Document-Oriented Source Code Transformation using XML

Michael L. Collard, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
collard@cs.kent.edu, jmaletic@cs.kent.edu

Abstract

The paper takes a document-oriented view of source-code transformation and describes how an underlying XML representation for source code can be used to support refactorings. The method preserves all documentary structure of the source and is applied in a straightforward manner.

1. Introduction

We view the transformation of source code, for the purpose of software evolution, as a heavily document-oriented activity. Transformations, such as refactoring, are a mapping from source-code documents to source-code documents. Automating these types of transformations is integral to supporting the evolutionary process since this can be a time consuming and error prone task if done manually. However, the final result of the transformation must be in the form readable by the developer for it to be useful for continued evolution. A successful software-evolution transformation must easily support all aspects of the document including its lexical, structural, syntactic, and documentary nature.

The transformation must preserve the programmer's view of the document. Not doing so may mean the rejection of the system, as described in a number of studies on real world projects [4, 8]. In [4] examples are given on large projects where any potential changes to the system had to be presented to the programmers in the exact view of the source code that they were familiar with. If not, the proposed changes were rejected.

In [8] the concept of the *documentary structure* of source code, whose elements include all white space and comments, is presented. It is described as what a programmer places in the source code for the sole purpose of assisting whoever is reading the program. Examples given show that white space, such as line breaks and indentation can be more important than comments and that the notion of a single comment is not well defined.

This documentary structure is often at odds with the linguistic structure of the program. Unfortunately for many parse-tree-based approaches, this documentary structure is completely lost. Attempts to preserve these ties often result in the documentary structure not being easily integrated back into the representation. There are exceptions to this problem however and one notable example is the DMS systems by Baxter [1]. The DMS project has gone to great lengths to address this specific issue by storing important textual items within the underlying abstract-syntax graph.

Any transformation approach for software evolution must preserve documentary structure and must allow it to be a first-class part of the document. A successful analysis and transformation system will allow a combination of changes to any of the text, whether or not it is part of the formal syntactic structure.

In contrast to these requirements, software-development tools typically take a totally compiler-centric approach of representing the source code as a syntax tree according to the formal linguistic structure of the program. It has been observed that these compiler-centric approaches are often not a good match to the problems that they are trying to solve [6, 8].

Our approach takes a very document-oriented XML approach to the transformation of source code. We developed a robust parser to markup C++ source code in a lightweight representation. This allows us to leverage XML technologies for tasks such as fact extraction and transformation.

In order to realize document-oriented source-code transformation, a sophisticated underlying infrastructure is necessary. As such, the approach is built on top of an XML representation of the source code developed to support program-analysis tasks, namely srcML [2, 3, 7]. This representation explicitly embeds high-level syntactic information within the source code in such a way that it does not interfere with the textual/documentary context of the source code. The representation is unique in that it preserves the programmer's view of the source code while at the same time explicitly adding parts of the

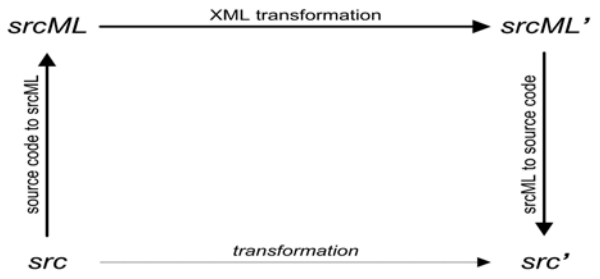

Figure 1. With srcML source-code transformations, such as refactoring, are raised to the level of XML transformations. The source code is translated into equivalent srcML, the srcML undergoes XML transformation, and the resulting srcML is translated back to source code.

abstract syntax to the source. srcML directly supports such tasks as lightweight fact extraction, source-code transformations, and the integration of source models (e.g., call graphs) using XML tools and standards. It also supports the embedding of meta-information into the source (e.g., hyperlinks).

We will describe our underlying approach and demonstrate via a simple example how it can be used to transform source code (i.e., refactor) in a document and syntactic preserving manner. Lastly we discuss our future plans and direction for this research.

2. Transformation of Source Code

srcML is an XML representation of the complete contents of a source-code file. All text (including white space) from the original source-code file is preserved in its original ordering. XML meta-characters, including '<', '>', and '&' are stored in an escaped form, i.e., '<', '>', and '&' respectively. A special representation is used for the form feed character that is not able to be directly represented or escaped in XML. Per the XML standard, line ends in the source-code file are normalized to a single character. Generating the original contents of the source-code file from the srcML representation is straightforward.

The XML elements of srcML surround the text that they describe including syntactic structures e.g., elements *class*, *function*, *if*, *while*, etc., documentary structure, e.g., element *comment*, and preprocessor statements, e.g., element *cpp:directive*. The srcML markup stops at the expression level with elements for identifiers (element *name*) and function calls (element *call*).

The srcML representation is relatively compact with a 5 to 7 times increase in file size over the source-code document. A robust source code to srcML translator exists with the ability to translate at over 7500 LOC per second. The translator currently handles the C/C++ languages. With srcML, XML transformations can be used to perform non-intrusive transformations on source code. Transformation can be performed on any selected element of the syntactic or documentary structure, while simultaneously preserving all other elements.

The transformation process is shown in Figure 1. The source-code document, *src*, is converted by the srcML translator into an equivalent srcML document, *srcML*. XML transformations of the srcML document convert *srcML* to *srcML'*. Afterwards, the transformed srcML document, *srcML'*, is translated back to a source-code document, *src'*.

We carefully define srcML so that srcML documents can be transformed using any desired XML transformation approach, e.g., SAX, DOM, XSLT. This is different than a *data-oriented* approach (e.g., GXL) that has little or no connection to the original document.

A srcML transformation can be performed non-intrusively. All elements of the source-code document, including documentary structure, such as white space, preprocessor directives, etc., can be passed unaltered through the transformation. An identity XML transformation at the srcML level, i.e., *srcML* is identical to *srcML'*, is equivalent to an identity transformation at the textual level, i.e., *src* is identical to *src'* (with the possible exception of a change in end-of-line character). Only the source-code elements that require changes are altered in any way. Of special note is the preservation of white space. Some XML processors consider white space in elements insignificant and by default normalizes them. However, our experience has found that careful use of these tools allows the process to preserve these elements.

A specific transformation of current interest is refactoring, a source-to-source transformation that preserves program behavior. The purpose of refactoring is to improve the overall structure of the code so that it can be more easily extended, repaired, and to increase comprehension.

Current refactoring tools have difficulty with the preservation of documentary structure, and in languages such as C++, only a limited number of refactorings from the catalog are supported with automation. In addition, tools that can perform refactorings are very carefully constructed for specific refactorings. Most tools cannot be easily enhanced or extended, and they do not provide a general purpose format and/or language that can be used by a programmer to adapt them to their own use, or to write additional transformations [8].

The rest of this section will demonstrate a non-intrusive refactoring of source code via using an XML transformation. The example shown is a refactoring from Fowler's [5] titled "*Replace Nested Conditional with Guard Clauses*". This refactoring replaces a conditional statement wrapped around normal (i.e., non-error)

processing with a guard clause placed before normal processing. The guard clause performs an early return from the method when an error occurs. Fowler argues for the removal of the nested conditional because it hinders the comprehension of the normal execution path. An example of a function undergoing the refactoring is shown in Figure 2 and Figure 3.

In order to perform the refactoring an XSLT program was written. The XPath template matching of XSLT is particularly well suited to XML document transformation. We will now describe important parts of the XSLT program.

In order to limit changes only to the parts of the document that is desired we use a default copy template so that, unless otherwise specified, the elements and text will go through the refactoring unaltered. This includes all elements and text. Doing so preserves all documentary structure, statements, and preprocessor statements.

```
int factorial(int n) {

  // factorial value
  int product = -1;

  // check for proper values
  if (is_non_negative(n)) {

    // calculate factorial
    product = 1;
    int i = 1;
    while (i <= n) {

      // update the product
      product *= i;

      // next value
      ++i;
    }
  }

  //now the factorial
  return product;
}
```

Figure 2. The original function uses a nested conditional to check the status of the input parameter.

The first issue is how to identify the nested conditional that will be replaced. srcML allows the formation of XPath expressions to refer to particular statements in the source code. For this refactoring a specific parameter, *$cname*, was provided to the XSLT program. The main XSLT template matches the `if`-statement that forms the nested conditional using the XPath expression:

```
src:if[src:condition/src:expr//src:name=$c
                    name]
```

This expression matches all *src:if* elements whose condition contains the same name as the parameter *$cname*. This parameter contains the name of a function call that this example uses to identify a nested conditional. After matching the nested conditional the main template generates the guard clause and converts the former contents of the guard clause so that they are at the top level of the function. In addition, it fixes the indentation on these contents so that their indentation lines up with the rest of the program. We will now look at how a document view can be used to support these two parts of the transformation.

The main template uses another template, with the mode attribute "guard", to construct the guard clause. An *if*-statement embeds the condition from the nested conditional and inserts a *return*-statement as below:

```
            if (!<xsl:value-of
    select="src:condition/src:expr"/>)
                  return -1;
```

```
int factorial(int n) {

  // factorial value
  int product = -1;

  // check for proper values
  if (!is_non_negative(n))
    return -1;

  // calculate factorial
  product = 1;
  int i = 1;
  while (i <= n) {

    // update the product
    product *= i;

    // next value
    ++i;
  }


  //now the factorial
  return product;
}
```

Figure 3. The transformed, refactored function on the right uses a guard clause and preserves the normal processing.

This demonstrates an advantage of the transparency of srcML. It is not necessary to markup all new, generated code in srcML. Explicit source-code text can be used in the transformation program interspersed with XML programming statements. If full srcML markup is needed, this partial srcML document can be converted to source code, and then processed by the srcML translator to generate a fully-marked srcML document. After the creation of the guard clause the main template calculates the previous indentation of the if-keyword using the XPath: `preceding-sibling::text()[1]`.

The remainder of the main template handles the contents of the nested conditional and "un-nests" the contents. The entire contents of the block are converted to a single string. The braces at the start and end of the block are removed by extracting a substring of the block that removes the first and last characters.

This allows the transformation to work at multiple levels. We are able to identify where the block is by using the syntactic elements. Even though the contents of the block contain syntactic elements, we are able to ignore the syntactic elements and process the contents of the block as a string.

This string is passed to a template that removes these characters from the beginning of each line. The template used for this is not shown, but is a recursive template that splits the strings at the new line and checks for needed trimming of the white space.

The current form of our refactoring has some limitations. First, the identification of a nested conditional (versus normal processing) is limited to a special function name. The user would have to indicate, perhaps by location, the nested conditional. Second, the `return`-statement has a fixed return value. The value could be indicated by the user, e.g., via a parameter, or deduced from the context. Improvements can also be made to the XSLT program to make it simpler and more straightforward by defining special XPath functions for the string handling, e.g., indentation.

This example demonstrated a flexible, non-intrusive approach to source-code refactoring. By using a source-code representation appropriate to the problem, all levels of source-code information were available for transformation. Although XSLT was used in the example, any XML transformation language or API could have been used.

3. Future Directions

Our current work continues to expand the example shown to the other refactorings in Fowler's catalog. Even with the example shown, there exist open issues regarding the specification of the location of the refactoring and the matching of the documentary structure. We also plan to compare our method with other transformational/refactoring approaches.

The presented refactoring involved a single source-code file. Refactorings that require more than one source-code file will involve the formation of complex srcML documents, i.e., a single srcML document that contains the representation of multiple source-code files. The srcML format already provides for this by allowing the main element of a source-code file, i.e., the *unit* element, to be nested. The element *unit* also already contains attributes to store directory and file information. Therefore a merge and split can be used to perform a single refactoring on multiple files.

A disadvantage of a direct document-oriented source-code representation is the lack of higher-level abstractions. These higher-level models are associations between potentially separate parts of the source code. These source models include call graphs, logical class structures, and task-specific abstractions. The document representation can only mark what is in the code and is unable to directly mark higher level associations.

The representation allows for the integration of the source-model information with the source code. In effect, we actually allowed one to imbed any type of meta-information into the source. XPath expression involving the source model can be used in the location of a particular element in the source code.

This work was supported in part by a grant from the National Science Foundation (CCR-02-04175).

4. References

[1] Baxter, I. D., Pidgeon, C., and Mehlich, M., "DMS: Program Transformations for Practical Scalable Software Evolution", in Proceedings of 26th International Conference on Software Engineering (ICSE04), Edinburgh, Scotland, UK, May 23 -28 2004, pp. 625-634.

[2] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143.

[3] Collard, M. L., Maletic, J. I., and Marcus, A., "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9 2002, pp. 34-41.

[4] Cordy, J. R., "Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 196-206.

[5] Fowler, M., Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[6] Klint, P., "How Understanding and Restructuring Differ from Compiling - A Rewriting Perspective", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 2-12.

[7] Maletic, J. I., Collard, M. L., and Marcus, A., "Source Code Files as Structured Documents", in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, France, June 27-29 2002, pp. 289-292.

[8] Van De Vanter, M. L., "The Documentary Structure of Source Code", Information and Software Technology, vol. 44, no. 13, October 1 2002, pp. 767-782.