `./xfile`, `../sub/xfile`, etc. Of course, those relative paths point to the same location because `.` and `..` denote the working diretory and its parent directory, respectively.

As well as a UNIX file path, an XPath expression also points to a location in an XML document. For example, the root of an XML document is denoted by `/`. A pattern in XML data is described by a combination of XPath expressions.

An example of XSLT rules for AST transformation are as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:exslt="http://exslt.org/common">

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="*">
```

[1]Several high-level interfaces for definition of code transformation rules are also under active development in the Xevolver project. One of such interfaces will be described in Chapter 2.

```
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="SgForStatement">
    <xsl:if test=".//*=SgForStatement">
    startLoopNest(); /* inserted */
    </xsl:if>
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
    </xsl:copy>
    <xsl:if test=".//*=SgForStatement">
    endLoopNest(); /* inserted */
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

The above XML file defines three rules, each of which is described within the <xsl:template> element. Based on these rules, an XML document is transformed to another XML document, called an output XML document.

The first rule matches the root of an XML document. The rule just invokes <xsl::apply-templates/> that by default dictates to visit all the sub-nodes and apply appropriate rules to them.

The second rule matches every XML element of an XML document, because its XPath expression is given by a wild-card operator, *. The rule is applied to an element unless a more specific rule matches the element. This rule simply copies the element and its attributes to the output XML document. The rule is recursively invoked because it invokes <xsl::apply-templates/>.

The third rule matches only an <SgForStatement> element. It checks if another <SgForStatement> element exists in the subtree of the matched element. Only if it exists, text data are inserted before and after the matched element.

If the above XSLT rules are applied to an XML AST, two function calls, `startLoopNest()` and `endLoopNest()`, are inserted before and after each nested loop, and a single loop is unchanged as shown below.

```
beginLoopNest(); /* inserted */
for(i=0;i<N;i++){
  for(j=0;j<M;j++){
    /* loop body 1 */
  }
}
```

```
endLoopNest();  /* inserted */

for(j=0;j<M;j++){
 /* loop body 2 */
}
```

This is an example of text insertion based on code pattern matching. Although this kind of rule is useful in practice, more advanced code transformations can be achieved by writing XSLT rules because XSLT can change the structure of an XML AST. Chapter 3 will describe how to write XSLT rules for AST transformation.

Although users can use any XSLT processor for code transformation, XevXML provides the `xsltexec` command for XSLT-based code transformation. In general, an XSLT file can import XSLT rules defined in other XSLT files. If a specified file written in an XSLT file for importing rules does not exist in the working directory, the `xsltexec` command looks for the file in the XevXML transformation library path specified by an environment variable, `XEVXML_LIBRARY_PATH`. The library path would usually be specified so that the command uses predefined XSLT rules offered by XevXML. The command also has a command line option to specify the library path.

NAME

`xsltexec` – a simple XSLT processor for XevXML

SYNOPSIS

`xsltexec [OPTIONS] XSLT-FILE`

DESCRIPTION

`xsltexec` applies XSLT rules to an XML representation of an AST, called an XML AST. XSLT rules are defined in a file given by the command-line argument. In the XSLT file, predefined XSLT rules offered by XevXML are also available without specifying the absolute paths of the XSLT files installed to the system.

The command reads an input XML AST from the standard input, and prints an output XML AST to the standard output. The command-line options are as follows.

`-L, --libdir=`<**path**>
: specify the library path.

`-h, --help`
: print the usage.

An environment variable, `XEVXML_LIBRARY_PATH`, is used to configure the default path of the transformation library.

EXAMPLES

`xsltexec sample.xsl < hello.xml`
This command will read an XML AST in `hello.xml` and transform the AST based on the

XSLT rules defined in `sample.xsl`. The transformed AST is printed out to the standard output.

1.4 Summary

This chapter describes an overview of the XevXML framework. Then, three basic commands, `src2xml`, `xml2src`, and `xsltexec`, are introduced for user-defined code transformations with XML specifications and tools. Using some simple examples, interconversion between ROSE ASTs and XML ASTs is explained, and also simple transformations are shown in this chapter.

Chapter 2

User-Defined Directives

In XevXML, transformation rules are defined by XSLT rules, and applied to an XML AST. By writing appropriate XSLT rules, XevXML as well as compilers can transform an AST in various ways. However, directly transforming an AST might be a too low-level approach for performance optimizations, especially if some optimization parameters need to be empirically configured in a try-and-error fashion as often seen in practical loop optimizations. Therefore, several high-level interfaces for XevXML are under development.

In practice, code modifications for loop optimizations are often expressed by combinations of well-known loop transformations. Moreover, text insertion and deletion based on code pattern matching are also frequently required for practical performance optimizations. Nonetheless, it is not very easy to correctly define their XSLT rules manually.

In this chapter, a high-level interface for user-defined code transformations is described. The interface is designed only for some specific purposes. Although the interface is less flexible than the XSLT approach of straightforwardly dictating AST transformations, it offers an easy way to quickly define a custom compiler directive that is associated with a composite of predefined rules. Such a directive can also be associated with text insertion and deletion.

The high-level interface is useful for generating a lot of various loop variants that are optimized with different loop transformation rules and parameters. Such loop variants are often required for so-called auto-tuning [8], which is automatic performance tuning based on empirical performance profiling. Besides, the high-level interface is also helpful for mechanically inserting some texts into a code, which are frequently required in practical performance optimizations.

2.1 XSLT rule generation

XevXML provides the `xsltgen` command that reads a simple JSON[2] file to generate XSLT rules, each of which is associated with one user-defined compiler directive. An XSLT rule generated by `xsltgen` is either a composite of predefined XSLT rules or text insertion/deletion. Although `xsltgen` is available only for these purposes, it provides a much easier way to define a custom compiler directive associated with such a rule.

NAME
`xsltgen` – XSLT rule generator

SYNOPSIS
`xsltgen [OPTIONS]`

DESCRIPTION
`xsltgen` converts a configuration file in JSON to an XSLT file of code transformation rules. `xsltgen` reads a JSON file from the standard input, and prints a XSLT file to the standard output.

EXAMPLES
`xsltgen < config.json > rules.xsl`
This command will read `test.json` and write XSLT rules to `rules.xsl`.

An example of a JSON file is as follows.

```
{
  "xev loop_tag1":{
    "target":"SgIfStmt",
    "insert-before":"!$test"
  }
  "xev loop_tag2":{
    "target":"SgFortranDo",
    "rules":[
      {"chillUnrollJam":{"loopName":"k","factor":4}},
      {"chillUnroll":{"loopName":"i","factor":2}}
    ]
  }
}
```

A JSON object enclosed in { and } is an unordered collection of any values. The root object whose { is in the first line contains two pairs of keys and values that are directive definitions. Notice that the colon character, `:`, is used to separate a key and its value. In a directive definition, the key indicates the directive name, and its value is a JSON object, called a rule definition, that defines the rule associated with the directive. In the above file, two compiler directives, `xev loop_tag1` and `xev loop_tag2`, are defined for Fortran. The pair whose key is `"target"` specifies what kind of statements the rule is applied. That is, the value is expected to be the AST node name of the statement that appears after the directive. If the directive name is `"*"`, the rule defined by its value is applied to the target AST node even if there is no directive attached to the statement.

The first directive, `xev loop_tag1`, assumes that it is followed by an `IF` statement. Then, it simply inserts a comment before the IF statement.

The second directive, `xev loop_tag2`, is associated with a composite of two predefined rules, `chillUnrollJam` and `chillUnroll`. This directive applies those loop optimization rules with the given parameters to the `DO` statement following the directive, which is an XML element of <`SgFortranDo`>. The predefined rules with their parameters are listed in an array of JSON, which is enclosed in `[` and `]`. Unlike an object of JSON, an array is an ordered list. The rules are applied in the same order as they appear in the array.

2.2 Loop optimization

The most time-consuming part of a scientific application is usually written as a loop, a so-called kernel loop. Thus, loop optimization is a key to improve the performance of such an application. There are a lot of loop optimization techniques, and most of them are supposed to be done by compilers. Typical loop optimization techniques are as follows.

- Loop unrolling
- Loop tiling
- Loop interchange
- Loop permutation
- Loop collapse
- Loop fusion
- Loop fission (aka. loop distribution)
- Unroll and jam (aka. outer loop unrolling)

In some cases, compilers are unable to perfectly apply those techniques to a kernel loop for various reasons. In such a case, manual optimizations of the kernel loop might be required to achieve high performance. However, even if the optimization is a certain combination of the techniques listed above, manual optimization of a kernel loop is not an easy task. This is because an appropriate combination of loop optimization techniques is unknown. In addition, most of loop optimization techniques have some parameters that need to be determined appropriately for high performance. To make matters worse, different platforms require different loop optimizations. Appropriate loop optimization, i.e., the combination and parameters, could change drastically depending on the target platform. Thus, manual code modification for loop optimization generally results in specializing the code only for a particular platform. Accordingly, the necessary information for loop optimizations should be separated from an application code.

Because of the importance, XevXML provides the `xsltgen` command as an easy way to compose loop optimization techniques and to associate the composite with a user-defined compiler directive. Code transformation rules for basic loop optimization techniques are predefined. The predefined rules in the XevXML transformation rule library are described in Section 3.1. Those predefined rules are themselves written in XSLT, and hence customizable for special demands of individual cases. A customized rule can also be used together with other predefined rules using the `xsltgen` command. Chapter 3 describes how to customize predefined rules and also how to define new rules.

2.3 Text insertion/deletion

The `xsltgen` command can define a custom compiler directive associated with not only AST transformations but also with text insertion and removal. Such a directive is useful to literally change some statements depending on the target platform. Although the C preprocessor can also achieve it using `#ifdef`, such an approach often makes an application code unmaintainable, so-called `#ifdef` hell. In the case of using `xsltgen`, the text to be inserted is written in an external file, which is an XSLT file generated by `xsltgen`. Therefore, the original code is not messed up with platform-dependent code fragments.

To define a directive for text insertion, a JSON object of the directive definition has one or more pairs whose keys are `"insert-before"`, `"insert-after"`, or `"replace"` and whose values are strings. As the names imply, `"insert-before"`, `"insert-after"`, and `"replace"` insert a text before the target statement, insert a text after the target statement, and replace the target statement with a text, respectively.

For example, the following JSON file will produce an XML rule that literally inserts function calls before and after a `for` loop in a C code.

```
{
  "*":{
    "target":"SgForStatement",
    "insert-before":"startLoop();",
    "insert-after":"endLoop();"
  }
}
```

In the above directive definition, the directive name is `*`. The rule does not need any directive in the code, and is applied to every `for` statement. More specifically, the rule is applied to the AST subtree whose root is `SgForStatement`. Therefore, the function call of `endLoop();` is inserted right after the loop body of every `for` loop.

If a pair of `"replace"` and an empty string is given in the directive definition, the directive will simply remove the target statement. Namely, such a directive can be used to remove the statement for a particular platform.

2.4 Summary

The `xsltgen` command is now under active development to provide a high-level interface for users to compose predefined loop optimizations. Although it is currently used only for some specific purposes, it offers a very easy way to define a user-defined code transformation associated with a custom directive. Therefore, it will be further extended to allow users to quickly define various kinds of code transformations.

Chapter 3

AST Transformation Rules

XevXML represents an AST in an XML format, and various XML tools are hence available for AST transformation. In XevXML, a transformation rule is (internally) represented as an XSLT rule. If a special code transformation is needed, the most flexible and expressive way is to write an XSLT rule for the transformation, even though high-level interfaces are also developed for some specific purposes as described in Chapter 2. An XSLT rule consists of XSLT template rules, each of which is applied to an XML element if the element matches a pattern of the XPath expression associated with the XSLT template rule.

XevXML provides some predefined XSLT template rules. Users can customize the predefined rules for their own purposes. Thus, an XSLT template rule can be used as a sample or a baseline for users to newly define their special transformation rules. This chapter describes how to use and customize an XSLT rule for code transformation.

3.1 Predefined Rules

Remember that, in XevXML, every transformation rule is internally written in XSLT. You can write such a rule from scratch if you want. But, a code transformation rule generally consists of many "XSLT template rules," and some of them are reusable in other code transformation rules. Therefore, XevXML offers some predefined XSLT template rules that can be used as a part of a user-defined code transformation rule.

Let's get started with a simple XSLT rule before explaining the predefined rules.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:exslt="http://exslt.org/common">

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>
```

```
  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="SgFortranDo">
    <xsl:if test="preceding-sibling::*[1]/SgPragma/@pragma='xev loop_tag'">
      !pragma is found
    </xsl:if>
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates mode="loopbody"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="*" mode="loopbody">
    ! in a loop body
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates mode="loopbody"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

As mentioned in Section 1.3, an XSLT rule usually consists of several "template" rules expressed by <xsl:template> elements. In the above example, four template rules are defined. Each template rule has a match attribute whose value is an XPath expression. When an XSLT rule consisting of multiple template rules is applied to an XML AST, every XML element in the XML AST, i.e. every AST node, is visited once in a depth-first manner. Then, if an XML element matches the XPath expression given by the match attribute of a template rule, the template rule is applied to the matched XML element.

In the above example, the XPath expression of the first template rule, i.e., /, matches only the root node of an XML AST that is an <SgSourceFile> element. Therefore, its rule is applied to the root node, and <xsl:apply-templates> is called for its child nodes.

The second template rule matches every node in the XML AST, i.e., *, and its rule is applied to the node unless a more specific rule matches the node. This rule also calls <xsl:apply-templates> for every child node.

The third template rule matches only an <SgFortranDo> element. This uses an <xsl:if> element to check if the XPath expression given by its test attribute is true. The expression is true only if

- The preceding sibling element of the <SgFortranDo> element has an <SgPragma> element

as a child node,

- The <SgPragma> element has a pragma attribute, and
- The attribute value is a string of "xev loop_tag".

If all the conditions are met, the template rule inserts a comment, "pragma is found," to the code before copying sub-nodes and attributes of the matched <SgFortranDo> element.

In the third template rule, an <xsl::apply-templates> element is used with a mode attribute. As a result, template rules with the same mode are applied to the child nodes. In this example, the fourth template rule is applied to every child node of an <SgFortranDo> element, for which all the above conditions are met. The fourth template rule instead of the second rule matches every node if <xsl::apply-templates> is used with the loopbody mode. In this way, different template rules can be applied only to a subset of AST nodes.

If the third rule and/or the fourth rule is modified so as to transform a loop structure and/or a loop body, the above XSLT rule can be considered as a loop transformation rule applied only to loops annotated with "xev loop_tag". Although such a loop transformation rule could be complex, its components, i.e. XSLT template rules, are reusable for many application codes. Therefore, XevXML provides a library of predefined XSLT template rules often required for basic loop optimizations.

In the template rule library, a code transformation rule is assumed to be comprised of three steps. One step is initialization, which usually finds an annotation being used to indicate where to transform. Another step is to move onto an XML element that is the root node of a subtree to be transformed. The other step is to transform the subtree, and print it out in XML. The library offers several predefined rules for each step.

Using such predefined template rules for each step, a special directive for unrolling Loop i can be defined as follow.

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="libCHiLL.xsl" />

  <xsl:output method="xml" encoding="UTF-8" />

  <xsl:template match="*" mode="xevInitHook">
   <xsl:apply-templates select="." mode="xevFindDirective">
      <xsl:with-param name="directiveName" select="'xev loop_tag'" />
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template match="*" mode="xevMoveHook">
    <xsl:apply-templates select="." mode="xevGoToLoop">
      <xsl:with-param name="loopName" select="'i'" />
    </xsl:apply-templates>
```

```
  </xsl:template>

  <xsl:template match="*" mode="xevTransformationHook">
    <xsl:apply-templates select="." mode="chillUnroll">
      <xsl:with-param name="factor" select="4" />
      <xsl:with-param name="loopName" select="'i'" />
    </xsl:apply-templates>
  </xsl:template>
</xsl:stylesheet>
```

The template rule with the `xevInitHook` mode represents the first step of the initialization. In the rule, <`xsl:apply-templates`> is used with the `xevFindDirective` mode. This results in using a predefined template rule associated with the `xevFindDirective` mode, and also finds a directive, `xev loop_tag`.

The template rule with the `xevMoveHook` mode represents the second step. In the rule, the `xevGoToLoop` mode is used when calling <`xsl:apply-templates`>. This results in visiting a loop whose index variable is `i`. The loop is to be transformed by the template rule of the third step.

The template rule with the `xevTransformationHook` mode represents the third step. In the rule, <`xsl:apply-templates`> is used with the `chillUnroll` mode for loop unrolling. The `chillUnroll` mode needs two parameters, `factor` and `loopName`. In this example, the third rule unrolls Loop i four times.

In this way, we can organize a custom compiler directive by directly using XevXML's low-level interface, i.e., XSLT.

In the current version of XevXML, all of predefined template rules are written for Fortran programs. It is ongoing to define such rules for C programs. The predefined rules currently included in the library are as follows.

Mode	Parameters	Description
Initialization rules		
`xevFindDirective`	`directiveName`	A compiler directive specified by the `directiveName` attribute is found by this rule.
Movement rules		
`xevGoToLoop`	`loopName`	The loop whose index variable is `loopName` is visited by using the rule.
`xevGoToVar`	`varName`	A variable reference to `varName` is visited by using the rule.
`xevSkipToNthLoop`	`loopName` `N`	The N-th loop from a loop whose index variable is given by `loopName` is visited by using the rule.
`xevGoToHere`	none	This is a dummy rule to do nothing at the second step of a code transformation.

(continue to next page)

(continued)

Mode	Parameters	Description
Basic loop optimization rules		
`xevLoopCollapse`	`firstLoop` `secondLoop`	Two loops are collapsed into a loop. Two parameters, `firstLoop` and `secondLoop`, specify the names of index variables of the loops to be collapsed.
`xevLoopFission`	none	A loop is broken into two loops. The first loop contains statements before a directive in the original loop body. The second loop contains the statements after the directive.
`xevLoopFusion`	none	Two consecutive loops are fused into a single loop. Statements in the two loops are moved into the body of the new loop.
`xevLoopInterchange`	none	Two consecutive loops are interchanged.
`xevLoopInversion`	none	A while loop body is inverted into a do while loop with a if statement.
`xevLoopReversal`	none	The order of a loop index is reversed.
`xevLoopSkewing`	none	The index of the inner loop of a loop nest is transformed into a new one that is dependent on the index of the outer loop.
`xevLoopStripMining`	`size` `factor`	The iteration of a loop is divided into two consecutive loops. `size` is the division size of the loop index.
`xevLoopTile`	`size1` `size2`	Two consecutive loops' iteration space is partitioned into blocks. `size1` is the block size of the outer loop and `size2` is the block size of the inner loop.
`xevLoopUnroll`	`loopName` `factor`	A loop is unrolled. `loopName` is the name of the index variable. `factor` is an unroll factor; every statement in the loop body is duplicated `factor` times.
`xevLoopUnswitching`	none	An conditional if statement in a loop body is moved outside of the loop.
CHiLL-compatible versions of optimization rules		
`chillFuse`	none	Two consecutive loops are fused into a single loop. Statements in the two loops are moved into the body of the new loop.

`chillPermute`	`firstLoop`
	`secondLoop`
	`thirdLoop`
		The order of up to three loops are changed. `firstLoop`, `secondLoop`, and `thirdLoop` are the names of index variables used by the loops to be permuted.

`chillSplit`	none	This is the CHiLL-compatible version of `xevLoopFission`.
`chillTile`	`size1` `size2`	Two consecutive loops' iteration space is partitioned into blocks. `size1` is the block size of the outer loop and `size2` is the block size of the inner loop.
`chillUnroll`	`loopName` `factor`	A loop in unrolled. `loopName` is the name of the index variable. `factor` is an unroll factor; every statement in the loop body is duplicated `factor` times.

(continue to next page)

(continued)

Mode	Parameters	Description
`chillUnrollJam`	`loopName` `factor`	The outer loop of a loop nest is unrolled. `loopName` is the name of the index variable. `factor` is an unroll factor; every statement in the loop body is duplicated `factor` times.

The above predefined template rules are basically designed under assumption of the three steps defined with the `xevInitHook`, `xevMoveHook`, and `xevTransformHook` modes. But we do not necessarily use the predefined rules for all of the three steps. If necessary, some steps can be manually described as in the first XSLT example of this section. Indeed, the `xsltgen` command currently uses predefined rules only for the third step. The XSLT template rules for the other two steps are written directly in individual XSLT files. See the XSLT file generated by the `xsltgen` command for details.

3.2 Custom Rules

This section is a step-by-step tutorial to define a custom XSLT rule.

Since XSLT is an expressive programming language, we can define an arbitrary XSLT template rule in various ways. But there is a frequently-used way to define an XSLT template rule for code transformation. One easy and typical way is summarized as follows.

Step 1. Write two versions of a code. One is the orginal version and the other is its translated version. Let C_{in} and C_{out} be the original version and the translated version, respectively. They can be considered as an input code example and an output code example of the code transformation to be defined below.

Step 2. Convert C_{in} and C_{out} to their XML representations, which are subtrees of XML ASTs, called a C_{in} subtree and a C_{out} subtree, respectively. The `src2xml` command can produce an XML AST of a code. Thus, those subtrees should appear in the XML AST if C_{in} and C_{out} are in the code.

Step 3. Write an XSLT template rule whose XPath expression matches the code fragment to be transformed. At this step, the XSLT template rule can be empty, i.e., the <`xsl:template`> element does not have child elements for writing XML elements into the output XML data. A macthed code fragment will be completely removed if the rule is empty.

Step 4. Copy the C_{out} subtree into the XSLT template rule. That is, the XML elements of the C_{out} subtree are simply used as child nodes of the <`xsl:template`> element. At this step, any code fragment matching the XSLT template rule is replaced with C_{out}.

Step 5. Generalize the rule so that some statements and/or expressions in the original code are copied to the translated code. By comparing the subtrees of C_{in} and C_{out}, we can consider which XML elements of C_{in} should be copied to the subtree of the output code.

Step 1. Write two versions of a code

In the following, an XSLT rule to translate a simple `DO WHILE` loop to its another version is defined as an example. The two versions are as follows. This loop transformation is so-called loop inversion [**?**]. Hereafter, the former version is called the original loop, and the latter is the target loop.

```
! original loop
do while(i<n)
  a(i) = 0
  i = i+1
end do
```

```
! target loop
if (i<n) then
  do
    a(i) = 0
    i = i+1
    if  (i<n)  cycle
    exit
  end do
endif
```

Step 2. Convert the loops to XML

By converting the target loop, we can get the following XML data, which is a subtree of an XML AST. If this subtree exists in an XML AST, the target loop (as is) appears in the output code when the XML AST is unparsed.

```
<SgIfStmt end="1" then="1">
  <SgExprStatement>
    <SgLessThanOp>
      <SgVarRefExp name="i" />
      <SgVarRefExp name="n" />
    </SgLessThanOp>
  </SgExprStatement>
  <SgBasicBlock>
    <SgFortranDo style="0" end="1" slabel="">
      <SgNullExpression />
      <SgNullExpression />
```

```
      <SgNullExpression />
      <SgBasicBlock>
        <SgExprStatement>
          <SgAssignOp>
            <SgPntrArrRefExp lvalue="1">
              <SgVarRefExp name="a" />
              <SgExprListExp>
                <SgVarRefExp name="i" />
              </SgExprListExp>
            </SgPntrArrRefExp>
            <SgIntVal value="0" string="0" />
          </SgAssignOp>
        </SgExprStatement>
        <SgExprStatement>
          <SgAssignOp>
            <SgVarRefExp name="i" lvalue="1" />
            <SgAddOp>
              <SgVarRefExp name="i" />
              <SgIntVal value="1" string="1" />
            </SgAddOp>
          </SgAssignOp>
        </SgExprStatement>
        <SgIfStmt end="0" then="0">
          <SgExprStatement>
            <SgLessThanOp>
              <SgVarRefExp name="i" />
              <SgVarRefExp name="n" />
            </SgLessThanOp>
          </SgExprStatement>
          <SgBasicBlock>
            <SgContinueStmt />
          </SgBasicBlock>
          <SgBasicBlock />
        </SgIfStmt>
        <SgBreakStmt />
      </SgBasicBlock>
    </SgFortranDo>
  </SgBasicBlock>
  <SgBasicBlock />
</SgIfStmt>
```

It is easy to get the XML data. If the original loop and the target loop are in a code, their subtrees appear in the XML AST of the code. The XML AST can be easily obtained by using the `src2xml` command. See Chapter 1 for details of the command.

Step 3. Write an initial XSLT template rule

The syntax of an XSLT template rule is as follows.

```
<xsl:template match=XXXX mode=ZZZZ>
   YYYY
</xsl:template>
```

Here, XXXX and YYYY must be correctly written to define an XSLT template rule. Roughly speaking, XXXX indicates what kind of an XML element is subject to this rule, and YYYY indicates how the XML element appears in the output XML data. For example, if we want to write a rule applied to every <SgWhileStmt> element, `XXXX` should be written as `"SgWhileStmt"`. If YYYY is empty, the matched XML element and its sub-nodes do not appear in the output XML data. ZZZZ is optinal but important for using predefined rules in XevXML. An XSLT template rule associated with the ZZZZ mode does not match XXXX unless it is invoked with the ZZZZ mode.

As shown in the first XSLT example in Section 3.1, we can write a rule so that it is applied only to an annotated code fragment. An "empty" XSLT template rule applied only to `DO WHILE` statements annotated with `!$xev loop_tag` is as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:exslt="http://exslt.org/common">

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="SgWhileStmt">
    <xsl:choose>
      <xsl:when test="preceding-sibling::*[1]/SgPragma/@pragma='xev loop_tag'">
        <!-- rule for annotated statements should be written here -->
      </xsl:when>
      <xsl:otherwise>
        <xsl:copy>
```

```
          <xsl:copy-of select="@*"/>
          <xsl:apply-templates/>
        </xsl:copy>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

Or, by using the predefined rules provided by XevXml, the above rule can be simplified as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:exslt="http://exslt.org/common">

  <xsl:import href="libCHiLL.xsl" />

  <xsl:template match="*" mode="xevInitHook">
   <xsl:apply-templates select="." mode="xevFindDirective">
      <xsl:with-param name="directiveName" select="'xev loop_tag'" />
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template match="*" mode="xevMoveHook">
    <xsl:apply-templates select="." mode="xevGoToLoop">
      <xsl:with-param name="loopName" select="'i'" />
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template match="*" mode="xevTransformationHook">
    <xsl:apply-templates select="." mode="myLoopInversion"/>
  </xsl:template>

  <xsl:template match="SgWhileStmt" mode="myLoopInversion">
    <!-- rule for annotated statements should be written here -->
  </xsl:template>
</xsl:stylesheet>
```

Since the fourth rule is invoked with the `myLoopInversion` mode, the rule is called the `myLoopInversion` rule. The above `myLoopInversion` rule will remove annotated `DO WHILE` statements because it is empty (only a comment is written now) and no XML element is hence output to the output XML

data. Therefore, in the following, the `myLoopInversion` rule is modified so that it outputs a transformed version of the original loop. The other template rules are unmodified.

Step 4. Copy the subtree of the target loop to the XML template rule

The comment in the `myLoopInversion` rule is replaced with the subtree obtained at Step 2. If this rule is applied to an XML AST, every annotated `DO WHILE` statement with its loop body is replaced with the target loop as is.

```
<xsl:template match="SgWhileStmt" mode="myLoopInversion">
  <SgIfStmt end="1" then="1">
    <SgExprStatement>
      <SgLessThanOp>
        <SgVarRefExp name="i" />
        <SgVarRefExp name="n" />
      </SgLessThanOp>
    </SgExprStatement>

    ... omitted ... (See Step 2)

  </SgIfStmt>
</xsl:template>
```

Step 5. Generalize the XSLT template rule

The `myLoopInversion` rule in Step 4 simply replaces an annotated `DO WHILE` loop with the target loop, and is usable only for loop inversion of the original loop (See Step 1). Therefore, we need to generalize the rule so that it can be reusable for loop inversion of other loops.

To generalize it, we usually need to copy some statements and expressions of the original loop to the target loop. One is the condition expression of the `DO WHILE` statement of the original loop, i.e., `(i < n)`. If a `DO WHILE` statement has a different condition expression, the expression should be used in the target loop. Nonetheless, the `myLoopInversion` rule in Step 4 always uses `(i < n)` as the condition expression of the transformed loop without repect to the condition expression of a `DO WHILE` loop to be transformed. Notice that the expression appears twice in the target loop (See Step 1). One is the condition of the first `IF` statement, and the other is the condition of the second `IF` statement.

See the subtree of the target loop in Step 2. The `myLoopInversion` rule in Step 4 is defined so as to output the subtree as is, and thus to always output `(i < n)`, which corresponds to the following <`SgExprStatement`> element and its sub-nodes.

```
...
<SgIfStmt end="1" then="1">
```

```
  <SgExprStatement>
    <SgLessThanOp>
      <SgVarRefExp name="i" />
      <SgVarRefExp name="n" />
    </SgLessThanOp>
  </SgExprStatement>
...
```

On the other hand, the condition of a `DO WHILE` statement corresponds to `SgWhileStmt/SgExprStatement` and its sub-nodes in an XML AST[1]. Since the `myLoopInversion` rule matches an <`SgWhileStmt`> element, the XPath expression of the condition is given by `./SgExprStatement` (relative path notation). Accordingly, the rule should be modified so that the <`SgExprStatement`> element specified by `./SgExpression` is copied to the output XML data.

```
...
<SgIfStmt end="1" then="1">
  <xsl:copy-of select="./SgExprStatement" />
...
```

To make the `myLoopInversion` rule reusable for other loops, we also need to copy all statements in the original loop body to the target loop body. See the subtree of the target loop in Step 2. Since the original loop body contains two expression statements, `a(i)=0` and `i=i+1`, the current `myLoopInvesion` rule always outputs these two statements in the target loop body, and thus is not reusable for other loops. The loop body is represented by the following <`SgBasicBlock`> element and its sub-nodes.

```
...
<SgBasicBlock>
  <SgExprStatement>
    <SgAssignOp>
      <SgPntrArrRefExp lvalue="1">
        <SgVarRefExp name="a" />
        <SgExprListExp>
          <SgVarRefExp name="i" />
        </SgExprListExp>
      </SgPntrArrRefExp>
      <SgIntVal value="0" string="0" />
    </SgAssignOp>
  </SgExprStatement>
```

[1]To figure out such a correspondence, it is helpful to convert the original loop to its XML representation.

```
  <SgExprStatement>
    <SgAssignOp>
      <SgVarRefExp name="i" lvalue="1" />
      <SgAddOp>
        <SgVarRefExp name="i" />
        <SgIntVal value="1" string="1" />
      </SgAddOp>
    </SgAssignOp>
  </SgExprStatement>
</SgBasicBlock>
...
```

On the other hand, the loop body of an `DO WHILE` statement is represented by `SgWhileStmt/SgBasicBlock` and its sub-nodes. Since the `myLoopInversion` rule matches an <`SgWhileStmt`> element, the XPath expression of the loop body is given by `./SgBasicBlock` (relative path notation). Accordingly, the `myLoopInversion` rule should be modified so that all of the sub-nodes of the <`SgBasicBlock`> element specified by `./SgBasicBlock` are copied to the output XML data.

```
...
<SgBasicBlock>
  <xsl:copy-of select="./SgBasicBlock/*" />
</SgBasicBlock>
...
```

Finally, a generalized `myLoopInversion` rule is obtained as follows.

```
<xsl:template match="SgWhileStmt" mode="myLoopInversion">
  <SgIfStmt end="1" then="1">
    <xsl:copy-of select="./SgExprStatement" />
    <SgBasicBlock>
      <SgFortranDo style="0" end="1" slabel="">
        <SgNullExpression />
        <SgNullExpression />
        <SgNullExpression />
        <SgBasicBlock>
          <xsl:copy-of select="./SgBasicBlock/*" />
          <SgIfStmt end="0" then="0">
            <xsl:copy-of select="./SgExprStatement" />
            <SgBasicBlock>
              <SgContinueStmt />
```

```
            </SgBasicBlock>
            <SgBasicBlock />
          </SgIfStmt>
          <SgBreakStmt />
        </SgBasicBlock>
      </SgFortranDo>
    </SgBasicBlock>
    <SgBasicBlock />
  </SgIfStmt>
</xsl:template>
```

The above rule is usable for loop inversion of not only the original loop but also other standard `DO WHILE` loops. In this way, we can gradually and incrementally generalize an XSLT template rule step by step so that the rule can cover a wider range of loops.

3.3 Summary

In XevXML, XSLT template rules for basic loop transformations are already predefined. The predefined rules will usually be used via the `xsltgen` command described in Chapter 2.

In practical performance optimizations, some systems, applications, application domains, and/or programmers may demand special code transformations, which are not predefined, for high performance, portability, maintainability, and so forth. In such a case, they can define their own code transformation rules for the special demands.

For defining a custom transformation rule, we first write a simple XSLT rule usable only for replacing a particular code to its target code, and then gradually generalize the rule so as to make it reusable for other codes. It is easy to write the initial simple rule by reference to an XML AST of the target code.

Chapter 4

Internal Structures and Behaviors

Since ROSE already has various features of code analyses and transformations useful to implement custom code transformation programs, XevXML provides some C++ classes and functions, which are helpful for such programs to read and write XML ASTs. As a result, code transformation programs developed with ROSE can handle XML ASTs.

This chapter describes XevXML classes and functions that allows ROSE to handle XML ASTs. By inheriting the classes, users can develop customized versions of `src2xml` and `xml2src` commands. For example, `src2xml` would be customized so that some additional XML attributes are written in an XML AST.

4.1 Utility functions

By using ROSE, it is easy to write an identity translator, i.e., the simplest translator.

```
/* identity.cpp */
#include <rose.h>

int main(int argc,char** argv){
  SgProject* sageProject=frontend(argc,argv);

  /* do something here for code transformation */

  return backend(sageProject);
}
```

The `frondend()` function reads a code and builds its AST. The `backend()` function passes the AST to the backend for unparsing and compilation.

XevXML offers a utility function, `XevConvertRoseToXml()`, for such a ROSE-based translator to print out a ROSE AST as an XML AST. All of the functions and classes provided XevXML are defined within a name space, `XevXml`.

For example, a simplified version of the `src2xml` command is as follows. It just prints out an XML AST, and then immediately ends.

```
/* simple.cpp */
#include <iostream>
#include <rose.h>
#include <xevxml.hpp>

int main(int argc,char** argv){
  SgProject* sageProject=frontend(argc,argv);

  /* print out an XML AST to the standard output */
  XevXml::XevInitialize();
  XevXml::XevConvertRoseToXml(std::cout,&sageProject);
  XevXml::XevFinalize();

  /* return backend(sageProject); */
  return 0; /* backend is not called in this example */
}
```

Run your compiler with appropriate options to specify the paths to necessary header files and libraries.

```
% g++ -I...  simple.cpp -o simple -L...  -lxevxml -lrose -lxalan-c -lxerces-c
```

The generated executable, `simple`, will print out an XML AST of a C or Fortran code to the standard output.

```
% ./simple input.c
<?xml version="1.0" encoding="UTF-8"?>
<SgSourceFile file="input.c" lang="2" fmt="2">
 <SgGlobal>
 ...omitted...
 </SgGlobal>
</SgSourceFile>
```

A simplified version of the `xml2src` command is as follows.

```
/* simple2.cpp */
#include <iostream>
```

```
#include <rose.h>
#include <xevxml.hpp>

int main(int argc,char** argv){
  SgProject* sageProject=0;

  /* read an XML AST from the standard input */
  XevXml::XevInitialize();
  if( XevXml::XevConvertXmlToRose(std::cin,&sageProject) == false ){
    std::cerr << " failed" << std::endl;
    abort();
  }
  else {
    /* check the AST (optional) */
    AstTests::runAllTests(sageProject);
    /* unparse the AST and print it out */
    XevXml::XevUnparseToStream(std::cout,&sageProject);
  }
  XevXml::XevFinalize();

  return 0;
}
```

The `XevConvertXmlToRose()` function reads an XML AST and converts it to a ROSE AST. If the conversion does not fail, the `XevUnparseToStream()` function is called to print the ROSE AST to a C++ stream. Accordingly, the above code, `simple2`, reads an XML AST from the standard input, and then unparses it. The unparsed code is printed out to the standard output.

The following combination of the above two commands works as an identity translator.

```
% ./simple input.c | ./simple2
```

```
#include <stdio.h>

int main()
{
  ...omitted...
  return 0;
}
```

The behaviors of the `XevConvertRoseToXml()` function can be changed using a `XevXmlOption` class object. For example, a slightly-modified vetsion of `simple.cpp` that uses the `XevXmlOption` is as follows.

```
/* simple-mod.cpp */
#include <iostream>
#include <rose.h>
#include <xevxml.hpp>

int main(int argc,char** argv){
  SgProject* sageProject=frontend(argc,argv);

  /* print out an XML AST to the standard output */
  XevXml::XevInitialize();
  XevXml::XevXmlOption opt;
  /* the following options are disabled by default */
  opt.getFortranPragmaFlag() = true; // parse Fortran pragmas
  opt.getPrintAddressFlag() = true; // print the address of every AST node

  /* execute the conversion with the above options */
  XevXml::XevConvertRoseToXml(std::cout,&sageProject,&opt);
  XevXml::XevFinalize();

  return 0;
}
```

In the above code, all the options disabled by default are enabled. By default, every "xev pragma" in a Fortran code, which starts with `!$xev`, is handled as a comment. By enabling the flag accessed by the `getFortranPragmaFlag()` method, such a pragma will be converted to an XML element. The other options are basically used for debugging.

4.2 Visitor classes

As described in the previous section, a C code whose file name is `input.c` will be converted to the following XML AST.

```
<?xml version="1.0" encoding="UTF-8"?>
<SgSourceFile file="input.c" lang="2" fmt="2">
 <SgGlobal>
 ...omitted...
 </SgGlobal>
</SgSourceFile>
```

ROSE uses C++ class objects, called *Sage III class objects*, to represent nodes of an AST. That is, every node of a ROSE AST is an object of a Sage III class. Each Sage III class is a subclass of

the SgNode class. For example, the SgGlobal class is one Sage III class, and its object represents the global scope of a code. On the other hand, every node of an XML AST is an XML element whose name is the same as the Sage III class name of its corresponding ROSE AST node. Hence, an <SgGlobal> element appears right after an <SgSourceFile> element, which corresponds to an SgSourceFile class object representing the source code. See the ROSE reference manual[4] for more details of Sage III classes. Chapter A also describes XML elements of Sage III classes currently supported by XevXML.

XML attributes are used to keep the necessary information to rebuild each ROSE AST. For example, <SgSourceFile> has a lang attribute so that an XML AST can record the language of the original source code. The attributes of XML AST nodes are described in Chapter A.

To customize the format of an XML AST, XevXML offers two internal C++ classes, XevSageVisitor and XevXmlVisitor. The former class traverses an AST of Sage III classes used in ROSE, and translates it to an XML AST. The latter traverses an XML AST to rebuild ROSE's AST.

The XevConvertRoseToXml() function internally uses the XevSageVisitor class for converting a ROSE AST to an XML AST. The XevSageVisitor class is a Vistor pattern class that visits every node of a ROSE AST in a depth-first fashion. Whenever XevSageVisitor visits an AST node, it writes an XML element whose name is the same as the Sage III class name of an AST node, e.g., SgGlobal. When writing XML attributes of each XML element, attribSg*() method is invoked (Sg* is the name of a Sage III class). Similarly, when writing the sub-nodes of each XML element, inodeSg*() method is invoked. Therefore, by overloading those methods, a user can customize XML documents generated by the XevSageVisitor class.

In the following code, a new attribute "foo" whose value is "bar" is added to an <SgGlobal> element.

```
/* custom.cpp */
#include <xevxml.hpp>

class MyVisitor: public XevXml::XevSageVisitor
{
public:
  MyVisitor():XevXml::XevSageVisitor() {}

  void attribSgGlobal(SgNode* node)
  {
    // print the default attributes
    XevXml::XevSageVisitor::attribSgGlobal(node);
    std::ostream& os = getOutputStream();

    os << " foo=\"str\"";
    os << " bar=\"1\"";
  }
};

int main(int argc,char** argv)
```

```
{
  SgProject* sageProject=frontend(argc,argv);
  MyVisitor visitor;
  XevXml::XevXmlOption opt;

  XevXml::XevInitialize();
  visitor.setXmlOption(&opt);
  visitor.write(std::cout,&sageProject);
  XevXml::XevFinalize();
  return 0;
}
```

The `MyVisitor` class is a subclass of `XevSageVisitor`. The `XevSageVisitor::setXmlOption()` method is used to pass an `XevXmlOption` object to the `MyVisitor` class object. After that, the `XevSageVisitor::write()` method is invoked to write an XML AST to a given `std::ostream` object.

The `XevSageVisitor::attribSgGlobal()` method is overloaded by `MyVisitor::attribSgGlobal()`. In the new method, the reference to an `std::ostream` object, which is given by the first argument of the `XevSageVisitor::write()` method, i.e., `std::cout`, is obtained by calling `getOutputStream()`. Then, two strings are passed to the object. As a result, the strings appear as XML attributes of an <`SgGlobal`> element.

```
  % ./custom input.c

<?xml version="1.0" encoding="UTF-8"?>
<SgSourceFile file="input.c" lang="2" fmt="2">
 <SgGlobal foo="str" bar="1">
 ...omitted...
 </SgGlobal>
</SgSourceFile>
```

The `XevConvertXmlToRose()` function internally uses the `XevXmlVisitor` class for converting an XML AST to a ROSE AST. The `XevXmlVisitor` class is a Vistor pattern class that visits every node of an XML AST in a depth-first fashion. Whenever `XevXmlVisitor` visits an XML element, `visigSg*()` method is invoked. Therefore, by overloading such a method, a user can customize the rebuilding process of an AST.

```
/* custom2.cpp */
#include <xevxml.hpp>
#include <xmlutils.hpp>
```

```
class MyXmlVisitor: public XevXml::XevXmlVisitor
{
public:
  SgNode* visitSgGlobal(xercesc::DOMNode* node, SgNode* parent)
  {
    SgNode* n =
      XevXml::XevXmlVisitor::visitSgGlobal(node,parent);

    std::string sval;
    if(XevXml::XmlGetAttributeValue(node,"foo",&sval)==true){
      std::cerr << " attribute \"foo\" is found" << std::endl;
      std::cerr << " the value is " << sval << std::endl;
    }
    else {
      std::cerr << " attribute \"foo\" is not found" << std::endl;
    }

    int ival;
    if(XevXml::XmlGetAttributeValue(node,"bar",&ival)==true){
      std::cerr << " attribute \"bar\" is found" << std::endl;
      std::cerr << " the value is " << ival << std::endl;
    }
    else {
      std::cerr << " attribute \"bar\" is not found" << std::endl;
    }

    float fval;
    if(XevXml::XmlGetAttributeValue(node,"baz",&fval)==true){
      std::cerr << " attribute \"baz\" is found" << std::endl;
      std::cerr << " the value is " << fval << std::endl;
    }
    else {
      std::cerr << " attribute \"baz\" is not found" << std::endl;
    }

    return n;
  }
};

int main(int argc,char** argv)
{
  SgProject* sageProject=0;
  MyXmlVisitor visitor;
  XevXml::XmlInitialize();
```

```
  visitor.read(std::cin,&sageProject);
  XevXml::XmlFinalize();
  return 0;
}
```

When an XML AST is traversed, every XML element is represented as an `xercesc::DOMNode` class object. Because of the depth-first AST traversal, its parent node is already converted to an `SgNode` object in many cases. The pointers to those two objects are given to `XevXmlVisitor::visitSg*()` as function arguments. If the `SgNode` object of the parent node is not available yet, the second argument is `NULL`.

`XevXmlVisitor::visitSg*()` has to return a pointer to an `SgNode` object, which is actually an `Sg*` object. For example, the `XevXmlVisitor::visitSgGlobal()` method is called when a <`SgGlobal`> element is visited. This method and its overloaded versions are expected to return a pointer to an `SgGlobal` object.

In the above code, `XevXmlVisitor::visitSgGlobal()` is overloaded by the `MyXmlVisitor` class so that a utility function, `XevXml::XmlGetAttributeValue()`, is called for checking if each attribute is given or not. Here, `XevXml::XmlGetAttributeValue()` is a C++ template function defined in `xmlutils.hpp`.

```
% ./custom input.c |./custom2
 attribute "foo" is found
 the value is str
 attribute "bar" is found
 the value is 1
 attribute "baz" is not found
```

4.3 Summary

XmlXML is an extensible code transformation framework. Its internal structures and behaviors can be customized if necessary. The utility functions and classes will be useful to develop tools for transformation, visualization, and analysis of an XML AST. Use of XML for representing an AST will by design be helpful to make those tools interoperable.

Chapter 5

Installation

5.1 Requirements

- ROSE compiler infrastructure – `http://rosecompiler.org/`
- Apache Xerces C++ 3.1.1 – `http://xerces.apache.org/`
- Apache Xalan C++ 1.0 – `http://xml.apache.org/xalan-c/`
- PicoJSON – `https://github.com/kazuho/picojson/`

5.2 Installation guide

1. First of all, some environment variables such as `LD_LIBRARY_PATH`, `JAVA_HOME`, and `CXX` must be correctly set so as to use ROSE, Xerces, and Xalan.
2. Create a new directory at the top diretocry for building the package.

   ```
   % mkdir mybuild
   % cd mybuild
   ```

3. Run the `cmake` command at the created directory to generate Makefile and copy necessary files.

   ```
   % cmake ../
   ```

   The `cmake` command accepts various options. For example, the install diretory is changed by using the `-DCMAKE_INSTALL_PREFIX` option.

   ```
   % cmake -DCMAKE_INSTALL_PREFIX=/usr/local/xevxml ../
   ```

If you need to use a specific version of a library or a header file, you can also use environment variables `CMAKE_LIBRARY_PATH` and `CMAKE_INCLUDE_PATH`. For example, if you need to use a library or a header file in `/home/user/local`, define those enviromnent variables as follows.

```
% export CMAKE_LIBRARY_PATH=/home/user/local/lib:$CMAKE_LIBRARY_PATH
% export CMAKE_INCLUDE_PATH=/home/user/local/include:$CMAKE_INCLUDE_PATH
```

Those environment variables must be correctly set so that all of the necessary header files and libraries such as `picojson.h` and `rose.h` are found by the `cmake` command.

See the `cmake` manual for more details [7].

4. Run the GNU make command.

   To compile the package,

   ```
   % make
   ```

   To install the package,

   ```
   % make install
   ```

   To test the package,

   ```
   % make test
   ```

   Some of tests will be failed. But it does not necessarily mean that something is wrong with the built binaries. Even if everything goes well, XevXML fails in some tests. This is mainly because XevXML is built on top of ROSE and unable to pass the tests if ROSE cannot properly parse/unparse the test codes.

Appendix A

XML elements and their attributes

A.1 Class hierarchy

```
SgNode -+- SgStatement   -+- SgDeclarationStatement
        |
        +- SgExpression  -+- SgBinaryOp
        |                 |
        |                 +- SgUnaryOp
        +- SgType         |
        |                 +- SgValueExp
        +- SgSupport
```

A.2 Statements

This section describes XML elements of `SgStatement` subclasses currently supported by XevXML.

A.2.1 XML elements

All XML elements of `SgStatement` subclasses are described below. If an XML element of a `SgStatement` subclass has its own XML attributes, the attributes are also listed in the description.

XML element name	Description
Declarations	
`SgAsmStmt`	asm statement (not tested) `code` (string): assembly code `volatile` (int): 0 (not volatile) or 1 (volatile)
`SgAttributeSpecificationStatement`	attribute specification (Fortran only) `kind` (int): See Section A.2.2.
`SgClassDeclaration`	declaration of a class or a structure `name` (string): class name

(continue to next page)

(continued)

XML element name	Description
	`type` (int): 0 (class), 1 (struct), or 2 (union)
`SgCommonBlock`	COMMON block (Fortran only)
`SgContainsStatement`	CONTAINS statement (Fortran only)
`SgDerivedTypeStatement`	derived type declaration (Fortran only)
	`name` (string): derived type name
	`type` (int): 0 (class), 1 (struct), or 2 (union)
`SgEntryStatement`	ENTRY statement (Fortran only)
	`name` (string): entry name
	`result` (string): result variable name
`SgEnumDeclaration`	enum declaration
	`name` (string): enum name
`SgEquivalenceStatement`	EQUIVALENCE statement (Fortran only)
`SgFormatStatement`	FORMAT statement (Fortran only)
`SgFortranIncludeLine`	INCLUDE (Fortran only)
	`filename` (string): filename name
`SgFunctionDeclaration`	function declaration
	`name` (string): function name
	`end_name` (int): `set_named_in_end_statement()`
`SgFunctionParameterList`	function parameter list
`SgImplicitStatement`	IMPLICIT statement (Fortran only)
`SgInterfaceStatement`	INTERFACE statement (Fortran only)
	`name` (string): interface name
	`type` (int): `SgInterfaceStatement::generic_spec_enum`
`SgModuleStatement`	MODULE statement (Fortran only)
	`name` (string): module name
	`type` (int): 0 (class), 1 (struct), or 2 (union)
`SgNamelistStatement`	NAMELIST statement (Fortran only)
`SgPragmaDeclaration`	pragma declaration
`SgProcedureHeaderStatement`	procedure declaration (Fortran only)
	`name` (string): procedure name
	`kind` (int): subprogram kind
	`result` (string): result variable name
	`pure` (int): 0 (not pure) or 1 (pure)
	`elemental` (int): 0 (not elemental) or 1 (elemental)
	`recursive` (int): 0 (not recursive) or 1 (recursive)
`SgProgramHeaderStatement`	program header (Fortran only)
	`name` (string): program name
	`elabel` (int): `set_end_numeric_label()`
`SgTypedefDeclaration`	typedef declaration
	`name` (string): typedef name
`SgUseStatement`	USE statement (Fortran only)
	`name` (string): module name
	`only` (int): 0 (no only option) or 1 (only option)

(continue to next page)

(continued)

XML element name	Description
SgVariableDeclaration	variable declaration name (string): variable name bitfield (int): bit field modifier (int): modifier
Other statements	
SgAllocateStatement	ALLOCATE statement (Fortran only)
SgArithmeticIfStatement	arithmetic IF statement (Fortran only)
SgBackspaceStatement	BACKSPACE statement (Fortran only) err (int): set_err() iostat (int): set_iostat() unit (int): set_unit()
SgBasicBlock	basic block
SgBreakStmt	break statement slabel (string): Fortran DO string label
SgCaseOptionStmt	case option of switch statement construct (string): case construct name
SgClassDefinition	class definition sequence (int): 1 for SEQUENCE ketword private (int): 1 for PRIVATE keyword abstract (int): 1 for ABSTRACT keyword
SgCloseStatement	CLOSE statement (Fortran only) err (int): set_err() iostat (int): set_iostat() status (int): set_status() unit (int): set_unit()
SgComputedGotoStatement	computed GO TO statement (Fortran only)
SgContinueStmt	continue statement slabel (string): Fortran DO string label
SgDeallocateStatement	DEALLOCATE statement (Fortran only)
SgDefaultOptionStmt	default option of switch statement construct (string): default construct name
SgDoWhileStmt	do-while statement
SgElseWhereStatement	ELSEWHERE statement (Fortran only)
SgEndfileStatement	ENDFILE statement (Fortran only) err (int): set_err() iostat (int): set_iostat() unit (int): set_unit()
SgExprStatement	statement of an expression
SgFlushStatement	FLUSH statement (Fortran only)
SgForAllStatement	FORALL statement (Fortran95)
SgForInitStatement	initial condition of for statement
SgForStatement	for statement
SgFortranDo	DO statement (Fortran only)

(continue to next page)

(continued)

XML element name	Description
	`end` (int): `set_has_end_statement()`
	`slabel` (string): `set_string_label()`
	`style` (int): `set_old_style()`
	`elabel` (int): `set_end_numeric_label()`
`SgFunctionDefinition`	function definition
`SgGlobal`	global scope
`SgGotoStatement`	goto statement
	`slabel` (string): string label
	`nlabel` (string): numeric label
`SgIfStmt`	if statement
	`end` (int): `set_has_end_statement()`
	`then` (int):`set_use_then_keyword()`
	`elabel` (int):`set_end_numeric_label()`
	`ellabel` (int):`set_else_numeric_label()`
`SgInquireStatement`	INQUIRE statemtent (Fortran only)
	*** To be described ***
`SgLabelStatement`	label statement
	`slabel` (string): string label
	`nlabel` (string): numeric label
`SgNullStatement`	empty statement (for(...);)
`SgNullifyStatement`	NULLIFY statement (Fortran only)
`SgOpenStatement`	OPEN statement (Fortran only)
	*** To be described ***
`SgPrintStatement`	PRINT statement (Fortran only)
	*** To be described ***
`SgReadStatement`	READ statement (Fortran only)
	*** To be described ***
`SgReturnStmt`	return statement
`SgRewindStatement`	REWIND statement (Fortran only)
	*** To be described ***
`SgStopOrPauseStatement`	STOP and PAUSE statements (Fortran only)
	`type` (int): 0 (unknown), 1 (stop), or 2 (pause)
`SgSwitchStatement`	switch statement
	`slabel` (string): string label
	`elabel` (string): end numeric label
`SgWhereStatement`	WHERE statement (Fortran only)
`SgWhileStmt`	while statement
	`end` (int): `set_has_end_statement()`
	`slabel` (string): `set_string_label()`
	`elabel` (string): `set_end_numeric_label()`
`SgWriteStatement`	WRITE statement (Fortran only)
	`err` (int): `set_err()`
	`fmt` (int): `set_format()`

(continue to next page)

(continued)

XML element name	Description

	`iostat` (int): `set_iostat()`
	`nml` (int): `set_namelist()`
	`rec` (int): `set_rec()`
	`unit` (int): `set_unit()`

A.2.2 XML attributes

All XML elements of `SgStatament` subclasses have `label` attribute to keep the numeric label number because Fortran allows a statement to have a numetic label.

In addition, XML elements of `SgDeclarationStatament` subclasses have the following attributes for modifiers.

- `declaration_modifier`

  In an XML AST, only one of the following flags can be specified by a decimal number to indicate the kind of a declaration.

  0x01 : unknown

  0x02 : default

  0x04 : friend

  0x08 : typedef

- `type_modifier`

  Multiple flags might be combined by using logical OR. The value of this XML attribute (combination of the following flags) is written as a decimal number to indicate the type information.

  0x00001 : unknown value (error)

  0x00002 : unknown value (default)

  0x00004 : restrict qualifier (for C/C++)

  0x00008 : allocatable attribute specifier (for Fortran 90)

  0x00010 : asynchronous attribute specifier (for Fortran 2003)

  0x00020 : bind attribute specifier (for Fortran 2003)

  0x00040 : data attribute specifier (for Fortran 77)

  0x00080 : dimension attribute specifier (for Fortran 77)

  0x00100 : intent(in) attribute specifier (for Fortran 90)

  0x00200 : intent(out) attribute specifier (for Fortran 90)

  0x00400 : intent(inout) attribute specifier (for Fortran 90)

  0x00800 : intrinsic attribute specifier (for Fortran 90)

  0x01000 : optional attribute specifier (for Fortran 90)

  0x02000 : optional attribute specifier (for Fortran 90)

0x04000 : optional attribute specifier (for Fortran 90)

0x08000 : save attribute specifier (for Fortran 77)

0x10000 : target attribute specifier (for Fortran 90)

0x20000 : value attribute specifier (for Fortran 2003)

- `cv_modifier`

  Must be either const, volatile, or neither.

  0x1 : unknown value (error)

  0x2 : default value

  0x4 : constant qualifier

  0x8 : volatile qualifier

- `access_modifier`

  Only one of the following flags can be specified by a decimal number to indicate the accessibility.

  0x01 : error value

  0x02 : private access (local to class members)

  0x04 : protected access (local to class members and membrs of derived classes)

  0x08 : public access (access within enclosing namespace)

  0x10 : default value (public access)

  0x20 : fortran default value

- `storage_modifier`

  Only one of the following flags can be specified by a decimal number to location or properties of declarations.

  0x01 : error value

  0x02 : default value

  0x04 : extern storage modifier

  0x08 : static storage modifier

  0x10 : auto storage value

  0x20 : (not used)

  0x40 : register storage modifier

  0x80 : mutable storage modifier

- `thread_local`

  0 or 1. If 1 is given, the variable is thread-local.

The <`SgAttributeSpecificationStatement`> element requires the `kind` attribute below.

- `kind`

  One of the following values can be specified (Fortran).

  0 : unknown
  1 : private
  2 : public
  3 : allocatable
  4 : asynchronous
  5 : bind
  6 : data
  7 : dimension
  8 : external
  9 : intent
  10 : intrinsic
  11 : optional
  12 : parameter
  13 : pointer
  14 : protected
  15 : save
  16 : target
  17 : value
  18 : volatile

In the case of `kind=9̈`, the `intent` attribute is required to be set to 600 (IN), 601 (OUT), or 602 (INOUT)

A.3 Expressions

This section describes XML elements of `SgExpression` subclasses currently supported by XevXML. XML elements of most `SgExpression` subclasses have `paren` and `lvalue` attributes whose values are integers. Their values are given to `SgExpression::set_need_paren()` and `SgExpression::set_lvalue()`, respectively. If an XML element of a `SgExpression` subclass has its own XML attributes, the attributes are also listed in the description.

XML element name	Description
Binary operators	

(continue to next page)

(continued)

XML element name	Description
SgAddOp	+ operator (addition)
SgAndAssignOp	&= operator (bitwise AND assignment)
SgAndOp	&& operator (logical AND)
SgArrowExp	-> operator (sturcture derefence (pointer))
SgAssignOp	= operator (assignment)
SgBitAndOp	& operator (bitwise AND)
SgBitOrOp	| operator (bitwise OR)
SgBitXorOp	^ operator (bitwise XOR)
SgCommaOpExp	, operator (comma)
SgConcatenationOp	Fortran // operator (string concatenation)
SgDivAssignOp	/= operator (division assignment)
SgDivideOp	/ operator (division)
SgEqualityOp	== operator (equal to)
SgExponentiationOp	Fortran ** operator (exponential)
SgGreaterOrEqualOp	>= operator (greater than or equal to)
SgGreaterThanOp	> operator (greater than)
SgIorAssignOp	|= operator (bitwise OR assignment)
SgLessOrEqualOp	<= operator (less than or equal to)
SgLessThanOp	< operator (less than)
SgLshiftAssignOp	<<= operator (bitwise left shift assignment)
SgLshiftOp	<< operator (bitwise left shift)
SgMinusAssignOp	-= operator (subtraction assignment)
SgModAssignOp	%= operator (modulo assignment)
SgModOp	% operator (modulo)
SgMultAssignOp	*= operator (multiplication assignment)
SgMultiplyOp	* operator (multiplication)
SgNotEqualOp	!= operator (not equal to)
SgOrOp	|| operator (logical OR)
SgPlusAssignOp	+= operator (addition assignment)
SgPointerAssignOp	Fortran => operator (pointer assignment)
SgPntrArrRefExp	array subscript (a[b])
SgRshiftAssignOp	>>= operator (bitwise right shift assignment)
SgRshiftOp	>> operator (bitwise right shift)
SgSubtractOp	- operator (subtraction)
SgUserDefinedBinaryOp	user defined binary operator name (string) :operator name
SgXorAssignOp	^= operator (bitwise XOR assignment)
Unary operators	
SgAddressOfOp	& operator (address)
SgBitComplementOp	~operator (bitwise NOT)
SgMinusMinusOp	-- operator (decrement) mode (int): 0 (prefix) or 1 (postfix)
SgMinusOp	Unary - operator (minus)

(continue to next page)

(continued)

XML element name	Description
	`mode` (int): 0 (prefix) or 1 (postfix)
`SgNotOp`	! operator (logical NOT)
`SgPlusPlusOp`	++ operator (increment)
	`mode` (int): 0 (prefix) or 1 (postfix)
`SgUnaryAddOp`	Unary + operator (plus)
`SgUserDefinedUnaryOp`	user defined unary operator
	`name` (string) :operator name
Values	
`SgBoolValExp`	boolean value
	`value` (int)
`SgCharVal`	character
	`value` (char)
	`string` (string): the value as a string
`SgComplexVal`	complex number value
`SgDoubleVal`	double-precision float value
	`value` (double)
	`string` (string): the value as a string
`SgEnumVal`	enum value
	`value` (int)
`SgFloatVal`	float value
	`value` (float)
	`string` (string): the value as a string
`SgIntVal`	integer value
	`value` (int)
	`string` (string): the value as a string
`SgLongDoubleVal`	long double value
	`value` (long double)
	`string` (string): the value as a string
`SgLongIntVal`	long int value
	`value` (long int)
	`string` (string): the value as a string
`SgLongLongIntVal`	long long int value
	`value` (long long int)
	`string` (string): the value as a string
`SgShortVal`	short int value
	`value` (short int)
	`string` (string): the value as a string
`SgStringVal`	a string literal
	`value` (string)
	`single` (int): 0 (double quote) or 1 (single quote)
`SgUnsignedCharVal`	unsigned character
	`value` (unsigned char)
	`string` (string): the value as a string

(continue to next page)

(continued)

XML element name	Description
SgUnsignedIntVal	unsigned int value value (unsigned int) string (string): the value as a string
SgUnsignedLongLongIntVal	unsigned long long int value value (unsigned long long int) string (string): the value as a string
SgUnsignedLongVal	unsigned long int value value (unsigned long int) string (string): the value as a string
SgUnsignedShortVal	unsigned short int value value (unsigned short int) string (string): the value as a string
SgWcharVal	whar_t value value (unsigned short) string (string): the value as a string
Other expressions	
SgAggregateInitializer	initialization with aggregated data (struct complex x={1,2}) implicit (int): 0 (need explicit braces) or 1 (not need)
SgAssignInitializer	initialization (i=0)
SgAsteriskShapeExp	Fortran * expression (write(*,*))
SgCastExp	type cast ((int*)x) ctype (int): cast type (set_cast_type()) implicit (int): 0 (explicit cast) or 1 (implicit cast)
SgColonShapeExp	Fortran colons for separating array dimensions (x(0:1))
SgConditionalExp	ternary conditional (a?b:c)
SgConstructorInitializer	initialization with a constructor (x=myclass(0,1)) name (int): set_need_name(). qual (int): set_need_qualifier(). paren_after_name (int): set_need_parenthesis_after_name(). unknown (int): set_associated_class_unknown().
SgDotExp	structure member access (x.y)
SgExprListExp	list of expressions
SgFunctionCallExp	function call (f())
SgFunctionRefExp	function reference name (string): function name. kind (int): 1(function(default)) or 2(subroutine) (for Fortran)
SgImpliedDo	Fortran implied loop expression ((/ /))
SgLabelRefExp	label refernce nlabel (int): numeric label value type (int): label type
SgNullExpression	null expression
SgPointerDerefExp	pointer dereference (*x, x is a pointer)
SgSizeOfOp	sizeof operator (sizeof(int))

(continue to next page)

(continued)

XML element name	Description
SgSubscriptExpression	Fortran subscript expressions
SgVarArgEndOp	end of vararg
SgVarArgOp	variable length list of arguments
SgVarArgStartOp	beginning of vararg
SgVarRefExp	variable reference name (string): variable name

A.4 Types

This section describes XML elements of SgType subclasses currently supported by XevXML. XML elements of most SgType subclasses have the kind attribute whose value is an integer. The attribute value is used as a type kind paremeter used in Fortran.

XML element name	Description
SgArrayType	array type
SgClassType	class type
SgEnumType	enum type
SgFunctionType	function type
SgModifierType	modifiers
SgPointerType	pointer type
SgTypeBool	boolean type
SgTypeChar	character type
SgTypeComplex	complex type (in C99, double _Complex x;)
SgTypeDefault	unknown type (internal use)
SgTypeDouble	double-precision floating-point type
SgTypeEllipse	ellipse type (a variable number of parameters, "...")
SgTypeFloat	single-precision floating-point type
SgTypeImaginary	imaginary type (in C99, float _Imaginary x;)
SgTypeInt	integer type
SgTypeLong	long integer type
SgTypeLongDouble	long double type
SgTypeLongLong	long long integer type
SgTypeShort	short integer type
SgTypeSignedChar	signed character type
SgTypeSignedInt	signed integer type
SgTypeSignedLong	signed long integer type
SgTypeSignedLongLong	signed long long integer type
SgTypeSignedShort	signed short integer type
SgTypeString	string type
SgTypeUnsignedChar	unsigned character type
SgTypeUnsignedInt	unsigned integer type
SgTypeUnsignedLong	unsigned long integer type

(continue to next page)

(continued)

XML element name	Description
SgTypeUnsignedLongLong	unsigned long long integer type
SgTypeUnsignedShort	unsigned short integer type
SgTypeVoid	void type
SgTypedefType	typedef type

A.5 Other elements

XML element name	Description
SgDataStatementGroup	a group of data statement objects (Fortran only)
SgDataStatementObject	data statement object (Fortran only)
SgDataStatementValue	data statement value (Fortran only) fmt (int): 0(unknown), 1(default), 2(explicit), 3(implicit), or 4(implied do)
SgFormatItem	item of FORMAT statement (Fortran only) fmt (string): format data single (int): non-zero to use single quotes double (int): non-zero to use double quotes
SgFunctionParameterTypeList	type lists of function parameters
SgNameGroup	group of names for SgNamelistStatement (Fortran only) group (string) : group name names (string) : comma-separated names (e.g. "a,b,c,...")
SgPragma	prama in a pragma declaration pragma (string): string(s) of a pragma
SgSourceFile	source file (root elemement of an XML AST) file (string): filename of the original code lang (int): 0 (error), 1 (unknown), 2 (C), 3 (C++), or 4 (Fortran) fmt (int): 0 (unknown), 1 (fixed), or 2 (free) (for Fortran)
SgTypedefSeq	list of typedefs, for which the current SgType is the base type
SgCommonBlockObject	object in a COMMON block (Fortran only) name (string): name of the COMMON block object
SgInitializedName	initialized name (declared symbols) name (string): name of the symbol prev (string): previous symbol for cray pointer
SgInterfaceBody	body of INTERFACE statement (Fortran only) name (string): name of the INTERFACE function
SgRenamePair	alias of a variable defined USE statement (Fortran only) lname (string): local name uname (string): use name

Bibliography

[1] Extensible Markup Language (XML) 1.1 (Second Edition). `http://www.w3.org/TR/xml11/`.

[2] Introducing JSON. `http://www.json.org/`.

[3] ROSE compiler infrastructure. `http://rosecompiler.org`.

[4] ROSE web reference. `http://rosecompiler.org/ROSE_HTML_Reference/index.html`.

[5] XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath/`.

[6] Michael Kay. *XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer)*. Wrox Press Ltd., 4 edition, 2008.

[7] Kitware. CMake. `http://www.cmake.org/`.

[8] Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda, editors. *Software Automatic Tuning – from concepts to state-of-the-art results*. Springer, 2010.

[9] Dan Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[10] Hiroyuki Takizawa, Shoichi Hirasawa, Yasuharu Hayashi, Ryusuke Egawa, and Hiroaki Kobayashi. Xevolver: An XML-based code translation framework for supporting HPC application migration. In *IEEE International Conference on High Performance Computing (HiPC)*, 2014.

[11] Hiroyuki Takizawa, Shoichi Hirasawa, and Hiroaki Kobayashi. Xevolver: An XML-based programming framework for software evolution. Poster presentation at *Supercomputing 2013 (SC13)*, 2013.

[12] D. Tidwell. *XSLT*. O'Reilly Media, 2008.