Received: 3 March 2016 | Revised: 8 July 2016 | Accepted: 6 September 2016

DOI 10.1002/smr.1831

WILEY Journal of Software: Evolution and Process

RESEARCH ARTICLE

Simplifying the construction of source code transformations via automatic syntactic restructurings

Christian D. Newman[1] | Brian Bartman[1] | Michael L. Collard[2] | Jonathan I. Maletic[1]

[1] Department of Computer Science, Kent State University, Kent, Ohio, USA

[2] Department of Computer Science, The University of Akron, Akron, Ohio, USA

Correspondence
Jonathan I. Maletic, Department of Computer Science, Kent State University, Kent, Ohio, USA.
Email: jmaletic@kent.edu

Abstract

A set of restructurings to systematically normalize selective syntax in C++ is presented. The objective is to convert variations in syntax of specific portions of code into a single form to simplify the construction of large, complex program transformation rules. Current approaches to constructing transformations require developers to account for a large number of syntactic cases, many of which are syntactically different but semantically equivalent. The work identifies classes of such syntactic variations and presents normalizing restructurings to simplify each variation to a single, consistent syntactic form. The normalizing restructurings for C++ are presented and applied to 2 open source systems for evaluation. The evaluation uses the system's test cases to validate that the normalizing restructurings do not affect the systems' tested behavior. In addition, a set of example transformations that benefit from the prior application of normalizing restructurings are presented along with a small survey to assess the effect of the readability of the resultant code.

KEYWORDS

adaptive maintenance, restructuring, syntactic isomorphism, transformation

1 | INTRODUCTION

Several critical software maintenance tasks can be addressed using a program transformation approach. Adaptive maintenance tasks are examples. Given an adaptive task, such as an API migration, developers identify all particular necessary changes and then develop a solution to address each. Adaptive maintenance is both routinely required and costly. For example, many companies have recently (in the past couple of years) migrated from Microsoft .Net 1.1/2.0 to .Net 3.5/4.0. The authors are familiar with the details from a major US insurance company that undertook this particular adaptive maintenance project. The migration involved more than 60,000 projects (approximately 11 million source code files) enterprise wide. The 2-year project incurred an effort of approximately 75,000 person/hours and a cost of more than $6.5 million. Although this is a substantial sum, this effort was only part of the organization's ongoing adaptive maintenance projects (more than $25 M during a 2-year period).

Although transformational approaches to solve adaptive maintenance tasks have proven to drastically reduce costs[1] and improve quality, these types of approaches are rarely used in practice on a wide scale.[2] Most organizations have developers manually fix each instance of an issue one at a time. There are several reasons for this gap between industry practice and research findings. Program transformation tools can be difficult to use, developers are unaware of the techniques or not properly trained, and even with proper training and usable tools, it is difficult to construct the actual transformation accurately for application to a large system (enterprise wide).

In using a transformational approach to solve a maintenance problem, a developer must identify each particular change and then develop a set of transformation rules to solve each problem. Ideally, a complete set of example changes are on hand to develop the entire set of transformations necessary to undertake the given maintenance fix. Typically, developers manually identify the necessary types of changes from examining systems that will be undergoing the maintenance task. In addition, the documentation for the new releases of APIs, compilers, and operating systems typically has lists of the major changes and some basic suggestions for what needs to be corrected to support new features and interfaces.

However, even given a complete set of necessary items to change to support an adaptive maintenance task, it will not cover all situations within the context of all software systems. That is, it is very difficult to

predict all the usages of an API, feature, or construct for any given system. As such, when constructing a transformation rule(s), we must also identify all syntactic situations in which it will be applied and make sure each situation is correctly accounted for.

As an example, take an API change that modifies a function to throw an exception instead of returning an error value. One solution is to have each call wrapped in a try-catch structure. The developer writes a transformation rule for expression statements that include the call and assumes it is complete. However, the developer can easily find that calls occur in expressions, not just expression statements. These calls may additionally occur in multiple syntactic variations, including variable declarations; the conditions of if, while, and for statements; return statements; and so on. These syntactic variations are not always obvious and are often discovered by trial and error. Each syntactic variation requires an extension or entirely new transformation rule.

The work presented here aims to reduce the complexity of the syntactic variations in the context of constructing transformation rules. To accomplish this, we developed a set of *restructurings*, which are program transformations that preserve semantics of the code. These *restructurings* are similar to refactorings; they are both transformations that preserve semantics. The difference is that a restructuring is not limited to making code cleaner or more readable. The restructurings we present convert a set of differing syntactic variations in a programming language into 1 standard form. The standard form is semantically equivalent to all original syntaxes, but now all syntactic variations are in a single, consistent syntax. This eases the construction of transformation rules. We leverage the fact that in any given programming language, there are often many different ways to express the same behavior. Take the following examples of declaring a variable and initializing its value. Both are semantically equivalent but syntactically different.

```
int sum = 0;
```

and.

```
int sum;
sum = 0;
```

If our maintenance task involves adapting a variable initialization, we need to either build a set of transformation rules to address each syntax individually or construct a single, more complicated, transformation rule to address both simultaneously. The approach taken here is to first automatically convert all these variations into one standard form and then build one transformation rule that works only on that single form. This greatly simplifies the construction of transformation rules to support various maintenance tasks and has the added value of simplifying the verification of the transformation process.

Our objective is to enumerate a set of standard forms for a programming language (C++) in the context of simplifying the transformation process. These standard forms must have the same behavior as the original code, which can be validated via unit tests as is typical with any change to source code. Moreover, we propose a set of restructurings that will convert a given segment of source code into its standard form.

We limit the discussion to C++. However, in practice, the same approach can be applied to any C-like imperative programming language. The contributions of this work are as follows:

- The definition of a set of *standard forms* within C++, where each standard form reflects a single syntax for set of syntactic variations with equivalent behavior.
- A validation of the standard forms.

Given these restructurings, program transformation tools can *selectively* process segments of code before some user-defined transformation rule is applied. The selective segments will first be normalized to a standard form and then the user's transformation rule can be applied. Thus, the user only needs to define one rule for one syntactic situation as opposed to many.

We also investigate how the normalization process effects the readability of the code. Any automated transformation process is in all likelihood going to have some negative effect on the readability of the code compared with manually correcting each situation. We are interested in the tradeoff between simplifying the development of transformation rules versus a possible reduction in readability of the final code.

This article is organized as follows. We first present related work to help motivate and differentiate this work. In Section 3, a more detailed motivational example is presented. Following in Section 4, we define the concept of syntactic standard forms and describe the process used to uncover them. The set of restructurings are named and presented in Section 5. An evaluation of the approach is given in Section 6 with threats to validity in Section 7. This is followed with a discussion of possible applications of the restructurings and lastly conclusions.

2 | RELATED WORK

Normalizing restructurings are similar to some of Fowler's refactorings.[3] When applied to a declaration statement, the declaration rule splits the assignment of the variable from the declaration of a variable. This is the inverse of Fowler's *replace assignment*, which combines the assignment of a variable with its declaration. The call rule introduces a temporary variable for the value of the expression; this is the inverse of Fowler's *inline temp* refactoring, which replaces a temporary variable with its value. Fowler's catalog includes many inverse refactoring, for example, *extract method* and *inline method*. One reason that an inverse to *replace assignment with initialization* is not in Fowler's catalog is that it does not make the code more understandable or readable. However, for purposes of reducing the number of isomorphisms a user must consider before applying transformations, they are applicable.

An important similarity to Fowler's refactorings is that normalizing restructurings do not change the behavior of the code. This permits the application of a test suite after the normalizing restructurings are performed, but before the user's transformation is applied to ensure that the normalizing restructurings did not affect the system's tested behavior. That said, our restructurings are not refactorings as they do not, by nature, make code easier to understand or cleaner.

Negara et al[2] examined several systems to see what types of refactorings are most popular among automated and manual refactorings. They noted that automated refactoring tools are

underused. We believe that part of the reason for this may be attributed to the difficulty in applying transformation tools due, in part, to the high cost of entry for applying a transformation over a code base. Normalizing restructurings address this issue by simplifying the development of transformation rules needed for some maintenance tasks.

Normalizing restructurings are related to term rewriting.[4] Term rewriting is a technique used to simplify expressions to a standard form. In textbook term rewriting, the primary focus is repeated application of simplification rules to an expression until that expression converges; the result of this convergence being the normal form. We do some of the same things with normalizing restructurings. Specifically, our restructurings transform statements to preserve their behavior under the assumption that the target will be moved from its original location. We also store the result of the target in a variable and replace the target in its original location with that same variable. The effect is akin to term rewriting in that the target is replaced with a single variable using (rewrite) rules for how different statements should be handled given that target (rules for targets within if statements, targets in function call arguments, targets within return values, etc). In compilers, term rewriting normally simplifies statements into less syntactically heavy forms that can be used to generate assembly or binary. Normalizing restructurings are generally not concerned with reducing the size of a target but simply removing it from its context and standardizing its syntax to make it easier to match and apply transformations.

Normalizing restructurings are analogous to a set of term rewriting rules catered specifically to address concerns with program transformation as it applies to software maintenance. Their relation to term rewriting is primarily in the principle of preserving equivalent semantics between targets after some transformation.

Ideas similar to normalizing restructurings have been explored and discussed in the past. One early example is Malton,[5] who briefly discuses normalizations as the act of reducing the translation space and, hence, a step to prepare code for easier translation to the target. Work performed by Ceccato et al[6] to eliminate go-tos in legacy Java used a related technique to match patterns of go-to usage and map them to more well-define, less error-prone looping and branching constructs, hence normalizing the syntax of various go-to patterns to a more standard form. Gama et al[7] introduced a technique to normalize various Javascript class styles into one form to increase maintainability and comprehension. Lawall[8] used a language that leverages a technique similar to normalizing restructurings. This language is called *semantic patch*.[9] Semantic patch shares several features with normalizing restructurings in that it handles isomorphisms by storing their result in a variable and by allowing the user to operate on that variable, much like the technique we have created and described. The difference between normalizing restructurings and semantic patch is that semantic patch has the user substitute variables in for places where isomorphisms may occur so that irrelevant parts of the code are disregarded at transformation time. This is a manual process. Our technique substitutes as an automatic preprocessing step; it works without the user having to specify where isomorphisms occur. That is, an implementation of these restructurings can automatically detect what restructuring to apply based on what code the user is interested in transforming and no further instruction.

Many transformation tools exist for modifying the syntax of source code DMS,[10] TXL,[11,12] RASCAL,[13] ASF + SDF,[14] ELAN,[15] STRATEGO,[16] SPOOFAX,[17–19] and Verbaer et al.[20] Some of these tools are used for heavyweight transformations[21] such as automated refactorings and even language migration. It is possible/likely that some of these tools implement concepts similar or equivalent to the normalizing restructurings described here. However, there is little or no detailed explanation of what restructurings these tools might use within published literature. Our work seems to be the most detailed exploration of a generalized, tool-independent, normalization step that precedes the user's transformation within the literature.

3 | CONSTRUCTING TRANSFORMATION RULES

In previous work,[1] we presented a transformational approach to automate adaptive maintenance tasks on two large-scale commercial software projects at ABB Inc. We addressed a wide range of specific adaptive tasks dealing with modifications in the new operator behavior, template classes that require specialization, fixing iterator variable scope, deprecated string functions, fully qualifying function pointers, and a change to the STL vector of data becoming private.

It is important to note that, after the transformation process, the software systems will be continually evolved and maintained. As such, the transformation process cannot change the style or structure of the existing code drastically. The transformation rules must have minimal effect on style and structure. Any changes to the structure or style often leads to a rejection[19] of the new changes, as described in several studies on past real-world projects.[11,22] Examples have been given[11] 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, developers rejected the proposed changes. Minimal effect on style was a requirement for our work with ABB.

To motivate the importance and need for the restructurings detailed in this article, we present one of the transformation tasks we did in the prior work to better understand the underlying difficulties. One of the systems was undergoing an adaptive maintenance task to adapt to a new version of the compiler in response to changes in the C++ standard (ie, Visual Studio 2003 to Visual Studio 2005). One particular change we addressed affected the semantics of the operator `new`. It was common practice to directly call the operator new in the initialization of a variable declaration, seen as follows:

```
Type *ptr = new Type;
```

Earlier versions of the language standard had new return 0 in the case of a memory error. Error checking was conducted by simply checking the result for 0 and calling an error handler if necessary. The behavior of the `new` operator was changed in the language standard to throwing an exception when memory allocation issues occurred. There are many ways to address this issue and the developers decided to fix the problem by wrapping the call in a try/catch block, preserving the original behavior and preventing any unhandled exception errors from crashing the system. The resulting fix for the above example is

```
Type *ptr;
try { ptr = new Type; }
catch (...) { ptr = 0; }.
```

In developing the transformation rules to address this change, the specific statements that include a call to the operator new must be identified. The expression that calls the operator new must then be isolated in a try/catch block. Finally, the result must be tied into the original statement.

First, it is necessary to search the code base for uses of the operator new to determine all of the different syntactic situations in which it was used. We identified 12 different situations for the use of `new` in the source code to be transformed. These included uses of `new` inside expression statements, the conditions of while loops and if statements, and in return statements. Each of these distinct situations required different processing for the transformation rule. 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.

Once all of the different syntax of the `new` operators in the source are found, replacement code can be constructed. For cases where `new` is used in a variable declaration, the original declaration is copied, but without the initialization. The next part of the transformation creates the appropriate try/catch block.

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

```
if ((ptr = new Type) == 0) {}
```

In this case, the required transformation must move the assignment of the variable outside the if statement, wrap it in a try-catch block, and then compare the variable to 0 inside the if statement's condition. In the case of a `new` operator in the condition of a while statement, the assignment with the expression will also be inserted at the end of the block of the while statement (although there are other ways to solve this case). A block will also need to be created if the while statement did not originally have one.

In general, we found that almost all of the syntactic issues involve an expression. In these cases, one cannot use a simple transformation. Rather, we need a rule specifically constructed for each situation. Because of the large number of different usages of new in the source code, the complete transformation actually comprised 12 different special cases needed for this code base. This transformation example, while seemingly simple, actually required a large amount of effort to develop. The different syntactic situations required us to build specialized rules for each. Our work here aims to bypass the need to develop separate rules by first removing the different syntactic situations, resulting in a single transformation based on a standard form.

4 | STANDARD FORMS

As seen in the previous section, a major difficulty in developing transformations is finding all the different syntactic situations in which the transformation must work. Here, instead of identifying all the special cases, we take a different approach and convert each syntactic situation into one simplified standard form. To understand how this can be done, we need to examine the language syntax and grammar.

There are often many ways to express the same underlying semantics with different syntax. We are only interested in situations that are syntactically isomorphic. A *syntactic isomorphism*[8] is a set of program statements that are semantically identical but syntactically different. If 2 or more statements or expressions are syntactically isomorphic, then they may freely replace one another without changing the behavior of the code. For example, the following are isomorphic in terms of the if statement and have the same semantics:

```
if (x == y){}
```

and

```
bool var = x == y; if (var){}
```

Let us now revisit the adaptive change involving the `new` operator, as the details of this change present a good example of the problems encountered in even this fairly simple transformation task. The typical usage being transformed is something like the following:

```
Type *ptr = new type;
```

That is, the use of a `new` operator within a declaration statement. However, there are several different syntactic situations where `new` can occur, not just in a declaration statement. Each must be addressed. In our experience, identifying all the different possible syntactic usages of a given construct and developing the specific transformations can be fairly complicated and error prone.[1] The following is a partial list of different syntactic uses of the `new` operator that we have generalized from the experiment preformed by Collard et al[1]:

```
ptr = new int[n];
foo (new int[n]);
return new int[n];
Type obj(new int[n]);
int* tbl[23] = {new int[10], new int[10]};
while ((ptr = new int[n]) != 0) {...}
```

We propose a set of standard forms for differing statement/expression combinations in C++. In the following subsections, we will define what we mean by standard forms and talk about how we created a set of standard forms.

4.1 | Definition of standard forms

As stated previously, one challenge to transformation is working with many ways to express a particular behavior; many expressions are semantically equivalent but syntactically divergent. To make matters worse, these expressions may appear in many types of statements: for-loops, while-loops, if statements, and so on. Restructuring these varying syntaxes yields a standard form. That is, we translate a given set of these statement/expression combinations (eg, statement (expression), where some statements can have multiple expressions—for loops, for example) into a syntax that is semantically equivalent to their original format and syntactically consistent in every location touched by the translation. A standard form does 2 things:

1. Allows transformations on standard forms to safely disregard various syntactic forms; the transformation no longer needs to know whether the expression is the condition of a while loop, target of a return statement, and so on.
2. Allows transformations on standard forms to safely disregard varying ways of expressing the same behavior; all expressions

with equivalent behavior and varying syntax are transformed to have the same syntax.

In this article, we propose a set of standard forms. These standard forms simplify the construction of transformations by translating statement/expression combinations into a single, consistent syntax that a user's transformation is then applied to. These standard forms were chosen by us based on previous experience[1] and knowledge of the C ++ programming language. The proposed standard forms are in no way the only choice to be made; many of them can be structured differently. Further, we do not assert that we have formulated a full set of these standard forms for all possible situations; it is possible, perhaps even likely, that we have missed several situations for which a standard form should be created. What we present here is two fold: (1) a set of standard forms we picked because of the positive shortening effect they had on transformations we wrote in previous work and their generalizability to other tasks, and (2) the processes and techniques we developed for generating standard forms. In general, we use the following guidelines to construct these standard forms:

- A standard form is consistent; every syntactic variant of a code segment is translated to have the same syntax. This new syntax is semantically identical to the original.
- A standard form is well defined; the documentation of the standard form gives a precise definition of what each syntactic variation of a code segment will look like after it is translated to the given standard form. This is so developers can easily take advantage of the standard form by simply looking at its definition. This has the bonus of allowing developers to decide whether a given standard form is right for their problem; whether it will simplify the user's transformation.

We see 2 potential situations arising in practice: The first is that one of the standard forms defined in this article fits both guidelines (consistent, well defined) and can be applied to the problem. The alternative is that none of the standard forms fits the problem and the developer constructs their own standard form to simplify their transformation task. That is, we present and define a technique that can be extended and refined through application and further research.

4.2 | Identifying standard forms

The process of creating standard forms required a rigorous exploration of open source software systems. The first step is to enumerate statement/expression combinations and their isomorphisms. That is, we took common types of statements in C++ (for, if, while, etc) and looked at 2 things: (1) the types of syntax occurs within those statements and (2) a means to restructure the statement into a new syntax with equivalent semantics. From there, using the guidelines mentioned previously, we created a standard form.

A base set was determined and then implemented as a tool for evaluation purposes. We discuss the details of this tool in Section 8; we constructed the tool to evaluate the technique and it is not meant to be used outside this purpose. We also applied our restructurings to a selection of open source systems and then compiled and ran unit tests for these systems. Using this method, we found some corner cases that were originally missed. These corner cases came in the form of compile time errors that showed up throughout the code base. That is, after the application of the restructurings, the code was compiled, and resulting errors uncovered omissions in the restructurings. In some cases, a subset of the standard forms did not properly handle specific situations; we updated the erroneous standard form to handle the given case more generally. In other cases, we found syntax for which we needed to create an entirely new standard form. For example, for loops require several standard forms to properly handle all variations. As the more difficult cases were resolved, we gained a better understanding of how to approach the creation of a standard form.

The set of standard forms we present in this work may not be a complete catalogue. That is, we have enumerated those standard forms that are helpful for our applications to transformation problems. It is likely that other problems (we discuss one in Section 7.3) may require standard forms not described here. In summary, the restructurings and their corresponding standard form were created via a process that combined manual and tool-based empirical exploration supported by the formal grammar of the language. The use of the tool allowed us to uncover and document restructurings that were not originally obvious.

5 | NORMALIZING RESTRUCTURINGS

In this section, we define and present the set of restructurings and standard forms specifically for C++. Users selectively apply these restructurings to a segment of code to produce a standard form. We want to stress that, in practice, the restructurings are applied to only those parts of the source code that will be the target of the user's transformation rule (eg, fixing the `new` operator). This selective application reduces and localizes the effect of code modifications so that they are restricted only to the specific maintenance task.

The collective set of restructurings are called *normalizing restructurings*. They are akin to term rewriting in compiler terminology, which is used to simplify program statements into equivalent forms for optimization and creation of executables. Normalizing restructurings are applied before a user-defined transformation; they translate the source code into a standard form. The goal of normalizing restructurings is to make the user's transformation less complex by reducing the number of syntax variations it must address; it only needs to be applied to the standard form.

In the case of the various syntactic situations of the `new` operator, we apply our normalizing restructurings to simplify each statement to standard form. That is, one format that only needs one (much simpler) transformation rule (for the adaptive change mentioned previously). Examples of restructurings are shown in Table 1. The first row corresponds to normalizing a declaration statement (eg, `type id = expr`). The latter two of these restructurings are examples of normalizing a while statement and a return statement, respectively.

In these examples, we use the restructurings to simplify each different usage of the new operator into a standard form. Because of the simplifications, we can apply the user's transformation rule to the standard form of the statements containing `new` instead of their original form. The user's transformation can safely disregard the original statements `new` was found within (while, for, etc) and may additionally disregard

TABLE 1 Examples of restructuring calls to the new operator that occur in different syntactic situations

Original	Restructured	Transformed

`int *ptr = new int[n];`	`int *ptr;`
	`ptr = new int[n];`
		`int *ptr;`
		`try { ptr = new int[n]; }`
		`catch (...) { ptr = 0; }`
		
		`ptr = new int[n];`

`int *ptr;`
`while ((ptr = new int[n]) != 0)`
`{`
`  //code`
`}`
	`int *ptr;`
	`while (true) {`
	`  ptr = new int[n];`
	`  bool var1 = ptr != 0`
	`  if(!var1){`
	`  break;`
	`  }`
	`  //code`
	`}`
		`int *ptr;`
		`while (true) {`
		`  try { ptr = new int[n]; }`
		`  catch (...) { ptr = 0; }`
		`  bool var1 = ptr != 0`
		`  if(!var1){`
		`  break;`
		`  }`
		`  //code`
		`}`

`return new int[n];`	`int *ptr;`
	`ptr = new int[n];`
	`return ptr;`
		`int *ptr;`
		`try { ptr = new int[n]; }`
		`catch (...) { ptr = 0; }`
		`return ptr;`

For all situations, the restructured code results in a single expression for the user-defined transformation to be applied (shown in bold).

differing ways to express the behavior of new (because we put them all into a single format). To complete our example, the statement

```
ptr = new int[n]
```

needs to be targeted by the user's transformation in each of the examples we give above to produce the following:

```
try { ptr = new int[n]; }
catch (...) { ptr = 0; }
```

This completes the full transformation. Notice that the user's transformation need only match that single form instead of the many forms in the examples above. It may also safely disregard the fact that new is present in different types of statements, some of which require more rigorous treatment than the one our standard form requires. Using the standard form, a user will only need to write a transformation to target 1 syntax. That is, a single, simple transformation rule as opposed to 3 (in this example).

Of course, restructuring rules for all syntactic elements must be defined. We define the restructurings as a transformation:

$$\tau : S \Rightarrow S'$$

where the antecedent, S, is a syntactic source code element and the consequence, S', is its corresponding standard form. The definitions we provide recursively apply τ until the code is completely normalized.

5.1 | Normalizing

We define the transformation rules for normalization in Table 2 through Table 4. In the tables, we note the target in bold. The *target* is the code that the user's transformation will manipulate. Targets are either expressions or declaration statements in C++. The normalization process involves creating a temporary variable to store the value of the

TABLE 2 Declaration, return, calls, constructor, and metafunction syntactic and their standard forms

τ-Rule	Original form (S)	Standard form $\tau(S) \Rightarrow S'$
Declaration	$\text{Type}_1\ \text{id}_1$ = **expr**	$\text{Type}_1\ \text{id}_1$; $\text{id}_1 = \tau$(**expr**)
Call	$\text{func}(\text{expr}_1, \ldots, \textbf{expr}_k, \ldots, \text{expr}_n)$	$\text{type}_n\ \text{id}_k = \tau(\textbf{expr}_k)$; $\text{func}(\text{expr}_1, \ldots, \text{id}_k, \ldots, \text{expr}_n)$
Nested call	$\text{func}_1(\text{func}_2(\ldots(\textbf{func}_k(\ldots))))$	$\text{type}\ \text{id}_k = \tau(\textbf{func}_k(\ldots))$; $\text{func}_1(\text{func}_2(\ldots(\text{id}_k(\ldots))))$
Return	return **expr**;	$\text{type}_1\ \text{id}_1 = \tau(\text{expr})$; return id_1;

Constructor initializer	constructor():
	$\text{member}_1(\text{expr}_1)$,
	…,
	$\text{member}_2(\textbf{expr}_k)$,
	…,
	$\text{member}_n(\text{expr}_n)$
	{
	//code
	}
		constructor():
		$\text{member}_1(\text{expr}_1)$,
		…,
		$\text{member}_n(\text{expr}_n)$
		{
		$\text{type}\ \text{id}_k = \tau(\textbf{expr}_k)$;
		$\text{member}_k = \text{id}_k$;
		//code
		}

Metafunction	typedef typename Metafunc < $\text{metafunction}_1 < T>$, …, $\textbf{metafunction}_k < T>$, …, $\text{metafunction}_n < T>>$::type resType;	typedef τ ($\textbf{metafunction}_k < T>$) Type_k; typedef typename Metafunc < $\text{metafunction}_1 < T>$, …, Type_k, …, $\text{metafunction}_n < T>>$::type resType;

The statement that contains the target is modified from the original form to the standard form. The expression, or statement, that contains the target is shown in bold (eg, **expr**).

TABLE 3 If statements and their standard form

τ-Rule	Original form (*S*)	Standard formτ (S) ⇒ *S′*

If	`if(cond){`
	`  //code`
	`}else{`
	`  //elsecode`
	`}`
		`{`
		`  type1 id1 = τ(cond);`
		`  if(id1){`
		`  //code`
		`  }else{`
		`  //elsecode`
		`  }`
		`}`

Else-if	`if(expr){`
	`  //code`
	`}else if(cond){`
	`  //elif code`
	`}else if(expr2){`
	`  //more elif code`
	`}`
	`…`
	`else if(exprn){`
	`  //final elif code`
	`}`
		`if(expr){`
		`  //code`
		`}else{`
		`  type1 id1 = τ(cond);`
		`  if(id1){`
		`  //elif code`
		`  }else if(expr2){`
		`  //more elif code`
		`  }`
		`  …`
		`  else if(exprn){`
		`  //final elif code`
		`  }`
		`}`

target and replacing the usage of the target with this variable. This preserves the semantics but normalizes the syntax.

We recursively apply the rules in Tables 2–4 until the target is normalized. Initially, we select an antecedent that is a parent of the target and is a statement in the grammar (eg, declaration statement, if statement, etc). This is the syntactic category that needs to be normalized in context of the target. One of the restructuring rules in Tables 2–4 can be directly applied to restructure this part of the code. Further restructuring may be necessary if the target is the antecedent of one of the rules or a child expression within an antecedent. For example, if the target is a condition of a while-loop, then the while-loop rule is applied. If the target is only part of the condition expression, then further rule applications are required to completely normalize the code.

If further restructuring is necessary, it implies that the target is an expression, a declaration statement, or a child of an expression or declaration statement. The first 3 rules in Table 2 cover all these situations. These rules are reclusively applied until the target is normalized. The call rule and the nested-call rule cover all expressions if all operators are written in prefix notation. We omit the infix versions of these rules for brevity and without loss of generalization.

If the target is a subexpression, we apply τ to just that subexpression. Take the following example code and assume the target is `func()`.

```
foo(bar(), baz(func()))
```

We apply the nested-call rule that results in the standard form:

```
TYPE id = func();
foo(bar(), baz(id));
```

It is important to note that choice of the target changes the way we normalize the syntax. This is evident in the different rules for restructuring for-loops. If the target is the initializer, a block is required to contain the scope of the temporary variable generated by the for-init rule. If the target is the condition, then no block is required.

The normalization process is iteratively applied until all targets are normalized. That is, if a statement has multiple instances of a target, each target is normalized separately. For example, take the following code example:

```
foo(new int[n], new char[10]);
```

In the case of changing the new operator, as discussed previously, then we need to apply the call rule twice: once for the first occurrence of new and then again for the second occurrence.

5.2 | Improving readability of resulting syntax

During applications of the restructuring rules, we attempt to simplify the resulting syntax to improve readability. There are general situations in which the resulting syntax can be simplified. One example of this is removing intermediate temporary variables. The implementation can address these as special cases rather than as (complicated) generalized rules. The cases are easily detected and the simplifications are straightforward. For example, take the for-loop as follows:

```
for(Handle* handle = baseHandle(); handle!=0; handle = getHandle()){
  //code
}
```

TABLE 4 Loop statements and their standard form

τ-Rule	Original form (*S*)	Standard formτ (S) ⇒ *S′*

While	`while(cond){`
	`  //code`
	`}`
		`while(true){`
		`  type1 id1 = τ(cond);`
		`  if(!id1) break;`
		`  //code`
		`}`

Do-while	`do{`
	`  //code`
	`}while(cond);`
		`while(true){`
		`  //code`
		`  type1 id1 = τ(cond)`
		`  if(!id1) break;`
		`}`

For-init	`for(init1,…,tinitn; expr2; expr3){`
	`  // code`
	`}`
		`{`
		`  type1 id1 = τ(init1);`
		`  for(id1,…,tinitn; expr2; expr3){`
		`  // code`
		`  }`
		`}`

For-condition	`for(expr1; cond; expr3){`
	`  // code`
	`}`
		`for(expr1; true; expr3){`
		`  ctype1 cid1 = τ(cond);`
		`  if(!cid1) break;`
		`  //code}`

For-incr	`for(expr1; expr2; tincr1,…,incrn){`
	`  // code`
	`}`
		`for(expr1; expr2; tincr1,…,tincrn−1){`
		`  // code`
		`  typen idn = τ(incrn);`
		`}`

If the target is `getHandle()`, then the normalization to a standard form results in the following:

```
for(Handle* handle = baseHandle(); handle!=0;){
//code
Handle* id = handle = getHandle();
}
```

Obviously, this is not what we want. In the implementation, the generation of temporaries can be restricted in these cases because the variable `handle` already exists. This line can be simplified to the following:

```
handle = getHandle();
```

Another case where rules can generate extra temporary variables is the following:

```
while(foo(bar(baz()))){
//code
}
```

If the target is `bar()`, then normalization to a standard form will obtain this form:

```
while(true){
Type id;
id = bar(baz());
Type id2
id2 = foo(id);
If(!id2){break}
//code
}
```

which generates an extra, unnecessary, intermediate temporary variable. The implementation can easily identify these situations and simplify them as follows:

```
while(true){
Type id;
id = bar(baz());
if(!foo(id)){break}
//code
}
```

These simplifications and others can be conducted as a postprocessing step after normalization as a set of refactorings to improve the readability of resulting code. Alternatively, these simplifications can be integrated into the restructuring process. That is, the generation of temporaries can be conducted lazily, only when it is clear we need one, for example. This would prevent both of the given situations above. For our evaluation tool, we chose the latter approach. It detects these situations as they arise and avoids generating unnecessary code.

5.3 | Example normalizing restructurings for C++

We now provide 2 sets of examples. The first set is aimed at better describing the process of normalizing and will show how τ is applied during the process. The second set of examples forgoes showing how τ is applied in favor of simply showing before/after application of restructurings. We start with our detailed examples and note that the target is in bold. The target is selected arbitrarily because we do not have a specific user-defined transformation. Our first example follows

```
foo1(foo2(foo3(x)));
```

First, we apply the call rule to the antecedent, resulting in the following:

```
Type1 id1;
id1 = τ(foo2(foo3(x)));)
foo1(id1);
```

Because the target is `foo2(foo3(x))`, the code is fully normalized with this step. Applying τ, therefore, does nothing:

```
Type1 id1;
id1 = foo2(foo3(x));
foo1(id1);
```

This results in the standard form. Let us look at a while statement with a declaration in the condition:

```
while(foo(bar())) {//code}
```

Applying the while-rule results in the following:

```
while(true){
if(!τ(foo(bar()))) break;
//code}
```

Because our target is `bar()` and it is a child of the expression we are currently apply τ to, we have to recursively apply τ. The next invocation of τ uses the nested-call rule:

```
while(true){
Type id;
id = bar();
if(!foo(id)) break;
//code
}
```

If, instead of `foo(bar())`, we choose to put `int x = foo(bar())` into the while's condition, then we end up with the following:

```
while(int x = foo(bar())){//code}
```

Applying the while-rule results in the following:

```
while(true){
τ(int x = foo(bar()))
if(!x) break;
//code
}
```

Here, an implementation that lazily generates temporaries avoids creating a variable because *x* already exists. The target is `bar()`, which we apply the nested-call rule to retrieve:

```
while(true){
Type id;
id = bar();
int x = foo(id);
if(!x) break;
//code
}
```

TABLE 5 Examples of the call rule

Original	Restructured

foo(new Type1(), **bar**(getSize()));	`Type1 temp1;`
	`temp1 = bar(getSize());`
	`foo(new TYPE1(), temp1);`

Type object(**new Type1()**, bar(getSize()));	`Type1 *object2;`
	`object2 = new Type1();`
	`Type object(object2, bar(getSize()));`

In the while-loop example in Section 5.2, we discussed an optimization to avoid removing everything within the condition of the if statement. This optimization cannot be applied here, as this example is different. In this case, applying an optimization to keep `int x = foo(id)` in the condition will break behavior of the original code. Again, this is easily detected in an implementation.

We now present a set of examples focused on the resultant standard form rather than how τ is applied. One issue we have avoided so far is the need for type resolution. The technique we present is flexible in that it is independent of implementation; any type resolution tool will suffice. Therefore, for the time being, we assume we can find the type. Of course, our tool for evaluation has to solve this problem, so in Section 8.2 we discuss how we implemented type resolution our evaluation.

Our first example, in Table 5, uses the call rule. In this instance, the function call has multiple arguments. Again, the target is bolded. Because new `Type1()` does not contain code that will be transformed by the user, we do not need to restructure it at all; we disregard it. In the second example, we have a constructor usage where new `Type1()` is the target instead of `bar()`.

Table 6 shows an example of normalizing a constructor's member initialization using the constructor initializer rule. Here, we add to the body of a constructor to deal with the function call. A dependency check may be required if the order of assignment affects the other attributes.

Another valid choice of standard form for the declaration rule is to use `TYPE name = EXPR`. We chose the form seen in the second row of Table 6 because it seems more natural to write. The return rule is in Table 6 shares many similarities with the call rule.

The choice of target will determine the standard form a given syntax is restructured to. Table 7 provides an example. In the first example, we apply the 'if' rule because the `&&` operator (ie, a function call) is the target. In the second example, `Go()` is the target, and so the nested-call rule is used. In the third example, an else-if is added that contains the target. We use the else-if rule in this case. The else-if rule splits the else-if into 2 separate blocks: an else-block and an if-block. If there is multiple else-ifs on the original expression, then further else and if blocks are nested as in Table 2.

Both looping construct examples in Table 8 are treated very similar to one another, where the target is the condition. The condition

TABLE 6 Example of a constructor inititalizer, declaration, and return

Original	Restructured
`Type(): Member1(new Type1()), Member2("String"), …, MemberN("String") {//code}`	`Type(): Member2("String"), …, MemberN("String") { Type1 *object; object = new Type1(); Member1 = object; //code}`
`Type object = otherObject.getWidth(foo());`	`Type object; object = otherObject.getWidth(foo());`
`Type* foo() { return new Type(); }`	`Type* foo() { Type* obj; obj = new Type(); return obj; }`

TABLE 7 Examples of if statement and a nested call

Original	Restructured
`if (Obj.IsEmpty() && Go()){ … }`	`Type temp; temp = Obj.isEmpty() && Go(); if (temp) { … }`
`if (Obj.IsEmpty() && Go()){ … }`	`Type temp; temp = Go(); if (Obj.isEmpty() && temp) { … }`
`if (cond1.MeetCond()) { doSomething(); } else if (cond2.MeetCond()){ doTheOtherthing(); }`	`if (cond1.MeetCond()) { doSomething(); } else { Type temp2; temp2 = cond2.MeetCond(); if (temp2) { doTheOtherThing() } }`

TABLE 8 Examples of loops

Original	Restructured
`while (!success()) { KeepTrying(); }`	`while (true) { Type temp; temp = success(); if(!temp) {break;} KeepTrying(); }`
`for(FLPtr* lp = aList->First(); lp! = nullptr; lp = aList->Next()) { lp->Run(); }`	`for(FLPtr* lp = aList->First(); true; lp = aList->GetNext()) { bool temp; temp = lp! = nullptr; if(!temp) {break;} lp->Run(); }`

TABLE 9 A meta function example

Original	Restructured

`template < typename T >`
`struct foo{`
`typedef typename`
`eval_if <`
`policy < Nested < T>>,`
`extract < Nested < T>>,`
`wrap < Nested < T>`
`>::type type`
`}`
	`template < typename T >`
	`struct foo{`
	`typedef policy < Nested < T> > Type1;`
	`typedef typename`
	`eval_if<`
	`Type1,`
	`extract < Nested < T>,`
	`wrap < Nested < T>`
	`>::type type;}`

expression is moved into the loop and replaced with `true`. Lastly, in Table 9, we present an example of the standard form for metafunctions. This involves taking expressions in the form of template metafunctions and placing them into separate `typedef`s. Notice that it is the same as the call rule except with syntax for templates.

The normalizing restructurings are constructed to have as little effect on the structure and style of the code as possible. Normalizing restructurings are only applied to the code within the scope of intended transformation.

6 | EVALUATION

We evaluate the approach from several perspectives. First, we examine how the transformation rules from Collard et al[1] are simplified using our technique. Second, using an evaluation tool created to apply the restructurings, we provide data about how well our restructurings preserve the semantics of the source code. There are 3 research questions we address using these 2 perspectives:

1. Do restructurings reduce the size and complexity of the transformations defined in our previous work?
2. Given the XSLT scripts and code from Collard et al,[1] will code resulting from a transformation written without restructurings be more readable, less readable, or about the same?
3. Do restructurings preserve observable behavior when applied to real code?

In the following subsections, we answer each of these questions.

6.1 | Simplifying transformation rules

n our previous work on constructing transformation rules,[1] the adaptation for addressing the `new` operator required us to write multiple transformation rules for all 12 of the syntactic variations that were found for this adaptive maintenance task. Each case is addressed with a separate transformation rule implemented as an XSLT template. In total, we developed 175 lines of XSLT code to cover all the syntactic variations (not including helper XSLT functions). In our investigation of normalizing restructurings, we discovered that there are additional syntactic cases that were not covered by those 12 rules. Those cases actually never occurred in the code base we were examining at ABB Inc.

TABLE 10 Number of Lines of XSLT for each statement type

Statement type	LOC of XSLT	No. rules
Return	13	**1**
Declaration statement	16	**1**
Call	23	**3**
Expression statement	28	**3**
While/if	95	**4**
For	100	**4**
Total	175 (275)	**12**

For loops were not included in the original solution. The number seen next to that statement type is an estimate based on a transformation written by the authors. The total at the bottom includes without for loop's count and with for loop's count (in parenthesis).

However, by first applying the normalizing restructurings to the locations where the adaptive maintenance needs to occur, only one single XSLT transformation rule instead of the 12 (or more) is necessary. The single XSLT rule required after application of restructurings is only 23 lines, an order of magnitude less (than the 175). There is clearly a large decrease in cost of writing and testing 1 rule versus 12. These 23 lines are actually very similar algorithmically to one of the simplest transformations in our original 12 rules, the transformation for declaration statements. In Table 10, each statement type, how many lines of XSLT it requires to complete the operator new transformation, and how many rules composed the XSLT to handle the statement is given.

As can be seen in the Table 10, `decl_stmt` and `return` are 2 of the simplest rules in terms of lines of code (they are also algorithmically simple in comparison). The other 10 templates developed in the previous work are quite complex by comparison and very difficult to understand/test because they had to deal with complex language structures. This addresses RQ1; by using our restructurings, we reduce the number of rules needed to solve the transformation problem described in[1] down to one single rule equivalent in LOC to the declaration statement XSLT given in the table. The transformation script after restructurings is an order of magnitude smaller than without the restructurings.

6.2 | Impact on code readability

As stated previously, one of our goals is to preserve programmer's view of the code (formatting, style, etc). Our restructurings purposefully make as few changes as possible to carry out their intended purpose. However, it is clear that the normalizing restructuring, by themselves, may negatively affect the readability of the code (or at least change the original as written by the programmer). We now examine this issue in a bit more depth to answer RQ2.

We again go back to the example in the previous section that addressed adaptation of the new operator. We found that using the normalizing restructurings with a simple 23-line XSLT template produced very similar final syntax formats for the simpler syntactic situations (ie, declarations, expression statement, and if statement) compared with using the larger, more complex 175-line XSLT template.

These are used in the bulk of the transformations. However, there are situations (eg, while-loop) that result in code that may be considered less readable. The primary reason for this stems from the fact that our standard forms are subjective when it comes to their syntax; there are other equivalent standard form syntaxes that can be chosen and may be more desirable to a given user.

To better understand how the normalizing restructurings affect readability, we constructed a small study of how developers perceived the normalized code after the `new` operator transformation. In the survey, we asked several developers at ABB Inc. to rate the two versions of a short block of provided code. In one version (unnormalized), we applied the set transformations previously developed[1] to solve the adaptation of the `new` operator using the 175 lines of XSLT. In the second version, we presented the developer with the code after it was normalized, and we applied the simple 23 line XSLT transformation.

Participants were given an explanation of the adaptive maintenance task, the original code, and the 2 different transformed versions: one transformed without restructurings being applied and one transformed after restructurings were applied. They were asked to rate the readability of the two versions on a scale from 0 to 10 with 0 being unreadable (ie, not the way they would expect to make the change manually) and 10 being very readable (ie, how they would make the change manually).

This maintenance task involved adapting the code to deal with a change in how the new operator in C++ behaved. The `new` operator changed from returning 0 in the case of unavailable memory to throwing an exception. The solution dictated that a try-catch block be added and the pointer value being assigned to 0 if the exception raised. We asked developers to look at an example of our restructurings in several situations: declaration statement, while-loop, and if statement. We gave developers example code using each of the above situations (see Appendix A) to get feedback on how each affects the readability of the resultant code.

In many cases, both approaches produced very similar, or exactly the same, resultant code. We manually inspected the results of both approaches and selected code examples for the survey to represent some of the worst-case scenarios for the normalizing restructurings in terms of readability. We made no changes to the original XSLT script developed in our previous study as it resulted in fairly readable code (based on feedback in the previous study).

Five professional developers at ABB Inc. completed the survey. All of them had 5+ years of general programming experience with the max being 18 and the median being 15. In terms of professional development, all of them have 2+ years of experience with a max of 10 and a median of 6. Although this small sample size does not allow us to generalize the results, it does give us some perspective of developer concerns in the context of the restructurings and their effect in the transformation process.

The results are presented in Table 11, which answers RQ2. We see that the developers preferred the normalized version in the case of the declaration and if statement code examples. However, we found that the normalized version of the code for the while-loop transformation is less well received. The developers gave the normalized version a lower rating on average than the corresponding unnormalized version. This is not entirely surprising; the restructurings for while loops change the largest portion of the code. However, the resulting code from the original approach was not uniformly viewed as readable by all subjects. It seems likely that developers generally see any large change to the original code as jarring.

TABLE 11 Results of survey

Question 1 declaration-stmt	No normalization	With normalization
Subject 1	2	9
Subject 2	7	7
Subject 3	6	8
Subject 4	9	9
Subject 5	8	6
Average	**6.4**	**7.8**
Question 2 while-stmt		
Subject 1	2	10
Subject 2	8	5
Subject 3	8	6
Subject 4	9	3
Subject 5	8	2
Average	**7.0**	**5.2**
Question 3 if-stmt		
Subject 1	4	8
Subject 2	8	6
Subject 3	5	7
Subject 4	3	9
Subject 5	6	8
Average	**5.2**	**7.6**

Using a sliding scale of 0—unreadable to 10—very readable.

As part of the survey, a space was provided to suggest a different transformation than the two given. For the while-loop example, the developers made no suggestions. We took this as an implication that they felt the transformations were reasonable although not as readable as desired. There were suggestions made for other questions but primarily pertained to the nature of the catch block; subjects suggested a more specific exception or would have logged the exception differently. In summary, the answer to RQ2 is that noncomplex restructurings (such as declaration and if statement) do not seem to make code less readable. However, complex restructurings can make changes to code that developers may not prefer. This situation can be addressed by a developer creating their own, custom standard form with a more desirable syntax. If there is no such standard form, the developer needs to consider whether using restructurings to decrease the complexity of their transformation is worth the trade off in readability.

Further study is required to fully grasp how developers perceive restructurings. It may be the case that a developer will turn down a complex standard form and attempt the transformation by hand only to find themselves having to take the same steps as the restructuring. In the end, the standard forms were created purely in the spirit of simplifying the overall maintenance of a system by allowing for cleaner, more easily written transformation scripts to be applied to a given code base. They reduce the transformation script size dramatically, which lowers the barrier of entry for automated transformation techniques.

6.3 | Maintaining observable behavior

We implemented a tool to apply the restructurings solely for this evaluation and to address RQ3. The tool is built using the srcML (srcML.org) infrastructure[24–27] and libxml2 (xmlsoft.org). The srcML format is used for the representation of source code to support transformation via normalizing restructurings.

srcML is an XML format for source code that wraps the source code text with XML elements to identify syntactic structures. This allows for locations in the syntactic structure of source code to be identified via XPath for exploration using queries and constructing transformations for manipulation. For transformation, srcML preserves all source code text and can be used to write identity transformations from code to code. In addition, it can represent code before preprocessing, preserving the actual developer's view of the source code. The srcML infrastructure provides tools to convert source code to the srcML format and back. The tools and the format are lightweight, highly scalable, and robust to programming language variations and errors.

A two-step process is used, as with most transformational approaches. First, we specify the locations at which normalizing restructurings are to be applied (the targets). With the srcML format, these locations in the code can be specified with an XPath expression that uses the srcML markup. For example, to match all if statements that have the new operator in a condition (for C++), the XPath for srcML is

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

Second, given the particular code to be transformed, the normalizing restructurings are applied. The tool takes the syntax and transforms it into a standard form with no user intervention. Once the restructurings are applied to the srcML version of a source code, the srcML toolkit is used to convert the normalized srcML back to plain source code.

The tool must deal with several issues involving code generation that the technique does not directly address. These issues are type resolution, naming of temporary variables, and dealing with name collision to name a few. In the implementation, we put as much effort into taking care of these issues as was required to evaluate the technique; we discuss this more in Section 8.

To validate that the normalizing restructurings do not affect the behavior of a program, we performed the following. We applied all normalizing restructurings automatically to two open source software systems. Both systems included a test suite that ran correctly on the original code. After automatically applying the restructurings, we ran the test suite to validate that there is no change in the observable, runtime behavior of the systems—or at least what is covered by the unit tests.

Although the intended use of normalizing restructurings is to selectively apply them only on the code that will be the target of a user-defined transformation, here we applied the restructurings to the entire code base in any location where one of the rules in Tables 2–4 matched. This provided many more chances for the restructurings to improperly transform code and was done to stress test the technique and to find out if any standard forms needed to be improved or added.

The experiment was conducted as follows:

1. Compile and build candidate system
2. Run system and verify that all unit tests pass
3. Apply normalizing restructurings to all targets in the system
4. Compile and build restructured candidate system
5. Run restructured system and verify that all unit tests pass

With this, we can infer that the post-restructured syntax of the code is at least equivalent to the original to the extent of code covered by the tests. Clearly, unit tests will not completely validate that two version of the code have the same behavior. However, unit tests do demonstrate that the two versions have equivalent *observable behavior* with respect to covered functions. That is, the behavior that developers felt most important to test are covered by the unit tests, and it is this behavior that we assuredly preserve even after our restructurings are applied. Of course, showing that two versions of code have the same behavior for all inputs is in general NP hard.

We used two C++ systems: the Bullet Physics Engine and Ogre3d. The Bullet Physics Engine (bulletphysics.org) provides several features used in games and movies to support physical interaction with simulated environments. Ogre3d (www.ogre3d.org) is a 3-D rendering engine meant to support the building of simulated environments. The version of Bullet we used (bullet-2.81-rev2613) has 117 KLOC and the version of Ogre we used (v1-9) has approximately 123 KLOC.

We selected these systems because each included test suites that ran correctly and can be compiled and built correctly on our hardware. These systems also have large user communities. This gave us some confidence that their test suites cover some reasonable percentage of their feature sets. In addition, both use a wide set of language features and syntax. We felt that they would exercise the normalizing restructurings well.

TABLE 12 Number of normalizing restructurings applied in Ogre.

Normalizing restructuring	Number of applications	Percentage
If	4691	32%
Declaration	4113	28%
Call/inner call	2932	20%
For	1623	11%
Return	619	4%
Else -if	526	4%
While	324	2%
Do while	7	<1%
Total	**14835**	**100%**

Percentage is of the total applied.

TABLE 13 Number of normalizing restructurings applied in Bullet.

Normalizing restructuring	Number of applications	Percentage
Declaration	6108	42%
If	3098	21%
Call/inner call	2797	19%
For	1470	10%
Return	699	5%
While	170	1%
Else-if	155	1%
Do while	32	<1%
Total	**14529**	100%

Percentage is of the total applied.

The number of normalizing restructurings that resulted for Bullet and Ogre is given in Tables 12 and 13. These tables give the number of times each type of normalizing restructuring was applied to the systems. This is a count of actual transformation rules applied to the software to implement the normalizing restructurings. As shown, there were more than 14,000 restructurings done in each of the systems. The application of the declaration, call, and if statement rules makes up the bulk of normalizing restructurings applied in both systems.

Bullet and Ogre each have their own unit test suite. The restructured version of both systems passed all of their respective test suites. That is, the restructured version had equivalent observable behavior to the unrestructured versions. However, this is dependent on the assumption that the test suites covered parts of the code that were actually restructured.

To get a clear picture of the amount of restructured code exercised by the test suites, we ran coverage tools on both systems' test suites. The coverage tool gave us all the functions in the systems that were covered by the test suites down to line granularity. That is, they show us which lines in functions that were covered were part of the execution trace. We then looked at which of these functions were touched by restructurings and examined which lines in those functions were ran to be certain that these lines did have a restructurings applied to them. This gave us the functions that were both run by the tests and restructured.

Table 14 presents a summary of the results of code coverage. We used the coverage tool GCOV (gcc.gnu.org). After running GCOV, we pulled signatures as well as line positions that were listed as having been executed within the .gcov file and then used the C++ tool c++filt to demangle function names. We determined which functions were both covered by the tests and had at least one normalizing restructuring applied within on a line number that had been executed according to gcov. The value in the table is the number of functions we were able to validate. That is, these were definitely covered by a test and definitely had a restructuring applied to them on a line that was executed. Table 15 presents the distribution of restructurings applied to either system and also covered by a test case. We combined the call and inner call rules in this table because in the implementation of the evaluation tool, we normalize all arguments out of every function call whether they are nested or not. So mechanically, both rules perform the same steps here.

In both systems, we see that at least half of the test suite covers functions that are restructured. On the basis of Table 15, we see that all but two of the rules were applied at least once. The two that were not tested are constructor initializer and metafunction. The metafunction rule is more difficult to test on real code as it requires a system that uses metafunctions and has tests for those metafunctions. We have tested it ourselves but have not found an open source system that the metafunction rule can be reasonably applied to. The constructor initializer rule is not used in either system. However, it follows the same steps as other restructurings—namely, remove target expression, generate a new name, and use that name in place of the original expression. Thus, we are confident that a reasonable portion of code was both restructured and executed. While this does not guarantee that the normalizing restructurings did not affect the behavior of the systems, it does give us a reasonable level of confidence that the transformations that make up the restructurings are reasonably sound, thus answering RQ3. Although there may be some effect to behavior that we are unable to detect, the number of functions both covered and restructured is reasonable enough that any problem that presents itself later will most likely only require simple tweaking of the technique rather than sweeping changes.

TABLE 14 Total number of restructurings covered by unit tests in bullet and ogre

System	Number of functions covered by test suite	Number functions covered by tests and restructured	Percentage of covered functions restructured
Bullet	211	131	62%
Ogre	948	500	52%

TABLE 15 Total number of restructurings applied and covered by units tests Bullet and Ogre

Rule	Combined number of applications covered by test suite of either ogre or bullet	Percentage of covered functions restructured
If	609	32%
Declaration	386	21%
Call/inner call	272	14%
For	246	13%
Return	222	12%
Else-if	105	6%
While	38	2%
Do-while	4	<1%
Total	1882	100%

7 | THREATS TO VALIDITY AND TECHNICAL REQUIREMENTS

There are several threats in the way we have evaluated the correctness and effect on readability of the restructurings. We will start with correctness. It is clear that unit tests are not a full-proof way to determine whether behavior between non-restructured and restructured code has changed. Although the behavior tested by these unit tests is assuredly preserved, it is possible that the unit tests do not fully cover all restructured code or execution pathways. The number of systems covered is also a threat. In our view, the addition of more systems is not likely to reveal a case that would render any of our standard forms null. Rather, what may be found are corner cases that need to be handled with slight modification to or a completely separate standard form. That is, given that there were a total of 1882 restructurings covered by the tests, and most restructurings were applied more than 100 times, we believe the standard forms are applicable to most cases and cases where they are not applicable are in the minority.

To determine the effect on readability, we surveyed 5 software researchers at ABB out of the 8 there are in the research department. Given the number of participants, the results are not generalizable and are vulnerable to outliers in the data set. We also do not have a golden set to compare to; no transformed code that is written “perfectly.” Instead, we took code from our previous study because it was

accepted by ABB as correct after transformation. Obviously, not all of the same people that participated in the former research also participated in this survey. Hence, some of the code that was accepted as correct did not get perfect or even high readability scores.

There are several technical issues to implement the normalizing restructurings in practice. We now discuss several of these issues as well as, to a limited extent, how the tool we implemented for evaluation dealt with them.

7.1 | Preprocessor statements and macros

Our implementation does not attempt to apply any sort of transformation to macros. If a macro is encountered, it is skipped. That is, srcML is typically applied to un-preprocessed code and attempts to mark macros as such. One of course can run the preprocessor before converting to srcML. Restructurings in the presences of macros is still somewhat of an open issue.

7.2 | Type inference

The restructurings introduce new names into local scope to store the value of expressions moved out of various statements. We need to be able to infer the type of the expression being assigned to a given variable so that they can be properly declared. In our description of the technique, it is assumed that the types can be determined. A given implementation of normalizing restructurings can solve this problem in many ways. There are tools/language features for type resolution, and many transformation systems may include methods for doing type resolution. In our implementation, we use the C++11 feature `auto`, which does type inference.

7.3 | Pre- and postconditions for restructurings

Variances between language standards also present an issue for our restructurings. For example, some language standards do not specify the order of evaluation for arguments to a function. For this reason, we now provide a set of 3 preconditions and 2 postconditions for the restructurings. These define a set of expectations that are language agnostic; they must hold regardless of the target language's standards in order for the restructurings to behave properly.

The first precondition deals with the case of a one-line block and applies to the following rules: do-while, while, for-init, for-incr, for-condition, and if. Some statement types (if, while, for, etc) do not always require an explicit block `{ }` in the case where they contain only one child statement. The precondition states that a block must be present in the statements these rules apply to. This can be addressed with a simple check and transformation to the code.

The second precondition concerns missing default constructors and applies only to the declaration rule. Many of the restructurings split a declaration statement so that the declaration and definition are separate statements (eg, `obj x = y;` becomes `obj x; x = y;`). This assumes that there is a default constructor, which is not always the case, as some languages do not guarantee generation of a default constructor, such as in C++. This precondition requires that a default constructor be provided for all objects (involved in the transformation) whether it is generated by the compiler or written by a programmer. Alternatively, objects without a default constructor may be avoided or another restructuring that does not cause the use of a copy constructor defined. For our evaluation, we skipped anything without a default constructor by keeping a list of objects with one.

The third precondition concerns argument evaluation ordering and applies to the following rules: call, nested call, and constructor initializer. The order in which arguments are processed in function calls may not be defined in a given language standard. C++ is one of these cases. For example, with the call `f(g(), h())`, the order of evaluation of `g()` and `h()` is not defined by the language, and different compilers can potentially evaluate these in different orders. This precondition states that the evaluation order in cases such as this must be well defined (so that an implementation can properly handle the ordering) or unimportant (so that the restructuring will not affect behavior either way). Essentially, if order does matter, then either the restructuring must not be applied or extra steps to ensure behavior is maintained must be taken.

The first postcondition addresses object visibility and applies to the following rules: call, nested call, and constructor initializer. Object visibility and deallocation are important issues in C++. The nature of restructurings may cause an object's lifetime to change if not careful. For example, temporary objects that are normally deleted when a function call finishes may last longer because the restructuring assigns them to a variable and removes them from their original scope. This postcondition states that if the lifetime of an object must be (ie, requirement of the implementation) tied to its locality, then this must be handled as a postrestructuring transformation or manually.

The second postcondition is concerned with name collisions and applies to all rules. Name collision is when two names in the same scope are identical. Because of the nature of our restructurings, this is a possible scenario. An implementation of the restructurings must take steps to ensure that name collisions are not generated. This postcondition states that, after a restructuring is applied, generated names or names that were moved from their previous scope must not collide with other names. Minimally, a symbol table can be constructed to keep a record of identifiers and their scope. This allows automatic checking before creation of a new variable. For our tool, we only needed to test and see if our normalizing restructurings worked as intended. We chose a naïve approach; we kept track of what names were assigned and appended a randomly generated value to the end of all variables created by the tool. Furthermore, we examined local and surrounding scope for potential name clashes when moving variables from 1 scope to another. Although this allowed the implementation to provide variables with names and avoid collisions, it is unsuitable for use in real systems.

8 | APPLYING RESTRUCTURINGS IN PRACTICE

We now examine different tasks that can benefit from the use of restructurings, namely, transformation, refactoring, and clone synchronization/merging. We provide a short example of each.

8.1 | General transformation

Transformation to a standard form removes syntactical differences between expressions. There are 3 types of statement/expression

combinations that restructurings simplify. The first situation is when the statement is different but the target is the same. Take the following two statements:

```
if( foo() ) {…};
while( foo() ) {…};
```

Here the target is the expression `foo()` in the condition of the if and while. The target is the same in both examples, but to apply a transformation, each of the statements needs to be treated separately (in lieu of using a normalizing restructuring). If the user's transformation will remove `foo()` from the if and while conditions, then the developer needs to write two transformation rules—one to apply the transformation and maintain behavior of the loop and another to do the same for the if statement.

The standard form of these two statements are

```
{Type1 id;
id = foo();
if( id ) {…};
}
```

and

```
while(true) {
Type1 id;
id = foo()
if(!id) break;
…
};
```

Given the standard forms of these statements above, if the user's transformation targets `foo()`, this transformation now does not need to deal with preserving the behavior of if or while; the standard form takes care of it. The user's transformation can also disregard matching the if statement and while statement entirely. It only needs to match the standard form: `id = foo()`, and modify as needed.

The second situation is when the syntax between multiple, semantically equivalent targets is different. The following two lines are examples:

```
Obj* ptr = getHandle(); if(ptr) {…};
if(Obj* ptr = getHandle()) {…}
```

Both conditions have the same semantics, but their syntax diverges. Again, to apply a transformation, two rules are required, one rule for each syntax that needs to be matched. It is possible there are cases where only one rule is required despite differing syntax, but this rule will necessarily have to be more complex to deal with differences between expression syntax.

Third, is when the target and syntactic both differ as follows:

```
Obj* ptr = getHandle(); if(ptr) {…};
// …
while(Obj* ptr = getHandle()) {…};
```

This is a mix of the two former cases and requires two different matches of targets and two different rules for the syntactic situations (ie, 4 transformation rules) to solve properly. Again, potentially there are languages that can solve these in fewer rules, but without restructurings, the rules will generally need to be more complex because of having to handle both varying statement-level behavior and expression-level syntax.

8.2 | Refactoring

We now examine some of Fowler's refactorings[3] as a simple application of restructurings to a maintenance task. To automate (or semi automate) many of Fowler's refactorings, several different syntactic situations need to be addressed. We can normalize these situations first, and then a single, automated transformation can be applied, drastically simplifying the implementation of refactoring tools. Many tools already support these refactorings, of course. The purpose of this example is to show how readily we can solve these issues even in lieu of the host of popular tools that implement the refactorings.

Take the inline temp refactoring.[3] In this refactoring, we replace every instance of a temporary variable with the expression assigned to it. The following is an example in which two variables are declared and passed to a function via a call:

```
Type a = g(x);
Type b;
b = f(x);
foo(a, b);
```

Inline temp takes each parameter and inlines the expression to

```
foo(g(x), f(x));
```

However, we see that there are two different syntaxes for the variable declaration and assignments. Hence, to transform this, either two transformations or one transformation of heightened complexity will be required. Applying normalizing restructurings will convert these to a single consistent syntax as follows:

```
Type a;
a = g(x);
Type b;
b = f(x);
foo(a, b);
```

With the code in this form, we now require only one rule to complete the inline temp refactoring. We match the general syntax of both statements `a = g(x)` and `b = f(x)`, replace them in their corresponding positions within foo, and remove the variables altogether to get the final form using a single transformation:

```
foo(g(x), f(x))
```

Hence, we have completed the inline temp refactoring using restructurings to simplify the construction of the user's transformation rule. Another example is the decompose conditional restructuring,[3] which simplifies complicated conditional. Here a conditional is as follows:

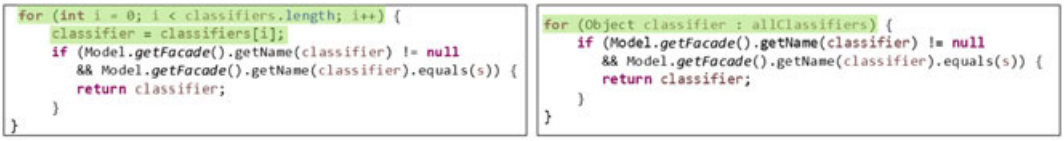
```
if (date.before(START) || date.after(END)) {
charge = winterCharge(quant);
} else { charge = summerCharge (quant); }
```

is extracted and simplified to something as in:

```
if (notSummer(date)) {
charge = winterCharge(quant);
} else { charge = summerCharge (quant); }
```

Normalizing restructurings will convert the conditional syntax (in any construct) into a form that can be easily matched and managed. All that then remains is to move this into a function, or agglomerate the result of all of them into a single:

```
bool cond;
cond = date.before(START) || date.after(END);
if (cond) {…} else {…}
```



(a) For loop implementation (b) Enhanced for loop implementation

FIGURE 1 Example of classic and enhanced for loops that have the same behavior but differing syntax. Clone detection would benefit from the syntaxes being closer

Once again, several tools already automate these refactorings. However, if these tools are not at our disposal, restructurings can be used to greatly simplify their automation as shown above.

Standard forms allow us to construct transformation and syntax matching rules without being concerned about the context or location of the target. The 3 situations provided above are those that the restructurings are able to simplify. Note that in all cases, restructurings assume that the user's transformation will need to move the target out of its context. If this is not the case, then use of restructurings will not simplify the transformation.

8.3 | Clone detection

Given that a part of the goal of restructurings is to simplify the matching part of a transformation (ie, matching a particular syntax to be transformed), restructurings can also have positive effect on any task that needs to match pairs of semantically identical code segments that may not have equivalent syntactic structure. One such task is merging and synchronization of clones. Some clones have been found to be semantically equivalent but syntactically different from one another.[28] It is possible that restructuring these semantically equivalent code segments will improve existing techniques for clone detection and merging. Take the example in Figure 1, which we found in the study of Guo.[29] In this example, the use of classic versus enhanced for-loop to implement the same behavior may cause difficulties when performing clone detection. It is a simple example, but it makes it easy to understand why detecting two semantically equivalent, syntactically dissimilar clones can be a hard problem and why applying a standard form could make it easier to detect such clones.

We have not provided a standard form that solves this example problem because none of our standard forms translate between classic and enhanced for-loops. However, one could easily be provided that translates all looping (classic for, while, enhanced for, go-tos) structures to a similar syntax. This will have a positive effect on making each syntactically different implementation look more similar and possibly make it easier to perform this type of clone detection. We do not explore this idea further in our work here but feel it is a worthwhile direction to help improve work that relies on code clones to solve other maintenance tasks. This is also a good example of how one can customize standard forms and restructurings to cater to a specific problem.

9 | CONCLUSIONS

A set of normalizing restructurings are presented for C++ that translate code into standard forms. This is the first time a comprehensive set of restructurings for simplifying transformations has been presented in the literature. Traditional approaches to program transformation in software maintenance currently require developers to deal with syntactic isomorphisms in the code. That is, they must write a separate transformation for every isomorphism corresponding to the expression(s) they are attempting to transform. Normalizing restructurings alleviate this problem by creating a single syntax of all of these isomorphisms. This not only makes the specification of the transformations themselves easier but also reduces the number of transformations one needs to construct to handle every case of varying syntax.

A tool was constructed to evaluate the restructurings, and an evaluation was conducted on two medium-sized, open source software systems. Both systems included a set of test cases, and these were used to validate that the normalizing restructurings are behavior preserving at least as far as unit tests are concerned. That is, all test cases passed before and after applying the restructurings.

Normalizing restructurings were made to operate within several constraints important to the maintenance and evolution of large software systems.[11,22] These constraints include preserving the programmer's view of the code, preserving comments, and having minimal effect on the structure of the code base. To this end, we conducted a small survey aimed at measuring how receptive professional developers are to transformations made on top of standard forms. The results suggest that using standard forms effects readability in a comparable manner to other approaches to transforming the code. That is, developers were roughly equally receptive to transformations made on top of standard forms compared with those that did not use standard forms. A more in-depth study is required to generalize the results. However, there is most likely a large cost and quality benefit gained by avoiding the development of multiple transformation rules.

The next step is to integrate these normalization restructurings into a program transformation system and do a much larger, more comprehensive evaluation of their effect on productivity, comprehension, and software maintenance. We envision the normalization process to be done in a declarative manner either as a complete pass through the code separate and before the user's transformation or as a prestep during the user's transformation step. The normalizing restructurings are only applied as determined by user's transformation. Again, this will allow developers to focus on constructing a single high quality transformation that apply to many syntactic variations.

ACKNOWLEDGMENTS

This work was supported in part by a grant from the US National Science Foundation (grant no. CNS 13-05292/05217).

APPENDIX

Code for survey questions

	Example code	Transformation 1	Transformation 2

Q1	Key* key = new Key();
	Value* value = new Value();
	if(key && value){
	  func(key, value);
	}
		Key* key;
		Value* value;
		try{
		  key = new Key();
		  value = new Value();
		}catch(exception e){
		  key = nullptr;
		  value = nullptr;
		}
		if(key && value){
		  func(key, value);
		}
			Key* key;
			try{
			  key = new Key();
			}catch(exception e){
			  key = nullptr;
			}
			Value* value;
			try{
			  value = new Value();
			}catch(exception e){
			  value = nullptr;
			}
			if(key && value){
			  func(key, value);
			}

Q2	vector < Node* > vec;
	while(Node* node = new Node()){
	  vec.push_back(node);
	}
		vector < Node* > vec;
		while(true){
		  try{
		  Node* node = new Node();
		  vec.push_back(node);
		}catch(exception e){
		  ...
		  break;
		 }
		}
			vector < Node* > vec;
			while(true){
			  Node* node;
			  try{
			  node = new Node();
			  }catch(exception e){
			  node = nullptr;
			}
			  if(!node){break;}
			  vec.push_back(node);
			}

Q3	if(Handle* handle = new
	  Handle()){
	  handle- > doThing();
	}
		Handle* handle = nlptr;
		try{
		  handle = new Handle();
		  if(handle){
		  handle- > doThing();
		  }catch(exception e){
		  handle = nullptr;
		}
			Handle* handle;
			  try{
			  handle = new Handle();
			}catch(exception e){
			  ...
			  handle = nullptr;
			}
			if(handle){
			  handle- > doThing();
			}

REFERENCES

1. Collard ML, Maletic JI, Robinson BP. A lightweight transformational approach to support large scale adaptive changes. In: 26th IEEE International Conference on Software Maintenance (ICSM'10). Timisoara, Romania; 2010.
2. Negara S, Chen N, Vakilian M, Johnson RE, Dig D. A comparative study of manual and automated refactorings. Presented at the Proceedings of the 27th European conference on Object-Oriented Programming. Montpellier, France; 2013.
3. Fowler M. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley;1999.
4. Baader F, Nipkow T. Term Rewriting and All That. Cambridge University Press; 1998.
5. Malton AJ. The software migration barbell. Presented at the ASERC Workshop on Software Architecture; 2001.
6. Ceccato M, Tonella P, Matteotti C. Goto Elimination Strategies in the Migration of Legacy Code to Java. In: *12th European Conference on Software Maintenance and Reengineering, 2008. CSMR 2008*; 2008:53–62.
7. Gama W, Alalfi MH, Cordy JR, Dean TR. Normalizing object-oriented class styles in JavaScript. Presented at the Proceedings of the 2012 IEEE 14th International Symposium on Web Systems Evolution (WSE); 2012.
8. Lawall JL. Tarantula: killing driver bugs before they hatch. In: *In The 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*; 2005:13–18.
9. Padioleau Y, Hansen R, Lawall JL, Muller G. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. Presented at the Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems. San Jose, California; 2006.
10. Baxter ID, Pidgeon C, Mehlich M. DMS: Program Transformations for Practical Scalable Software Evolution. Presented at the Proceedings of the 26th International Conference on Software Engineering; 2004.
11. Cordy JR. Comprehending Reality—Practical Barriers to Industrial Adoption of Software Maintenance Automation. In: 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR; 2003:196–206.
12. Cordy JR, Dean TR, Malton AJ, Schneider KA. Source transformation in software engineering using the TXL transformation system. *Inform Softw Technol*. 2002;44:827–837.
13. Klint P, Storm TVD, Vinju J. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. Presented at the Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation; 2009.
14. Brand MGJVD, Heering J, Klint P, Olivier PA. Compiling language definitions: the ASF + SDF compiler. *ACM Trans Program Lang Syst*. 2002;24:334–368.
15. Borovanský P, Kirchner C, Kirchner H, Moreau P-E, Vittek M. ELAN: A logical framework based on computational systems. *Electron Notes Theoretic Comp Sci*. 1996;4:35–50.
16. Visser E. Stratego: a language for program transformation based on rewriting strategies. Presented at the Proceedings of the 12th International Conference on Rewriting Techniques and Applications; 2001.
17. Kats LCL, Visser E. The spoofax language workbench: rules for declarative specification of languages and IDEs. *SIGPLAN Not*. 2010;45:444–463.
18. Kats LCL, Visser E. The spoofax language workbench: rules for declarative specification of languages and IDEs. Presented at the Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. Nevada, USA: Reno/Tahoe; 2010.

19. Jonge MD, Visser E. An algorithm for layout preservation in refactoring transformations. In: *Software Language Engineering. Lecture Notes in Computer Science*. Vol.(6940);2012:40–59.

20. Verbaere M, Ettinger R, Moor OD. JunGL: a scripting language for refactoring. Presented at the Proceedings of the 28th international conference on Software engineering, Shanghai, China; 2006.

21. Akers RL, Baxter ID, Mehlich M, Ellis BJ, Luecke KR. Case study: re-engineering C++ component models via automatic program transformation. *Inform Softw Technol*. 2007;49:275–291.

22. Van De Vanter ML. The documentary structure of source code. *Infor Softw Technol*. 2002;44:767–782.

23. Brunel J, Doligez D, Hansen RR, et al. A foundation for flow-based program matching: using temporal logic and model checking. Presented at the Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Savannah, GA, USA; 2009.

24. Collard ML, Decker v, Maletic JI. Lightweight transformation and fact extraction with the srcML toolkit. In *11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11)*. Williamsburg, VA; 2011:10.

25. Collard ML, Maletic JI, Marcus A. Supporting document and data views of source code. In ACM Symposium on Document Engineering (DocEng'02). McLean VA; 2002:34–41.

26. Maletic JI, Collard ML, Marcus A. Source code files as structured documents. In 10th IEEE International Workshop on Program Comprehension (IWPC'02). Paris, France; 2002:289–292.

27. Collard ML, Decker M, Maletic JI.. srcML: an infrastructure for the exploration, analysis, and manipulation of source code. In 29th IEEE International Conference on Software Maintenance (ICSM'13) Tool Demonstration Track. Eindhoven, The Netherlands; 2013:1–4.

28. Juergens E, Deissenboeck F, Hummel B. Code similarities beyond copy & paste. Presented at the Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering; 2010.

29. Guo Q. Mining and analysis of control structure variant clones, Masters, Computer Science, Concordia University; 2015.

How to cite this article: Newman, C. D., Bartman, B., Collard, M. L., and Maletic, J. I. (2016), Simplifying the construction of source code transformations via automatic syntactic restructurings, *J Softw Evol and Proc*, doi: 10.1002/smr.1831