A Lightweight Transformational Approach to Support Large Scale Adaptive Changes

Michael L. Collard
Department of Computer Science
The University of Akron
Akron, OH 44325
collard@uakron.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, OH 44242
jmaletic@cs.kent.edu

Brian P. Robinson
ABB Corporate Research
Raleigh, NC 27606
Brian.p.robinson@us.abb.com

***Abstract*— An approach to automate adaptive maintenance changes on large-scale software systems is presented. This approach uses lightweight parsing and lightweight on-the-fly static analysis to support transformations that make corrections to source code in response to adaptive maintenance changes, such as platform changes. SrcML, an XML source code representation, is used and transformations can be performed using either XSLT or LINQ. A number of specific adaptive changes are presented, based on recent adaptive maintenance needs from products at ABB Inc. The transformations are described in detail and then demonstrated on a number of examples from the production systems. The results are compared with manual adaptive changes that were done by professional developers. The approach performed better than the manual changes, as it successfully transformed instances missed by the developers while not missing any instances itself. The work demonstrates that this lightweight approach is both efficient and accurate with an overall cost savings in development time and effort.**

Keywords- Source Code Transformation, static analysis

I. INTRODUCTION

Many industrial software products have very long life-cycles. These systems represent substantial investments and are valuable assets that support the business of the organization. Not only do these systems change as new features are added, but they must also undergo modification to address changes to dependent platforms and libraries. Here, we are particularly interested in these adaptive maintenance tasks. These tasks involve changing the software to respond to changes in its environment or platform (e.g., operating systems, compilers, hardware components, APIs, etc).

Adaptive maintenance changes are unique, as they are typically outside of an organization's control. Hardware component obsolescence, changes to underlying operating systems, and new compiler or platform versions are all examples of unexpected changes that have occurred recently at ABB Inc. As these changes are oftentimes unexpected, organizations are forced to react to these adaptive changes as they are discovered. This has a negative impact on the schedule and cost for projects. Moreover, the changes are primarily repetitive in nature and can be highly fault prone when done manually by a developer.

When an organization is faced with these changes, they must either adapt the system or, if possible, postpone the changes until a later time. While postponement is quite common, especially in operating systems and platforms, there are many practical reasons to make the investment. These include access to new features and technology that may be included in the new version as well as customer or market requirements. It may also prevent customers from having to perform invasive workarounds.

Ideally, changes to a critical part of a software's environment will be clearly documented in the change log or release notes. Unfortunately, there is rarely enough detail to clearly direct a developer to the needed changes. Due to this, many developers identify adaptive maintenance problems by experimentation. The impact environmental changes have on a system can be observed statically at build time, or dynamically at run time. Once the changes necessary to address the problems are identified, developers can be assigned to manually identify the errors and make corrections in the source code. Regression testing can then be done to identify any errors that occur due to the changes.

This paper presents a lightweight transformational approach to automate adaptive maintenance changes using the srcML toolkit [4, 11]. srcML was selected, as it preserves the programmer's view of the source document. As the resulting code from these transformations will be manually maintained by developers going forward, any changes to the overall structure or style may lead to a rejection of the new changes, as described in a number of studies on past real world projects [5, 13]. Examples have been given [5] 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.

The approach is also selected because it is robust in working with incomplete code, different compiler versions, and partially migrated source code. Well known XML technologies such as XPath, XSLT, and LINQ can be used, thus reducing the overhead of training and other adoption costs. The technique is evaluated by automating six types of adaptive maintenance changes on two different ABB Inc. products that were previously changed manually by developers. The study shows that this approach is very

efficient and is more accurate at identifying the changes than the manual change process.

While other approaches have been successfully used to perform these types of adaptive maintenance changes, most notably the work by Baxter [1], here we demonstrate that a combination of lightweight parsing and lightweight static analysis is sufficient to accomplish this task in the context of large scale systems. Additionally, we demonstrate that standard, well known XML technologies can be used to address this problem in a cost effective manner.

This paper is organized as follows. Section II presents the technique and the srcML tool. Section III presents the transformations necessary for the migration along with the XSLT implementations. A brief discussion of srcML extension functions that provide the lightweight static analysis needed is given in Section IV. Section V describes the results of a case study of our transformations applied to two large industrial software systems that underwent adaptive maintenance changes. We follow that with related work and conclusions.

```
#include "rotate.h"

// rotate three values
void rotate(int& n1, int& n2, int& n3)
{
    // copy original values
    int tn1 = n1, tn2 = n2, tn3 = n3;

    // move
    n1 = tn3;
    n2 = tn1;
    n3 = tn2;
}
```

Figure 1. Source code example.

II. SRCML OVERVIEW AND IMPROVEMENTS

Our approach for supporting adaptive maintenance tasks is based on the srcML format and toolkit. srcML (SouRce-Code Markup Language) [4, 11] is an XML format used to augment source code with syntactic information from the AST to add explicit structure to program source code. All original text of the source code, including comments, preprocessing information, and formatting, is preserved and identified for use by program-comprehension tools and development environments. The focus is to construct a document representation in XML instead of a traditional source-code document or data representation. This representation supports a programmer-centric, rather than compiler-centric, view of the source code.

The srcML format is supported with a toolkit, src2srcml and srcml2src, that supports conversion between source code and the format. Multiple languages, including C, C++, and Java, are supported. This format has been previously used for lightweight fact extraction [2], source code transformation [3], and pattern matching of complex code [8]. The format and translation tools for srcML have many advantages for adaptive maintenance tasks.

Complete view of the code: srcML takes an unprocessed view of the source code. This means that all preprocessor statements, template definitions, etc. are preserved and can be transformed. For example Figure 1 shows a small C++ function and Figure 2 shows the srcML for this code. Note that all original program elements and text are preserved.

Robustness: srcML is able to represent code that cannot be compiled. The src2srcml tool is able to convert incomplete code even when there are translation problems. These problems only have an effect on the quality of the markup and do not lead to a loss of any of the original text.

Efficiency: The tool is very efficient with a translation speed of 25 KLOC per second and can handle almost 3,000 files per minute, e.g., the entire Linux kernel can be converted to the srcML format in less than seven minutes. Going from srcML back to source code is handled by the tool srcml2src, which is even faster with speeds over 250 KLOC per second.

Independence: The srcML translator is not based on the parser of a particular compiler. It contains its own parser that only needs to understand enough about the code to insert the proper markup. Because it stops at this markup, it can allow syntax that a particular compiler cannot.

XML: The srcML format is an XML representation, and is designed to take advantage of current and future XML tools for transformation, validation, etc. To date the format has been successfully used with XSLT, DOM, and SAX.

The representation and toolkit are used to support lightweight fact extraction, querying, transformation, and validation of source code. With this format, our approach must address two requirements to fully address the problem of adaptive changes. First, we must specify the location of the transformation. For a given maintenance task, the area of the code that needs to be transformed must be identified, e.g., specific method or statement. With the srcML format, these locations in the code can be specified with either an XPath expression or a LINQ query, both of which use the srcML markup. For example, to match all if-statements that have the new operator in a condition (for C++) the XPath is:

```
.//src:if[src:condition/op:operator='new']
```

An example LINQ query is:

```
var ifStmts = from stmt in doc.Descendants
    (SrcML.SRCNS + "if")
    where stmt.Element(SrcML.SRCNS + "condition").
    Descendants (SrcML.OPNS + "operator")
    .Any(x => x.Value == "new") select stmt;
```

Second, given the particular statement or location to be transformed, we must state the transformation. The transformation describes how the code is modified, including any additions and/or deletions to the code. The generation of the new code often requires information from the original code, some of which may not be explicit, and static analysis of the code may be required to obtain this pertinent information. Based on the srcML format, transformations for adaptive maintenance can be written using any XML transformation tool, e.g., SAX, DOM, etc.

For the rest of this paper, we write our transformations in XSLT. This representation for the transformation was chosen mainly because it allows us to express the location of

```
<cpp:include>#<cpp:directive>include</cpp:directive> <cpp:file>"rotate.h"</cpp:file>
</cpp:include>

<comment type="line">// rotate three values</comment>
<function><type>void</type> <name>rotate</name>
<formal-params>(<param><type>int&amp;amp;</type> <name>n1</name></param>,
<param><type>int&amp;amp;</type> <name>n2</name></param>,
<param><type>int&amp;amp;</type> <name>n3</name></param>)</formal-params>
<block>{
    <comment type="line">// copy original values</comment>
    <decl-stmt><decl><type>int</type> <name>tn1</name> = <name>n1</name>,
<name>tn2</name> = <name>n2</name>, <name>tn3</name> = <name>n3</name></decl>;
</decl-stmt>
    <comment type="line">// move</comment>
    <expr-stmt><expr><name>n1</name> = <name>tn3</name></expr>;</expr-stmt>
    <expr-stmt><expr><name>n2</name> = <name>tn1</name></expr>;</expr-stmt>
    <expr-stmt><expr><name>n3</name> = <name>tn2</name></expr>;</expr-stmt>
}</block></function>
```

Figure 2. Source code from Figure 1 marked up in srcML with all original text preserved.

the needed transformation in XPath. It is also a widely used XML technology that is well understood.

Once an XSLT template is created that matches the location of the transformation, the body of that XSLT template can be used to construct the necessary modifications. This modified code can include literal text for added code, copies of parts of the original code, static analysis conducted directly in the transformation, or any combinations of these. More details on the transformations are provided in Section V.

Another advantage of using XSLT is that XPath extension functions can be created. For this application, these extension functions are used to encapsulate complicated and difficult-to-understand XPath patterns for locating the source of a transformation. More importantly, they can be used to encapsulate the on-the-fly static analysis that may be needed to create the desired transformation. This approach has been successfully used with very complicated code patterns applied to methods [8].

Once the XSLT transformations for adaptive maintenance have been constructed, a number of important practical issues arise concerning how to apply the transformations to a complete large-scale system. It is possible to use the src2srcml tool to convert a single source code file to srcML, apply the transformation to the srcML file using any XSLT tool, e.g., xsltproc, then use the srcml2src tool to convert the transformed srcML back to source code. However, for large projects it is more practical to apply the transformation to the entire project at once. This requires a change to the srcML toolkit which adds support for a srcML archive. This format allows for an entire project, consisting of multiple source-code files, to be converted and stored in a single srcML document. Transformations are applied to the entire project at once. The modified source code files and their directory structure can then be extracted from the modified srcML archive.

While representing an entire project in one srcML archive is convenient, the archive can become exceedingly large and create scalability issues for XSLT transformations. Similar to XML DOM approaches, XSLT requires the creation of the complete XML tree in memory before it can execute the transformation. This is unlike SAX approaches that only store what is needed at the moment of transformation.

To deal with the scalability issue, the srcml2src tool is also extended to support applying an XSLT transformation to a complete project in srcML in a more efficient manner. The tool takes a srcML archive and applies the XSLT transformation to each individual source code file, and then combines the output into a transformed srcML document. Since the XSLT transformation is only applied to the source code of a single file in the srcML archive at a time, the scalability issues are avoided. This allows the

```
<xsl:template match="src:decl_stmt[src:decl/src:init/src:expr/op:operator='new']">
  <!-- Copy the declaration, without any part of the initialization -->
  <xsl:copy-of select="src:exclude(., src:decl/src:name/following-sibling::node())"/>

  <!-- Wrap a try catch around the initialization of the variable, now in separate statements -->
  <xsl:variable name="trycatch">
try {
  <xsl:copy-of select="src:decl/src:name | src:decl/src:name/following-sibling::node()"/>;<xsl:text>
</xsl:text>} catch (...) {
  <xsl:value-of select="src:decl/src:name[1]"/> = NULL;
}</xsl:variable>
  <!-- Copy the generated try catch with the indentation of the original statement -->
  <xsl:copy-of select="src:indent(src:indentation(.), $trycatch)"/>
</xsl:template>
```

Figure 3. XSLT template used for transformation of declaration statements due to change in behavior of operator new

```
<!-- Match arguments to a template which are themselves templated,
     but no template parameter is given -->
<xsl:template match="src:argument_list/src:argument/src:name[
                    ancestor::src:template and ancestor::src:class/src:name=.]">
<!-- Insert a template parameter -->
<xsl:value-of select="."/>&lt;<xsl:value-of
select="src:list(ancestor::src:template/src:parameter_list/src:param/src:name,
                 ', ')"/>&gt;</xsl:template>
```

Figure 4. Main portion of the transformation to fix a use of a C++ templated class that requires specialization

transformation of very large systems stored in a single srcML document in a reasonable amount of time. For example, an XSLT transformation can be applied to the entire Linux kernel in under five minutes.

III. EXAMPLE TRANSFORMATIONS

This study focuses on creating transformations in srcML for two large adaptive maintenance changes which occurred recently at ABB. These two changes involve two separate products, one of which is an embedded device and the other is a Windows desktop application. These changes arise due to changes in the VxWorks and Microsoft Visual Studio development platforms. Experienced developers at ABB originally performed the necessary changes manually on the source code. These changes dealt with changes to system APIs, runtime changes in the underlying platform, and changes to the compilers used to build the products. Each of the specific adaptive changes are described, along with how the particular transformation source is located and how the transformation constructs the transformed code.

A. *Change to Operator `new`*

The first adaptive change is in response to changes in the compiler due to a change in the C++ standard. This change affects the semantics of the operator `new`. It was common practice to directly call the operator new in the initialization of a variable declaration, as we see below:

```
CNICmdFactory *cmdFactory = new CNICmdFactory;
```

Earlier versions of the language standard had `new` return NULL in the case of a memory error. Error checking was done by simply checking the result for NULL and calling an error handler if necessary. The behavior of the `new` operator was changed to throwing an exception when memory allocation issues occurr. A local fix for the problem is to wrap the call in a `try/catch` block, preserving the original behavior and preventing any unhandled exception errors from crashing the system. An example is shown below:

```
CNICmdFactory *cmdFactory;
try {
   cmdFactory = new CNICdmFactory;
} catch (...) {
   cmdFactory = NULL;
}
```

For this transformation, specific statements that include a call to the operator `new` must be identified, and the expression that calls `new` must be isolated in a `try/catch` block. Finally, the result must be tied into the original statement. A portion of an XSLT program that can detect this and make this transformation for a declaration is shown in Figure 3. First, the XSLT template is made to match any declaration statements (`src:decl_stmt`) that include a call to the operator `new` in the initialization. The srcML translator optionally supports the markup of operators with the option "--operator". These operators have their own namespace, and the existence of the new operator can be checked using the predicate `op:operator='new'`. In order to determine all of the uses of `new` in the source code, an XPath query is created using that predicate. The results of this query identified twelve different contexts for the use of `new` in the source code to be transformed. These included uses of `new` statements inside expression statements, while loops, if statements, and return statements. Each of these distinct contexts may require a different transformation. For example, a function that uses the `new` operator in a return statement will need to be transformed differently than a `new` operator used in an assignment statement.

Once all of the uses of the `new` operators in the source are found, replacement code can be generated. For cases where new is used in a variable declaration, the original declaration is copied, but without the initialization. In order to simplify the transformation, an extension function, `src:exclude`, is used to copy the declaration statement and exclude the initialization. The next part of the transformation creates the appropriate `try/catch` block. This uses a combination of explicit text, i.e., `try`, and code copied from the original, i.e., variable name and assignment to initialization. The `try/catch` block is created in a variable and then copied to the output of the transformation reflecting the indentation used by the original declaration statement. Determining the indentation for the original statement is encapsulated in the srcML extension function `src:indentation()`, and applied to the generated `try/catch` with the srcML extension function `src:indent()`.

Expression statements are handled in a similar manner with the entire expression moved into the `try/catch` block. However, for other statements the transformation is more involved. For example, it is common practice in this codebase to assign the value of a variable using a new expression in the condition of an `if` statement and trap the error using the if statement:

```
if ((nioMemCpy = new DataByte) == 0 ) {}
```

In this case, the required transformation must move the assignment of the variable outside the `if` statement and then compare the variable directly to NULL inside the `if` statement. In the case of a `new` in the condition of a `while` statement, the assignment with the expression must also be inserted at the end of the block of the `while` statement, or a block statement created if the `while` statement did not originally have one.

Due to the large number of different usages of `new` in the source code, the complete transformation used twelve small XSLT templates. By using srcML as a fact extractor initially, all uses of the `new` operator can be identified quickly and transforms can be created. In addition, if a use requires a complicated transformation, but occurs in a small number of places, the developer can decide to perform those changes manually and just create transformations for the more frequently occurring cases.

B. *Template Class Requires Specialization*

Another recent adaptive maintenance change involves a change in the compiler where a default must now be explicitly stated for C++ templates where arguments in a template definition must be specialized. For example, in the following code the template class `CItemTraitsHelper` is used as a template argument.

```
template <class T>
class ATL_NO_VTABLE CItemTraitsHelper :
   public
CPropertyObjectHelper<CItemTraitsHelper, T>
```

Note that when used as a template argument, the class is not specialized. In this example we need to replace the use of the class `CItemTraitsHelper` with `CItemTraitsHelper<T>`, where `T` is the template parameter. After the transformation the resulting code is:

```
template <class T>
class ATL_NO_VTABLE CItemTraitsHelper :
   public
CPropertyObjectHelper<CItemTraitsHelper<T>, T>
```

The transformation consists of a single XSLT template and is shown in Figure 4. The XPath expression for the location starts with matching all arguments:

```
src:argument_list/src:argument/src:name
```

To refine this to template arguments the predicate checks that the context is in a template definition, i.e., includes the check that this is a template argument: `ancestor::src::template`. At this point we have the names of all template arguments. Now we need to determine if the name of the argument is the same as the templated class: `.=ancestor::src:class/src:name`.

Next, the template argument that requires specialization is corrected. First, we copy the name of the argument: `<xsl:value-of select="."/>`. Then a template argument list is inserted. The template argument angle brackets are literally escaped and inserted into the text. It is not a requirement that a srcML document be fully marked, so we only have to put in the text and not try to match the complete, correct srcML elements.

The other piece of information that needs to be derived from the original code is the name of the template parameter from the class template. From our location in the code, a list of the template parameters can be found with the XPath:

```
ancestor::src:template/src:parameter_list/
                                src:param/src:name
```

There may be more than one template parameter, and in a template parameter list a comma must separate these. To make the transformation more clear, we have used a srcML extension function `src:list` that takes every element of the first list, i.e., a template parameter, and separates it with a comma and a space.

With this transformation we have identified where templated classes are used unspecialized in their own declaration, prevented false positives, i.e., other code which uses parameters and templates, and handled multiple template parameters.

```
<!-- Match declarations in initialization of for statements
     where the code following the for statement uses the name -->
<xsl:template match="src:for[
                  src:init/src:decl/src:name=following-sibling::src:*//src:name]">

  <!-- explicitly output the initialization as a declaration before statement on its own line -->
  <xsl:copy-of select="src:init"/><xsl:text>
</xsl:text>
  <!-- output rest of for, but without initialization -->
  <xsl:copy-of select="src:exclude(., src:init/src:decl)"/>
</xsl:template>
```

Figure 5. Transformation to move declarations of iterator variables when they are used outside of a for-statement

C. *Iterator Variable Scope*

In this adaptive change, the scope of a variable declaration in a `for` statement was previously that of the enclosing block, e.g., `for (int i = 0;;) { … }`. Many programmers assumed, to be the same as

```
int i = 0;
for (;;) { … }
```

This creates a problem when the variable `i` is used in the code following the `for` statement. This transformation involves locating `for` statements where this occurs, and moving the variable declaration outside of the loop.

The main portion of the transformation is shown in Figure 5. First, we must match the occurrences of iterator-variable declarations that must be moved:

```
src:for[src:init/src:decl/
 src:name=following-sibling::src:*//src:name]
```

This example matches all `for` statements where the name of a declaration in the initialization, `src:init/src :decl/src: name`, is used anywhere in the statements that follow the for-statement:

```
<!-- Match calls to deprecated str*cpy functions -->
<xsl:template match="src:call[src:name='strcpy']">
  <xsl:text>strcpy_s(</xsl:text>
  <xsl:copy-of select="src:argument_list/src:argument[1]"/>
  <xsl:text>, </xsl:text>
  <xsl:copy-of select="src:default(src:defnsize(src:argument_list/src:argument[1]),
                                                              '/* FIXSIZE */')"/>
  <xsl:text>, </xsl:text>
  <xsl:copy-of select="src:argument_list/src:argument[2]"/>
  <xsl:text>)</xsl:text>
</xsl:template
```

Figure 6. Transformation to replace deprecated strcpy functions with the safer alternatives strcpy_s. The new call requires an extra parameter which is the size of the destination buffer. Whenever possible lightweight static analysis is applied on the context of the call by the srcML extension function *src:defnsize*. If not found, a comment is inserted to indicate to the developer that the size needs to be manually inserted.

```
following-sibling::src:*//src:name
```

In this case the following-siblings axis includes all statements that follow the completion of the `for` statement, i.e., they do not include the statements in the for-statement block. These are the statements that are in the old scope of an iterator variable declaration. We examine these statements and determine if the iterator variable is being used. If so, then this scope problem on the iterator variable is fixed.

Matching and then moving particular statements is not straightforward in XSLT. In XSLT we match parts of the input and indicate the processing of these parts. Rearranging statements may involve multiple templates. Generally, this would be done with an empty template to match the old position, and another template that inserted the moved statement along with the existing code. The first transformation for this problem used this approach.

To avoid using multiple templates, we created an XSLT extension function that allows us to perform this in one template. To perform the move first we copy the current iterator variable declaration so that it now occurs outside of the for-statement: `<xsl:copy-of select="src:init"/>`. We put this copied declaration on its own line using the `<xsl:text>...</xsl:text>`. Now we need to copy almost all of the for-statement, excluding the iterator variable initialization. To make this easier, we created a srcML extension function `src:exclude(to-copy, to-exclude)`. This function recursively copies the first parameter, but during the recursive copy any elements in the second parameter are excluded. The effect is to copy the tree *to-copy* but exclude the subtree *to-exclude*. The second parameter is treated as a node-set, so it can include more than one element, i.e., more than one subtree can be excluded.

In this particular case, we want to copy the `for` statement but exclude the declaration that is in the initialization:

```
<xsl:copy-of
select="src:exclude(.,src:init/src:decl)"/>
```

Note that we copy the entire initialization, `src:init`, but exclude only the declaration, `src:init/src:decl`. The reason for this is that we have to change the declaration into a declaration statement, i.e., the semicolon has to be duplicated.

D. Deprecated String Functions

In order to prevent buffer overruns, string functions that do not take the size of the destination buffer into account need to be replaced with those that do. In general, this is a transformation dealing with security issues. In our case, the new version of the compiler issues a deprecation-type warning whenever the function `strcpy` is used. The solution is to replace this call with the safer function `strcpy_s` that includes a destination buffer size. The function `strcpy_s` is Microsoft specific and is similar to the standard function `strncpy`. The main portion of the transformation to make this change is given in Figure 6. The template matches any calls to the function `strcpy:src:all[src:name='strcpy']`. Most of this part of the transformation is constructing the text of a new call. The name of the new call is inserted as text: `<xsl:text>strcpy_s(</xsl:text>)`. The arguments are mapped from the old call to the new call. For the first parameter this is:

```
<xsl:copy-of
select="src:arg_list/src:argument[1]"/>
```

What cannot be directly copied from the original code is the derivation of a new second parameter to the `strcpy_s` function, which is the size of the first argument. This size depends on how the buffer for this argument was created, i.e., in its declaration, use of `malloc`, use of `new`, etc. Also, the variable used for the first parameter may have been set through various aliases. Accurately determining this would require heavyweight analysis of the program and in some cases may not be possible. For this lightweight approach we analyze the current file and, when we are unable to automatically determine the size, a comment is added identifying where manual changes are needed.

To hide the complexity of the analysis to determine the proper size of the buffer, we have used a srcML extension function `src:defnsize` that, given the name of a variable, tries to determine the size. In this transformation it is used as follows:

```
src:defnsize(src:argument_list/src:argument[1])
```

If the size of the buffer cannot be determined by the function `src:defnsize`, the default text `/* FIXSIZE */` is inserted. Another srcML extension function, `src:default` is used to avoid if-else code. So the second parameter is a result of

```
src:default(src:defnsize(src:argument_list/
        src:argument[1]), '/* FIXSIZE */')
```

With this transformation we have successfully identified calls to a specific function, e.g., `strcpy`, and constructed a new call based on the arguments to the original call. We

have also determined the size of a buffer based on the original code and demonstrated the use of srcML extension functions to hide complexity, and provide reusable functionality. Note that the developer will have to manually determine the proper buffer sizes in some cases. However, these are clearly marked with an identification mark that can be easily found. In our case studies, we were able to identify the size correctly 38% of the time. Most of the cases where we are unable to automatically identify the size involve global pointers to complex data types. The manual changes performed by the developers required adding new functions to report the size of these complex data types.

the expression with the name of the macro, `AfwSafeVectorBegin` and inserting the current variable name with the expression: `.//src:name/src:name`. The nested use of the `src:name` element in the path is to accommodate a variable name that is indexed. A related problem is the application of the address operator to the end of the vector, as in the expression `&(*v.end())`, with a corresponding macro `AfwSafeVectorEnd(v)`. The template to make this transformation is the second template in Figure 7.

These transformations identified expressions where invalid references to vector data are used. These expressions are modified to call a macro to handle the situation safely.

```
<!-- Match arguments of the form &v[0] where v is any variable name  -->
<xsl:template match="
     src:argument/src:expr[
         op:operator='&amp;' and
         src:name/src:index[src:expr='0'] and
         count(*)=2
     ]">AfwSafeVectorBegin(<xsl:value-of select=".//src:name/src:name"/>)</xsl:template>

<!-- Match arguments of the form &(*v.end()) where v is any variable name  -->
<xsl:template match="src:argument/src:expr[
     op:operator[1]='&amp;' and op:operator[2]='(' and op:operator[3]='*'
     ]">AfwSafeVectorEnd(<xsl:value-of select=".//src:name"/>)</xsl:template>
```

Figure 7. Main template of the transformation to keep STL vector data private.

E. STL Vector Data is Private

Changes to a run-time framework can cause run-time errors to occur, where they did not occur previously. One case of this occurred with the STL vector concerning access to the start of vector data when done with the expression `&v[0]` (where `v` is an STL vector). This expression was used as an argument to COM methods and caused a number of run-time errors in the new platform version. In order to get around this problem, the macro `AfwSafeVectorBegin(v)` was defined to safely dereference `v` and to handle the case where the vector is empty. The fix is to detect uses of `&v[0]` and to replace these with a call to the new macro.

The basic transformation consists of a single XSLT template as shown in Figure 7. The srcML for the location, with the operator & escaped, is:

```
<op:operator>&amp;</op:operator><name>
  <name>v</name><index>[<expr>0</expr>]
   </index></name>
```

The XPath expression for the location starts with matching all expressions used as arguments: `src:argument/src:expr`. At this point we need to narrow this down using pattern matching to match the exact uses of `&v[0]`. First, we check that the expression contains the address operator. The existence of the address operator can be checked using the predicate `op:operator='&'`. Then we need to determine if the address is being taken of a variable indexed to 0 with the predicate: `src:name/src:index[src:expr='0']`. Finally, the predicate checks that the expression consists only of the address operator and the index of the variable to 0.

Once these locations are discovered, the expressions need to be rewritten to use the macro. This is done by replacing

One limitation of the lightweight approach is that the index is compared directly to the literal value 0. If a macro, or constant, or expression was used that evaluated to 0, then this would not have been identified. The transformation can be extended to special cases, but not easily to the general case with a lightweight approach. However, we didn't find any of these examples in this study.

F. Fully-Qualifying Function Pointers

Another situation that arises specifically when compiler migration is performed involves cases when a variable is not fully prefixed. In our case study this occurred with function pointers passed as arguments. For example, the function pointer `OnLookupAgain` used as an argument to a call should be fully qualified as `&AfwNSI::CQM::OnLookupAgain`.

This case presents a challenge to a lightweight approach. Based on the usage of the variable and without a definition, it is not possible to determine if a variable is a function pointer. Since it is common for the declaration of a function pointer to be defined in a source file external to the one being transformed, or even at run-time based on startup options, an examination of all files in the system may have to be done.

Since the purpose of these transformations was to assist the developer by semi-automating their task, a full solution was not used. In this case, the transformation is written specifically for this function. The transformation is a single template that matches expressions where the name of the function pointer is used (as provided by the developer), and qualifies them with the provided qualification. This transformation can be generalized to convert any function pointer with a fully qualified name if the developer provides both. In this case, the transformation serves as a search and replace, but with knowledge of context.

We examined methods of automatically generating this list of needed replacements. One approach is to use the compiler to find the lines where this error occurred. While this gave us the names of the function pointers, it did not solve the problem of the proper qualification. A solution to this is to first scan the entire system for a list of function pointers, and then run the transformation on that list.

IV. Evaluation

In order to evaluate the transformations described in Section III, a set of case studies are conducted, consisting of adaptive maintenance changes selected from two different industrial systems developed at ABB. The first set of adaptive changes deal with changes to the C++ standard deployed in a new version of the VxWorks development platform. The second set of adaptive changes deal with changes to the Microsoft Visual Studio compilers and underlying runtime framework. The goal of these studies is to compare the manually changed code and the automatically transformed code. Cases where the technique transformed code in an incorrect place represent false positives, while missing transformations represent false negatives. The computer used for the automated transformation was a Lenovo ThinkPad W500, containing a 2.5 GHz Intel Core 2 Duo CPU with 4 Gigabytes of RAM, running the Windows XP operating system.

A. *Case Study 1*

The product in the first study contains approximately 122 KLOC of C and C++ code contained in 405 files. The code is composed of a hybrid mix of procedural (54%) and object-oriented code (46%). This system had previously undergone manual adaptive maintenance, due to changes in the way the gcc compiler handles the new operator in C++. Specifically, the compiler adopted the new standard of throwing an exception when memory is not available, as opposed to just returning NULL. Due to this change, all instances of the new operator had to be changed. Two baselines were used in this study. The first is taken from just before the manual adaptive maintenance changes were made, and is the starting point for the automated transformation. The second baseline was taken just after the manual changes were completed and is used to verify that the automated transformation was successful. In total, 479 manual adaptive maintenance changes were made to accomplish this compiler update.

First, the source code for the entire system was converted into srcML by the toolkit. It took only a few seconds to convert the 122 KLOC into the srcML format. The transformation was then run on the code, also taking only a few seconds. Finally, the transformed srcML was converted back to source code in only one second. Once the transformations were run on the system, they were validated against the manual changes to identify any false positives or false negatives. Finally, the source repository was studied to identify any cases where the original manual transformation was incorrect, resulting in a later change. In these cases, the automated transformation is also compared to these later changes to determine if the tool performed more accurately than the original manual transformation.

All 479 manual changes were made correctly by our automated transformation approach when compared to the manual changes. That is, we did not miss any of the changes performed manually and did all of them correctly. In addition, our automated approach identified 40 cases that were missed by the developers during the initial manual change (for a total of 519 changes). Upon examining later versions of these files and their version history in the source repository, it was determined that all of these 40 missed changes were later identified and corrected manually. These represent changes that were originally missed and detected at later points in time by other forms of verification and validation. However, our automated approach was able to identify and correct all without additional effort or cost. There is nothing unique about the 40 changes missed by developers. They are all just instances of other changes that the developers simply missed.

TABLE I. Results for the second case study

	Correct	Incorrect	Manually Missed
Template Class	3	0	0
Iterator Variable	231	0	86
Deprecated Str	406	0	5
Data Private	1419	2	213
Qualifying Functions	1	0	0
Totals	**2060**	**2**	**304**

B. *Case Study 2*

The system in the second study contains approximately 3.9 MLOC of C and C++ code spread among 13,800 source files. This system is composed of mostly object-oriented C++ code (84%) with the remaining 16% being procedurally designed C code. The system has recently undergone adaptive maintenance changes due to a C++ compiler migration (i.e., Visual Studio 2003 to Visual Studio 2005). Two different baselines of the source code are used for this study, one just before, and one immediately after, the adaptive maintenance changes were performed. All of the changes were logged in a version control system as a single transactional check-in. In total, the developers manually performed 1756 adaptive maintenance changes.

First, the source code for the entire system was converted into srcML by the toolkit. It took 211 seconds to convert the entire 3.9 MLOC into the srcML format. All five transformations listed in Table I were then run on the system, taking a total of 653 seconds. Finally, the transformed srcML was converted back to source code, which took 51 seconds. Once the transformations were run on the system, a separate author validated the transformed changes against the manual changes and any false positives or false negatives were identified. Finally, the source repository was studied to identify any cases where the original manual transformation was incorrect, resulting in a later change. In these cases, the automated transformation is also compared to these later

```
<func:function name="src:defnsize">
  <xsl:param name="name"/>
  <xsl:variable name="attempts" select=
        src:use2defn($name)//src:index/src:expr
        src:use2defn($name)//src:init//src:call
              [src:name='malloc']/src:argument_list/src:argument/src:expr |
        preceding::src:expr[src:name[1]=$name and
op:operator[1]='=']/src:call[src:name='malloc']/src:argument_list/src:argument/src:expr |
        preceding::src:expr[src:name=$name and op:operator='new']//src:index/src:expr
)[last()]"/>
  <func:result select="$attempts"/>
</func:function>
```

Figure 8. The srcML extension function *src:defnsize* is an example of how static analysis of the code can be encapsulated. Given a name, the function finds the size of the buffer by looking at the declaration. The buffer may be sized statically, or dynamically by a malloc or new. Each form of buffer sizing is tried, with the last one found used. If no size can be found, the result is empty.

changes to determine if the tool performed more accurately than the original manual transformation.

Table I shows the results of the transformations for the second case study. All 1756 manual changes were made correctly by our automated transformation approach. Our approach did incorrectly transform two cases of the private data problem. One case was due to imprecise matching of the code pattern and could be corrected quite easily in the transformation. The other case required more precise type resolution to address properly. Doing this for all cases may not be possible in our approach and require either full type analysis of the system or developer input. Our automated approach also identified 304 adaptive changes that were missed by the developers during the initial manual change. Upon examining later versions of these files, it was determined that all of the 304 missed changes were later identified and corrected in the source repository. Again, for the most part these situations were not particularly unique, as the missed changes were instances of changes that are correctly made elsewhere in the source code.

V. Extension Functions

One of the advantages of the srcML format is that it provides access to all parts of the source code. No detail of what the developer writes, including comments and white space, is lost. Queries and transformations can be performed on all of these aspects. The format exists at a level right on top of the source code text. In order to provide an abstraction for concepts at a higher level then directly expressed in the source code, and directly represented in srcML, we have created srcML extension functions. Many of these were used in the preceding transformation examples. These extension functions can be used along with the full XPath language in expressions for location of the source of the transformation, in the source code analysis needed to perform the transformation, and in the generation of the transformed code.

An example srcML extension function, `src:defnsize`,, is shown in Figure 8. These functions are helpful for fact extraction and analysis, which would otherwise require more complex XSLT and XPath statements. Another srcML extension function, `src:use2defn($name)`, is used to find the definition where `$name` is the name of the variable. From the variable definition for a statically sized array we can find the expression of the size with the XPath expression: `src:index/src:expr`. We also need to look for dynamic allocation using `malloc` and `new`. This may occur in the declaration, or as an assignment in a previous statement. This is handled by other parts of the XPath expression. Many of the srcML extension functions used in this paper perform very general needs, such as determining variable scope or type determination. Building a useful set of extension functions will significantly help developer adoption of this technique.

VI. Related Work

We observed that automated source code transformations intended to be handed back to a developer must preserve the programmer's view of the document, i.e., preserve white space, comments, and the expressions of literals, and failure to do so may mean the rejection of the result [5, 13] and tool. In [13] the concept of the *documentary structure* of source code, whose elements include all white space and comments, is presented. 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.

In contrast to these requirements, software-development tools typically take a totally compiler-centric approach of representing the source code as an abstract syntax tree. It has been observed that these approaches are often not a good match to the problems that they are trying to solve [9, 13]. There are exceptions to this problem with compiler-centric approaches however, with one example being the DMS systems by Baxter [1]. Baxter has gone to great lengths to address this specific issue by storing important textual items within the underlying abstract-syntax graph. Also, as a full compiler (i.e., heavy weight) approach, it allows for static analysis to be built into the transformation. Our approach is very lightweight by comparison and uses widely available and accessible XML technologies. One approach is to move down to the level of lexical analysis and provide for the transformation at that level, as in [7]. This allows for the preservation of all of the text, but at a cost of complex

regular expressions. Also, with this approach, it is not as easy to provide for abstractions that reflect static analysis. Another approach that preserves the programmer's view is to move the transformation to the level of the grammar as in TXL [6]. Using this approach, the transformations are written as part of the grammar for parsing the language. The approach shares many of the advantages of our approach: preservation of programmer's view, scalability, robustness, etc. The difference is in the format of the transformation. Instead of grammar rules, our approach treats the text of the source code as data in XML, and the transformations are XML transformations.

The Proteus system [14] addresses similar problems of performing transformations on large C++ systems while preserving the layout and handling code before preprocessing. They refer to this as "high-fidelity" transformations. An AST approach is used, with white space and comments stored in additional AST nodes. They provide their own language YATL for transformations on the AST. Additionally, in [10] these documentary structure issues are seen as a cross-cutting concern in the form of annotated parse trees. Other approaches include using an intermediate language to describe the source, as in the case of the C Intermediate Language (CIL) [12].

The lightweight approach we have presented preserves the documentary structure, as is done in some of these approaches, while at the same time integrates static analysis into transformations that go down to the lexical level. No other work integrates a lightweight approach and static analysis in an efficient and useable manner.

VII. Conclusions And Future Work

The cases studies presented demonstrate that lightweight parsing combined with lightweight static analysis is adequate to support a transformational approach to automatically addressing certain adaptive maintenance tasks on two large real world systems of different domains and different development platforms. The approach presented work as well as developers in that it correctly modified all cases that were manually done (except for two cases – less than one percent for that particular type of adaptive change). Moreover, it worked better than developers in that it identified a significant number (13%) of cases that the developers missed in the original changed version. The developers later addressed these missed cases over a period of time from later testing and failures. Our approach made changes to all these cases correctly.

One clear limitation of our approach and the presented transformation is that the determination of the proper buffer size for the deprecated string change. In this case more complete static analysis would help but pointer alias analysis is a very difficult problem. We intend to see if we can improve the extension function `src:defnsize` to increase the accuracy, but we may need to do complete static analysis to address this problem fully.

We believe that the results are applicable to not only compiler migration, but to other adaptive maintenance tasks. The use of XPath extension functions to hide the details of complex XPath expressions made the transformations easier to understand. Details of the exact generated code to fix an adaptive maintenance issue may be project or even programmer specific. These transformations are easily modified to reflect this. They also provide a reusable set of functions for further transformations. We are also creating a larger library of extension functions to support general maintenance transformations.

References

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

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

[3] Collard, M. L. and Maletic, J. I. Document-Oriented Source Code Transformation using XML in Proceedings of 1st Intl. Workshop on Software Evolution Transformation (SET'04) (Delft, The Netherlands, Nov. 9, 2004), 11-14.

[4] 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), 34-41.

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

[6] Cordy, J. R., Dean, T. R., Malton, A. J., and Schneider, K. A. Source transformation in software engineering using the TXL transformation system. *Information and Software Tech*, 44, 13 (2002), 827-837.

[7] Cox, A. and Clarke, C. Relocating XML Elements from Preprocessed to Unprocessed Code in Proceedings of Proceedings of the IEEE 10th Intl. Workshop on Program Comprehension (IWPC'02) (Paris, France, June, 2002), 229-238.

[8] Dragan, N., Collard, M. L., and Maletic, J. I. Reverse Engineering Method Stereotypes in Proceedings of 22nd IEEE Intl. Conference on Software Maintenance (ICSM'06) (Philadelphia, PA USA, Sept 25-27, 2006), 24-34.

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

[10] Kort, J. and Lammel, R. Parse-tree annotations meet re-engineering concerns in Proceedings of the Intl. Workshop on Source Code Analysis and Manipulation (2003), 161–171.

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

[12] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science* (2002), 213-228.

[13] Van De Vanter, M. L. The Documentary Structure of Source Code. *Information and Software Tech*, 44, 13 (October 1 2002), 767-782.

[14] Waddington, D. and Yao, B. High-fidelity C/C++ code transformation. Science of Computer Prog, 68, 2 (2007), 64-78.