Crossver: a Code Transformation Language for Crosscutting Changes

Kouhei Sakurai
Kanazawa University
k_sakurai@acm.org

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

ABSTRACT

Software evolution sometimes requires changes of module interfaces, which in turn cause *crosscutting changes*, or changes of module clients that are spreading over a program. Such changes on the client-side can be too complicated to be automatically achieved by text replacement and refactoring tools. We propose a code transformation language, called Crossver, for consistently updating code fragments in a program. Crossver offers a source-level pattern sublanguage to express complicated transformation conditions. The patterns are robust against variety among clients thanks to the dataflow-based pattern matcher. In the paper, we overview the design and core semantics of Crossver.

1. INTRODUCTION

Software evolution occasionally involves with changes of interfaces of modules, even if a software system has been carefully designed at the beginning. Coping with such changes is a painful task [6] because an interface change of a module also requires changes of its clients, which can be many, and spread over many modules in a system. We hereafter refer to such changes as *crosscutting changes*.

Change of an interface of a module sometimes causes non-trivial changes on its clients, i.e., its call sites. Those clients often spread over many modules in a system. Changes of the clients have been dealt in an ad-hoc and error-prone manner.

Even though language-level modularization mechanisms, such as aspect-oriented programming (AOP)[5] are promising for crosscutting problems, they are less useful to crosscutting changes. One reason is that we want to change a program itself rather than behavior of the program for the sake of future maintainability. Another reason is that most of those language-level mechanisms rely on runtime information for judging complicated conditions like data dependency.

We propose a novel program transformation language *Crossver*, which can express a set of crosscutting changes as a concise script. Notable features of Crossver are: (1) rather than manipulating intermediate data structures like abstract syntax trees, the programmer describes code transformation rules in a pattern language at the source-program level; (2) the patterns are robust against variations of coding styles as they are matched by using intra-procedural dataflow; and (3) transformation rules can be parameterized so as to subsume differences in the code fragments to be transformed.

```
1  int maxRate = handler.getRequest()
2                    .getUser().getMaxDownloadRate();
---
3  Authority[] maxDownloadRates = handler.getRequest()
4           .getUser().getAuthorities(
5                      TransferRatePermission.class);
6  int maxRate = 0;
7  if (maxDownloadRates.length > 0) {
8     maxRate = ((TransferRatePermission)
9           maxDownloadRates[0]).getMaxDownloadRate();
10 }
```

Figure 1: a changed code fragment in FtpServer: the older (top) and the newer version (bottom)

The rest of the paper is organized as follows. Section 2 shows a concrete example of crosscutting changes from existing open-source software. Section 3 overviews Crossver's design. Section 4 explains semantics of the proposed language. Section 5 concludes the paper.

2. CROSSCUTTING CHANGES

We first show a concrete case of crosscutting changes, one of which is shown in Figure 1. The code fragments above and below the line are respectively appear in older and newer versions of a particular update[1] of Apache FtpServer[2], an open source FTP server program written in Java. The fragments appear in the `execute` method of the `RETR` class, which handles the `get` command, but other classes like `STOR` (which handles the `put` command) are also changed in a very similar but slightly different manner.

2.1 Changes in FtpServer

The change in Figure 1 is caused by an API change of the `getMaxDownloadRate()` method. In the older version, the method should be called on an `User` object, which is

[1]The commit ID is d605b9d, time-stamped on December 30, 2006 in the git repository.
[2]`https://mina.apache.org/ftpserver-project/`

```
1  aspect MaxRateUpdate {
2     context(Handler h, int rate):  withincode(* (RETR||STOR).execute(..))  {
3      User u = h.getRequest().getUser();
4      replace(rate : u) { rate = #getMaxRate(u); } with {
5        Authority[] maxRates = u.getAuthorities(TransferRatePermission.class);
6        rate = 0;
7        if(maxRates.length > 0) rate = proceed((TransferRatePermission) maxRates[0]);
8     }}
9     pointcut getMaxRate(User u):  target(u) &&
10        (call(int User.getMaxDownloadRate()) || call(int User.getMaxUploadRate()));
11 }
```

Figure 2: The definition of `MaxRateUpdate` in Crossver

obtained from a sequence of method calls, namely `getRequest()` and `getUser()`, on a `Handler` object (in the `handler` variable). In the newer version, the method should be called on an `Authority` object obtained from a sequence of method calls, namely `getRequest()`, `getUser()`, and `getAuthorities()`.

2.2 Parameterized changes

The pair of the abovementioned versions has similar yet slightly different changes in more than one modules. The change in `STOR` is similar to the one shown above, but slightly different. It is about the call of `getMaxUploadRate` instead of `getMaxDownloadRate`.

We call such changes *parameterized*; when there is a code fragment that calls either `getMaxDownloadRate()` or `getMaxUploadRate()` (let say m) in a specific way, we transform the fragment into a particular form in which a method name is filled with m. This kind of parameterized changes make transformation languages complicated. However, in a general software evolution scenario, it is not rare that a change has many parameters.

Parameterized changes are not easily realized by using existing refactoring tools like CatchUp![4], RefactroingCrowler[3] and Cider[10]. In order to carry out parameterized changes, a tool needs to recognize a code fragment to be transformed with its parameter positions so as to fill the parameters in the resulted code. However, those automated refactoring tools are not powerful enough to recognize complicated source code fragments.

2.3 Toward scripting crosscutting changes

We examined crosscutting changes in revision histories of several open source software systems, and observed that crosscutting changes cause make a program less maintainable, hard to be reasoned about and error-prone. Based on our examination, we designed a code transformation language that has the following properties.

1. Scriptable: Developers want to manage a crosscutting change as a script, so that they can apply it to recurring change requests, and modify it for different change requests in future.
2. Concise yet robust pattern specification: We proposed a source-level pattern language that matches based on dataflow in order to deal with similar yet slightly different changes. In those change variations, we sometimes found that the source code fragments have the same dataflow structure even though they syntactically differ. For example, two code fragments commonly call `getMaxDownloadRate()` on a `User` object that is obtained through calls of the `getRequest()` and `getUser()` methods. However, at the syntactic level, the one fragment chains those intermediate method calls in one expression whereas the other separates them into different statements interleaved with other operations.

   For the purpose, there are existing techniques such as matching with abstract syntax tree in term-rewriting systems or the AspectJ's pointcuts language. However, naive adoption of them is insufficient for specifying many subsequent expressions in a code fragment. The system needs to directly represent code fragments in concrete code syntax and also to abstract as patterns for matching wide-spreading code fragments. For instance, a call chain of methods may form into separated statements via local variables or composed expressions within a single statement. So, the system needs to specify both cases at once.
3. Parameterized Transformation: The system also needs to abstract out parts in code fragments of crosscutting changes. A change script will have a set of transformation specification which generates an updated code fragment from a specified code fragment. The system needs to support of filling parametric parts for each code fragment. This means that the system refers the collected information while specifying the code fragment, such as data-flow and concrete elements specified by abstract patterns.

There are many program transformation systems like TOM[1], JunGL[12], RASCAL[7] and MetaBorg[2]. They tend to support implementing code analysis or domain specific languages, and therefore, they are not suitable for most developers to use to define scripts of crosscutting changes. For instance, in TOM, JunGL and RASCAL, developers cannot write concrete syntax code for pattern matching. MetaBorg has ability to write concrete code for matching. However the patterns are independent and global, thus it is difficult to write for matching a subsequent expressions like method chains in a same context.

Arcum[11] is a language that has ability to modularize changes themselves. However, it seems less expressive specification for matching widely spreading code fragments.

DeltaJ[8, 9] is also a language for modularizing changes. DeltaJ provides description of changes of existing module interfaces (such as method additions or deletions), however it does not provide transformation within code fragments of methods.

3. THE CROSSVER LANGUAGE

In this section, we proposes Crossver as a new transformation language for Java. Note that Crossver borrowed several keywords and constructs from AspectJ which is an aspect-oriented extension for Java. Figure 2 is an example script definition in Crossver for the crosscutting changes shown in the previous section.

We first present a brief overview of Crossver before explaining each feature of the language.

3.1 Overview of Crossver

We designed Crossver for the developer who needs to cope with crosscutting changes of the design of components in the code base that cause breaks in the code fragments of their client code. The developer writes a script as an aspect, which has expressive descriptions for matching and replacing the breaking code fragments. The developer can automatically obtain upgraded code for the crosscutting change by applying the aspect and the code base to the Crossver transformation engine. For example, the aspect in Figure 2 upgrades the code in Figure 1 from the top to the bottom. The figure shows only the code of the class `RETR`, but the aspect can also upgrade the class `STOR` at once.

The aspect has enumeration of contexts and each context description specifies a sort of code fragments relating to the crosscutting change. The context has the following features.

1. For the specification, it can write patterns in the concrete syntax. Such patterns can match not only exact code based on syntax tree but also code involving same data flow via local variables or compositions of expressions. For instance, the pattern `o.m1().m2()` can match `v=o.m1(); v.m2();` via the local variable `v`.
2. Moreover, it can also match arbitrary or selective parts in the fragments through AspectJ's pointcut descriptions and regular expression's star-operator-like form.
3. A context description involves partial replacements of matched code fragments. The replacements achieve to transform the code fragments with referring to collected information while matching.

Those features promote writing parametric transformation, i.e. more general and semantic specifications and generations with ignoring variance of syntactic code styles.

The rest of the section explains each language constructs followed by the example in Figure 2.

3.2 Definition of contexts of changes

Line 1 declares the aspect `MaxRateUpdate` as the script for the changes.

Changes are described as a context like lines 2–8, with the syntax

$$\texttt{context(}T_i\ v_i,\ldots\texttt{):}\ P\ \texttt{\{}\ S_c \ldots\ \texttt{\}}$$

where T_i v_i are formal parameters, P is AspectJ's pointcuts and S_c are statement patterns. A context describes a set of patterns matching target code fragments, and contains partial replacements. The parameters v_i are variables of the context values and can be used in S_c. Parameters v_i and pointcut P can narrow matching target of S_c. Those parameters match with occurrences of those values with types T_i. In line 2, the `withincode` pointcut specifies `execute` methods in both `RETR` and `STOR` as matching targets.

The statement patterns S_c can accept patterns for matching subsequent statements and expressions in the concrete Java syntax. Line 3 is a statement pattern that matches a sequence of method calls starting from occurrence of a some `Handler` value (described as `h`). It matches subsequent calls `getRequest()` to the value, and the `getUser()` call to the returned value. The returned value of the latter call is defined as the additional (local) value `u` in the context, which can be used in the subsequent matching and transformation.

The statement patterns are expressive. Developers can easily describe intra-procedural dataflow between expressions through composition of them (e.g. method call chains) and local variables in the context.

3.3 Transformation code as partial replacements

Statement patterns can contain descriptions of replacements which partially transform matched code. Lines 4–7 are a replacement in the context, followed by the syntax

$$\texttt{replace(}out\texttt{:}_{opt}\ in\ldots\texttt{)}\ \texttt{\{}\ S_p \ldots\ \texttt{\}}\ \texttt{with}\ \texttt{\{}\ S_r \ldots\ \texttt{\}}$$

where *out* and *in* are references of context variables, S_p are statement patterns, S_r are replacement statements. The `replace` construct specifies replacements of matching S_p subsequent to the previous statement pattern within the enclosing context. Note that the statement patterns S_p in `replace` cannot recursively contain `replace` descriptions.

In line 4, `replace(rate : u)` specifies the variable `rate` as the output value of the context, and it is assigned within both subsequent two sub-blocks. Also, `u` can be used in the same blocks as an input value.

The special form `#`pc`(`e`,...)` refers named pointcut pc with arguments e. This can matches a statement by a composed pointcut pattern. The use of pointcut languages within statement patterns can deal with parameters in code fragments. For instance, `#getMaxRate(u)` of line 4 refers the named pointcut of line 9–10, which matches a call of `getMaxDownloadRate()` or `getMaxUploadRate()`.

The special form `proceed(`e`,...)` appears in the replacement block, refers the original code matched by the statement patterns of `replace` with replacing arguments to e. In line 7, `proceed` calls the matched original call of `getMaxDownloadRate()` or `getMaxDownloadRate()` with an argument obtained from the array `maxRates` instead of `u`. It returns an integer and used as the output `rate`.

```
1  public class RubyFixnum extends RubyInteger {
     ...
2    public RubyFixnum op_plus(RubyFixnum other) {
3      return m_newFixnum(getRuby(),
4            getValue() + other.getValue());
-------------------------------------------------
5    public RubyNumeric op_plus(RubyNumeric other) {
6      if (other instanceof RubyFloat) {
7        return RubyFloat.m_newFloat(getRuby(),
8            getDoubleValue()).op_plus(other);
       ... //omitted code for the big integer (RubyBignum)
9      } else { return m_newFixnum(getRuby(),
10           getValue() + other.getLongValue());
11     }
```

Figure 3: change lines in the JRuby's `RubyFixnum` class: the former version (top) and the latter version (bottom)

```
1  aspect FixnumFloat {
2    context(Ruby r, RubyFixnum other, Object ov,
3            RubyFixnum rv):  op(*) {
4     replace() { ov = other.getValue(); } with {}
5     replace(rv :)  {
6       rv = m_newFixnum(r, anycode(getValue(), ov));
7     } with {
8      if (other instanceof RubyFloat) {
9       rv = #op(RubyFloat.n_newFloat(r,
10           getDoubleValue()), other);
11     } else {
12      rv = proceed(r, other.getLongValue());
13   }}}
14   pointcut op(RubyNumeric oth):
15           withincode(* op_*(..))  && args(oth);
16 }
```

Figure 4: The definition of `FixnumFloat` in Crossver

3.4 Arbitrary code matching for flexible transformations

Crossver has ability to describe flexible transformations thanks to the mechanism of arbitrary code matching. Here, we will show that with another example of crosscutting changes.

Figure 3 shows another example of crosscutting change appeared in JRuby[3], the commit 0c68e63. JRuby is an interpreter for Ruby scripting language, and it has several classes for representing runtime number values such as `RubyFixnum` and `RubyFloat`. `RubyFixnum` represents an integer value and it can be computed through methods `op_plus(..)` and `op_minus(..)` with taking another value as the operand. The figure shows two versions of the `op_plus` method. The newer version supports the adding operation of the integer and an floating point number, which is achieved by the conditional branch regarding `RubyFloat` (lines 6–8). Note that the older code for the operation is partially updated as the call from `other.getValue()` (in line 4) to `other.getLongValue()` (in line 10).

The changes are crosscutting because it happens not only the adding operation (`op_plus`) but also the subtracting operation (`op_minus`). Most parts of the code for the changes are identical but only the operator (`+` and `-`) and a calling method (`op_minus` for `op_plus` in line 8) are different.

[3] `http://www.jruby.org`

Figure 4 shows the aspect in Crossver for Figure 3. A statement for the operation is replaced by the two separated `replace` blocks. As the separation, we want to describe transformation of the call from `getValue()` to `getLongValue()`. The first replacement specifies the call as replaced with an empty block, which means the deletion of the call, and the second replacement specifies the enclosing expressions of the call. This makes the `proceed` call in the second replacement exclude the call for `other.getValue()` but include other operations, the first `getValue()`, the `+` operation and the `m_newFixnum` call.

The included operations in the second replacement are varying with each code fragment in `op_plus` and `op_minus`, thus we need to write patterns for matching them. The operations are the sort of binary operation (`+` and `-`), and not suitable for writing name-based pattern matching like pointcuts `op` in lines 14–15 of Figure 4 (Note that it can successfully match both `op_plus` and `op_minus` by `op_*`). In general, such operations may be plural operations composed by arbitrary expressions.

To achieve matching such operations, we can use the form `anycode(`e`,...)` as a star operator in terms of regular expressions. The `anycode` abstracts a set of expressions and statements in a method, and matches to them. In line 7 of Figure 4, the `anycode` form abstracts the binary operation, with constraints that it takes two values from `getValue()` and `ov` and be passed to `m_newFixnum` as the second argument.

4. SEMANTICS OF CROSSVER

This section presents a brief overview of semantics of Crossver as transformation rules of the term T consisting of the following syntax:

$T ::=$	$K(T, \ldots)$	Expression
	$\mid V$	Variable
	$\mid C$	Constant value
	$\mid$ `replace(`$V : V, \ldots$`)` T `with` T	Replacement

$K ::=$ `if` $\mid$ `while` $\mid$ `call` $\mid$ `assign` $\mid$ `sequence` $\mid$ `anycode` $\mid \ldots$

where V is local variables and C is constant values including symbols. We assume expressions take the SSA (static single assignment) form. The assignment t to v is translated as the term `assign`(t, v) and denoted by v`=`t for short. The term `sequence`(t_1, t_2) is a sequence of two subterms and denoted by t_1`;`t_2 for short.

A transformation is denoted by $t/p \Rightarrow t', S$, where t is the translated term, and p is the pattern term, and t' and S is the generated term and bindings after the transformation, S consists of a sequence of the form $v \mapsto t$.

Figure 5 shows transformation rules of Crossver. Rules for variables (T-Var and T-PVar) means unifying the variable as a binding. Rules for sequences (T-Seq1 and T-Seq2) means traversing the pattern to the term tree.

Rules for constants (T-Cnst) and expressions (T-Exp) means just matching with the concrete instances of the syntax tree. Note the rule for expressions covers matching with a se-

$$v/t \Rightarrow t, \{v \mapsto t\} \text{ (T-Var)} \quad \frac{t \notin V}{t/v \Rightarrow t, \{v \mapsto t\}} \text{ (T-PVar)} \quad \frac{t_1/p \Rightarrow t_1', S}{t_1\,;t_2/p \Rightarrow t_1'\,;t_2, S} \text{ (T-Seq1)} \quad \frac{t_2/p \Rightarrow t_2', S}{t_1\,;t_2/p \Rightarrow t_1\,;t_2', S} \text{ (T-Seq2)}$$

$$\frac{c = c'}{c/c' \Rightarrow c, \emptyset} \text{ (T-Cnst)} \quad \frac{k = k' \quad S_0 = \emptyset \quad S_{i-1} \circ t_i/S_{i-1} \circ p_i \Rightarrow t_i', S_i' \quad S_i = S_{i-1} + S_i'}{k(t_1, \ldots, t_n)/k(p_1, \ldots, p_n) \Rightarrow k(t_1', \ldots, t_n'), S_n} \text{ (T-Exp)}$$

$$\frac{t_1/p \Rightarrow t_1', S \quad t_1'' = t_1'[\oslash/S'.x_j] \quad t_l = S'.x_j; \ldots [a_{l_i}/S.v_i] \ldots \quad t_2' = S \circ t_2[t_l/r_l] \ldots}{t_1/\texttt{replace}(v_0 : v_i, \ldots)\ p\ \texttt{with}\ t_2 \Rightarrow t_1''\,;t_2', S} \text{ (T-Repl)}$$

where all $k_j(\ldots) \in p$ and $k_j \neq \texttt{sequence}$, let $S' = \{x_j \mapsto k_j(\ldots)\}$, and $r_l = \texttt{proceed}_l(a_{l_i}, \ldots) \in p$

$$\frac{k = k' \quad S_0 = \emptyset \quad S_{i-1} \circ t_i/S_{i-1} \circ \texttt{anycode}(p_i, \ldots) \Rightarrow t_i', S_i' \quad S_i = S_{i-1}, S_i' \quad \{p_i \mapsto \_\} \subseteq S_i}{k(t_i, \ldots, t_n)/\texttt{anycode}(p_i, \ldots) \Rightarrow k(t_i', \ldots, t_n'), S_n} \text{ (T-Any1)}$$

$$\frac{t/p_i \Rightarrow t', S}{t/\texttt{anycode}(p_i, \ldots) \Rightarrow t', S} \text{ (T-Any2)} \qquad t/\texttt{anycode}(p_i, \ldots) \Rightarrow t, \emptyset \text{ (T-Any3)} \quad \frac{(t', S) = \mathit{matchPointcuts}(t, n, a_i, \ldots)}{t/\texttt{pointcut}(n, a_i, \ldots) \Rightarrow t', S} \text{ (T-Pc)}$$

Figure 5: Transformation rules

quence of patterns. We denote application of the bindings S to the term t by $S \circ t$, namely, substitution of all v occurred in t with t_v if $\{v \mapsto t_v\} \in S$. $S + S'$ means a composition of two bindings. For the composition, if there exists bindings for same variable in both S and S', it prefers the latter instance in S'. The rule for expressions translates each sub-term step by step, and composes the obtained bindings with the one of the previous step. It derives the bindings obtained at the last step.

The rule for replacements (T-Repl) means a replacement of matching pattern including the support of `proceed` form. The rule first matches the pattern p with t_1, second removes matched all sub-terms, which are associated with x_j in the temporary bindings S', from the transformed term t_1', and gets t_1''. We denote the substitution of the term t with t' within s by $s[t'/t]$, and $\oslash$ means the empty term. $S.v$ means obtaining the associated term for v from S. The rule also constructs the term t_l for each $\texttt{proceed}_l$, as the original term applied by the proceed. The term t_l is composition of matched sub-terms ($S'.x_j$) and substitutions of arguments of the proceed ($[a_{l_i}/S.v_i]$). It is applied as a substitution to the replacement t_2 and gets t_2'. The rule produces a sequence of t_1'' and t_2'.

The three rules for `anycode` supply the feature of arbitrary matching, similar to the star operator in the regular expression. The term `anycode` can be matched with any kind of expressions with recursively down to sub-terms (T-Any1). The matching is limited to the cases of all arguments p_i of `anycode` matches to some sub-term t_i (denoted by $\{p_i \mapsto \_\} \subseteq S_i$). It can also matches any terms whose sub-term p_i matches to the term (T-Any2) or nothing (T-Any3).

The rule for pointcuts (T-Pc) supports pointcut matching. The term `pointcut` takes a name of pointcut declaration n and arguments a_i, and calls *matchPointcuts* which matches the specified pointcuts with arguments to the given term t followed by AspectJ's pointcuts semantics.

5. CONCLUSION

In this paper, we proposed a new transformation language Crossver which can express a set of crosscutting changes as a concise script. With the language, developers can write a set of transformation rules between versions in expressive and robust way. The language supports writing expressive patterns namely, matching and substitution with intra-procedural data-flow, concrete syntax and regular expressions. We presented semantics of Crossver as transformation rules of terms.

We are currently working on development of the implementation of Crossver, which is built on top of existing AspectJ and Eclipse Java compilers. As a future work, we are considering to integrate the language into an existing version control system (e.g. git).

6. REFERENCES

[1] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: piggybacking rewriting on Java. *RTA '07*, pp. 36–47, 2007.

[2] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. *OOPSLA '04*, pp. 365–383, 2004.

[3] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. *ECOOP '06*, pp. 404–428, 2006.

[4] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. *ICSE '05*, pp. 274–283, 2005.

[5] G. Kiczales et al. Aspect-oriented programming. *ECOOP' 97*, pp. 220–242, 1997.

[6] M. Kim et al. An empirical investigation into the role of API-level refactorings during software evolution. *ICSE' 11*, pp. 151–160, 2011.

[7] P. Klint et al. RASCAL: a domain specific Language for source code analysis and manipulation. *SCAM '09*, pp. 168–177, 2009.

[8] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. *SPLC'10*, pp. 77–91, 2010.

[9] S. Schulze, O. Richers, and I. Schaefer. Refactoring delta-oriented software product lines. *AOSD '13*, pp. 73–84, 2013.

[10] M. Shomrat and Y. A. Feldman. Detecting refactored clones. *ECOOP' 13*, pp. 502–526. 2013.

[11] M. Shonle et al. A framework for the checking and refactoring of crosscutting concepts. *TOSEM*, 21(3):15:1–15:47, 2012.

[12] M. Verbaere, R. Ettinger, and O. Moor. JunGL: a scripting language for refactoring. *ICSE '06*, pp. 172–181, 2006.